Languages and Compilers
(SProg og Oversættere)

Lecture 1
Overview of the course and Language processors

Bent Thomsen

Department of Computer Science

Aalborg University

What is the Most Important Open Problem in Computing?

Increasing Programmer Productivity

- Write programs quickly
- Write programs easily
- Write programs correctly

- Why?
  - Decreases development cost
  - Decreases time to market
  - Decreases support cost

How to increase Programmer Productivity?

3 ways of increasing programmer productivity:

1. Process (software engineering)
   - Controlling programmers
   - Good process can yield up to 20% increase
2. Tools (verification, static analysis, program generation)
   - Good tools can yield up to 10% increase
3. Better designed Languages --- the center of the universe!
   - Core abstractions, mechanisms, services, guarantees
   - Affect how programmers approach a task (C *vs.* Haskell)
   - New languages can yield 700% increase

Quicksort in C and Haskell

```c
// To sort array a[] of size n: qsort(a,0,n-1)

void qsort(int a[], int lo, int hi) {
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];

    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    a[hi] = a[l];
    a[l] = p;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

```haskell
qsort [] = []
qsort (x:xs) =
    qsort (filter (< x) xs)
    ++ [x] ++
    qsort (filter (>= x) xs)
```

Programming Languages and Compilers are at the core of Computing

All software is written in a programming language

Learning about compilers will teach you a lot about the programming languages you already know.

Compilers are big – therefore you need to apply all you knowledge of software engineering.

The compiler is the program from which all other programs arise.
Get it wrong and a lot of people will be affected!

What is a Programming Language?

- A set of rules that provides a way of telling a computer what operations to perform.
- A set of rules for communicating an algorithm
- A linguistic framework for describing computations
- Symbols, words, rules of grammar, rules of semantics
  – Syntax and Semantics
  – (Libraries, Frameworks, Patterns and Pragmas)

Why Are There So Many Programming Languages

- Why do some people speak French?
- Programming languages have evolved over time as better ways have been developed to design them.
  - First programming languages were developed in the 1950s
  - Since then thousands of languages have been developed
- Different programming languages are designed for different types of programs.

Levels of Programming Languages

High-level program

```
class Triangle {
   ...
   float surface()
     return b*h/2;
   }
```

Low-level program

```
LOAD r1,b
LOAD r2,h
MUL r1,r2
DIV r1,#2
RET
```

Executable Machine code

```
0001001001000101
0010010011101100
10101101001...
```

Types of Programming Languages

- First Generation Languages

  Machine

  0000 0001 0110 1110
  0100 0000 0001 0010

- Second Generation Languages

  Assembly

  LOAD x
  ADD R1 R2

- Third Generation Languages

  High-level imperative/object oriented

  ```
  public Token scan ( ) {
  while (currentchar == ‘ ’
  || currentchar == ‘\n’)
  {….} }
  ```

  Fortran, Pascal, Ada, C, C++, Java, C#

- Fourth Generation Languages

  Database

  ```
  select fname, lname
  from employee
  where department=‘Sales’
  ```

  SQL

- Fifth Generation Languages

  Functional

  ```
  fact n = if n==0 then 1
  else n*(fact n-1)
  ```

  Logic

  ```
  uncle(X,Y) :- parent(Z,Y), brother(X,Z).
  ```

  Lisp, SML, Haskel, Prolog

Beyond Fifth Generation Languages

- Some talk about
  - Aspect Oriented Programming (Not so much ☺)
  - Agent Oriented Programming
  - Intentional Programming
  - Natural language programming
- Maybe you will invent the next big language

The principal paradigms

- Imperative Programming
  – Fortran, Pascal, C
- Object-Oriented Programming
  – Simula, SmallTalk, C++, Java, C#
- Logic/Declarative Programming
  – Prolog, SQL
- Functional/Applicative Programming
  – Lisp, Scheme, Haskell, SML, F#
- (Aspect Oriented Programming)
  – AspectJ, AspectC#, Aspect.Net
- (Reactive Programming)
  – RxJava, Angular, React, Vue, Functional reactive

The Multi-Paradigm Era

Microsoft fellow Anders Hejlsberg, who heads development on C#, said:

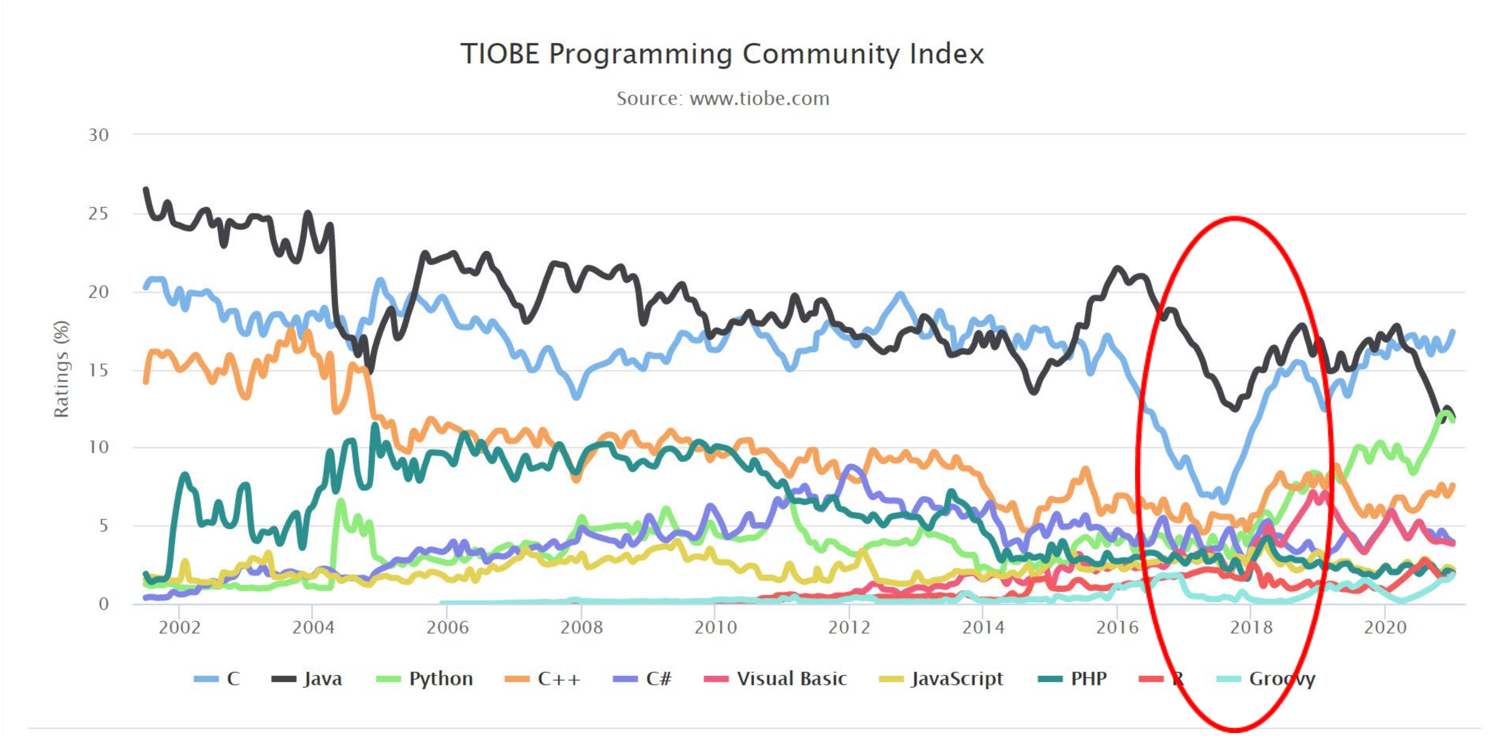
"The taxonomies of programming languages are starting to break down,"

He points to dynamic languages, programming languages, and functional languages.

He said "future languages are going to be an amalgam of all of the above.

If in doubt, take a look at C#

The 10 most popular programming languages



Swift
DART
Erlang
Scala
Lisp
Kotlin
F#
Haskel

https://www.tiobe.com/tiobe-index/

What determines a “good” language

- Formerly: Run-time performance
  - (Computers were more expensive than programmers)
- Now: Life cycle (human) cost is more important
  - Ease of designing, coding
  - Debugging
  - Maintenance
  - Reusability
- FADS
  - A fad is any form of behavior that develops among a large population and is collectively followed enthusiastically for a period of time, generally as a result of the behavior being perceived as popular by one's peers or being deemed "cool" Source Wikipedia

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Evidense Based Programming Language Design

- New direction in PL Resreach (ca. 2005)
  - Use social science techniques
    - Data Mining of repositories or MOC (massive Online Course)
    - Questionaeres
      - E.g. Perl vs. Python (age difference)
      - E.g. ObjectiveC (Most like used in small companies)
  - Use medical science techniques
    - Controlled experiments
      - E.g. Static vs. Dymanic types
    - Placebo effects
      - E.g. Quorum vs. Perl. Vs Randomo
  - Use HCI techniques
    - Eye tracking and Brain Scans
    - (Usability Lab)
    - Discount Method for Programming Language Evaluation
- Actually not that new
  - SmallTalk and Logo designers used observational studies in the 70ies

Programming languages are languages

- But Computer languages lack ambiguity and vagueness
- In English sentences can be ambiguous
  - *I saw the man with a telescope*
    - Who had the telescope?
  - *Take a pinch of salt*
    - How much is a pinch?
- In a programming language a sentence either means one thing or it means nothing

Programming Language Specification

- Why?
  - A communication device between people who need to have a common understanding of the PL:
    - language designer, language implementor, language user
- What to specify?
  - Specify what is a ‘well formed’ program
    - syntax
    - contextual constraints (also called static semantics):
      - scope rules
      - type rules
  - Specify what is the meaning of (well formed) programs
    - semantics (also called runtime semantics)

Programming Language Specification

- Why?
- What to specify?
- How to specify ?
  - Formal specification: use some kind of precisely defined formalism
  - Informal specification: description in English.
  - Usually a mix of both (e.g. Java specification)
    - Syntax => formal specification using CFG
    - Contextual constraints and semantics => informal
    - Formal semantics has been retrofitted though
  - But trend towards more formality (C#, Fortress)
    - fortress.pdf
    - Ecma-334.pdf

Specification of Method invocation in C# according to the ECMA 334 standard

14.5.5 Invocation expressions

An *invocation-expression* is used to invoke a method.

invocation-expression:
    primary-expression (*argument-list$_{opt}$*)

The *primary-expression* of an *invocation-expression* shall be a method group or a value of a *delegate-type*. If the *primary-expression* is a method group, the *invocation-expression* is a method invocation (§14.5.5.1). If the *primary-expression* is a value of a *delegate-type*, the *invocation-expression* is a delegate invocation (§14.5.5.2). If the *primary-expression* is neither a method group nor a value of a *delegate-type*, a compile-time error occurs.

The optional *argument-list* (§14.4.1) provides values or variable references for the parameters of the method.

The result of evaluating an *invocation-expression* is classified as follows:

- If the *invocation-expression* invokes a method or delegate that returns `void`, the result is nothing. An expression that is classified as nothing cannot be an operand of any operator, and is permitted only in the context of a *statement-expression* (§15.6).
- Otherwise, the result is a value of the type returned by the method or delegate.

14.5.5.1 Method invocations

For a method invocation, the *primary-expression* of the *invocation-expression* shall be a method group. The method group identifies the one method to invoke or the set of overloaded methods from which to choose a specific method to invoke. In the latter case, determination of the specific method to invoke is based on the context provided by the types of the arguments in the *argument-list*.

The compile-time processing of a method invocation of the form `M(A)`, where `M` is a method group (possibly including a *type-argument-list*), and `A` is an optional *argument-list*, consists of the following steps:

- The set of candidate methods for the method invocation is constructed. For each method `F` associated with the method group `M`:
  - If `F` is non-generic, `F` is a candidate when:
    - `M` has no type argument list, and
    - `F` is applicable with respect to `A` (§14.4.2.1).
  - If `F` is generic and `M` has no type argument list, `F` is a candidate when:
    - Type inference (§25.6.4) succeeds, inferring a list of type arguments for the call, and
    - Once the inferred type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of `F` satisfy their constraints (§25.7.1), and the parameter list of `F` is applicable with respect to `A` (§14.4.2.1), and
  - If `F` is generic and `M` includes a type argument list, `F` is a candidate when:
    - `F` has the same number of method type parameters as were supplied in the type argument list, and
    - Once the type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of `F` satisfy their constraints (§25.7.1), and the parameter list of `F` is applicable with respect to `A` (§14.4.2.1).
- The set of candidate methods is reduced to contain only methods from the most derived types: For each method `C.F` in the set, where `C` is the type in which the method `F` is declared, all methods declared in a base type of `C` are removed from the set. Furthermore, if `C` is a class type other than `object`, all methods declared in an interface type are removed from the set. [*Note*: This latter rule only has affect when the method group was the result of a member lookup on a type parameter having an effective base class other than object and a non-empty effective interface set (§25.7). *end note*]
- If the resulting set of candidate methods is empty, then no applicable methods exist, and a compile-time error occurs.
- The best method of the set of candidate methods is identified using the overload resolution rules of §14.4.2. If a single best method cannot be identified, the method invocation is ambiguous, and a compile-time error occurs. When performing overload resolution, the parameters of a generic method are considered after substituting the type arguments (supplied or inferred) for the corresponding method type parameters.
- Final validation of the chosen best method is performed:
  - The method is validated in the context of the method group: If the best method is a static method, the method group shall have resulted from a *simple-name* or a *member-access* through a type. If the best method is an instance method, the method group shall have resulted from a *simple-name*, a *member-access* through a variable or value, or a *base-access*. If neither of these requirements is true, a compile-time error occurs.
  - If the best method is a generic method, the type arguments (supplied or inferred) are checked against the constraints (§25.7.1) declared on the generic method. If any type argument does not satisfy the corresponding constraint(s) on the type parameter, a compile-time error occurs.

Once a method has been selected and validated at compile-time by the above steps, the actual run-time invocation is processed according to the rules of function member invocation described in §14.4.3.

[*Note*: The intuitive effect of the resolution rules described above is as follows: To locate the particular method invoked by a method invocation, start with the type indicated by the method invocation and proceed up the inheritance chain until at least one applicable, accessible, non-override method declaration is found. Then perform overload resolution on the set of applicable, accessible, non-override methods declared in that type and invoke the method thus selected. *end note*]

171

Method invocation in Fortress

13.4 Dotted Method Invocations

Syntax:

Primary	::=	*Primary . Id StaticArgs? ParenthesisDelimited*
ParenthesisDelimited	::=	*Parenthesized*
	|	*ArgExpr*
	|	( )
Parenthesized	::=	(*Expr*)
ArgExpr	::=	*TupleExpr*
	|	((*Expr* ,)* *Expr* ...)
TupleExpr	::=	((*Expr* ,)$^+$ *Expr*)

A *dotted method invocation* consists of a subexpression (called the receiver expression), followed by '`.`', followed by an identifier, an optional list of static arguments (described in Chapter 9) and a subexpression (called the *argument expression*). Unlike in function calls (described in Section 13.6), the argument expression must be parenthesized, even if it is not a tuple. There must be no whitespace on the left-hand side of the '`.`' and the left-hand side of the left parenthesis of the argument expression. The receiver expression evaluates to the receiver of the invocation (bound to the self parameter (discussed in Section 10.2) of the method). A method invocation may include explicit instantiations of static parameters but most method invocations do not include them.

The receiver and arguments of a method invocation are each evaluated in parallel in a separate implicit thread (see Section 5.4). After this thread group completes normally, the body of the method is evaluated with the parameter of the method bound to the value of the argument expression (thus evaluation of the body occurs after evaluation of the receiver and arguments in dynamic program order). The value and the type of a dotted method invocation are the value and the type of the method body.

We say that methods or functions (collectively called as *functionals*) may be *applied to* (also "*invoked on*" or "*called with*") an argument. We use "call", "invocation", and "application" interchangeably.

$$[\text{R-M}\textsc{ethod}] \quad \frac{\texttt{object}\ O\ \_\ (\overrightarrow{x{:}\_})\ \_\ \texttt{end} \in p \qquad mbody_p(f[\![\overrightarrow{\tau'}]\!], O[\![\overrightarrow{\tau}]\!]) = \{(\overrightarrow{x'}) \rightarrow e\}}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).f[\![\overrightarrow{\tau'}]\!](\overrightarrow{v'})] \longrightarrow E[\overrightarrow{v}/\overrightarrow{x}][O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x'}]e]}$$

Programming Language Specification

- A Language specification has (at least) three parts
  - Syntax of the language:
    - usually formal in BNF or EBNF + RE for lexems
  - Contextual constraints:
    - scope rules (often written in English, but can be formal)
    - type rules (formal or informal)
  - Semantics:
    - defined by the implementation
    - informal descriptions in English
    - formal using operational or denotational semantics

The Syntax and Semantics course will teach you how to read and write a formal language specification – so pay attention!

Does Syntax matter?

- Syntax is the visible part of a programming language
  - Programming Language designers can waste a lot of time discussing unimportant details of syntax
- The language paradigm is the next most visible part
  - The choice of paradigm, and therefore language, depends on how humans best think about the problem
  - There are no <u>right</u> models of computations – just different models of computations, some more suited for certain classes of problems than others
- The most invisible part is the language semantics
  - Clear semantics usually leads to simple and efficient implementations

- But syntax does matter!
  - Syntax that suggest underlying semantics seems to be important to programmers

Language Processors: What are they?

A programming language processor is any system (software or hardware) that manipulates programs.

Examples:

- Editors
  - Emacs
- Integrated Development Environments
  - Eclipse
  - NetBeans
  - Visual Studio .Net
- Translators (e.g. compiler, assembler, disassembler)
- Interpreters

Interpreters

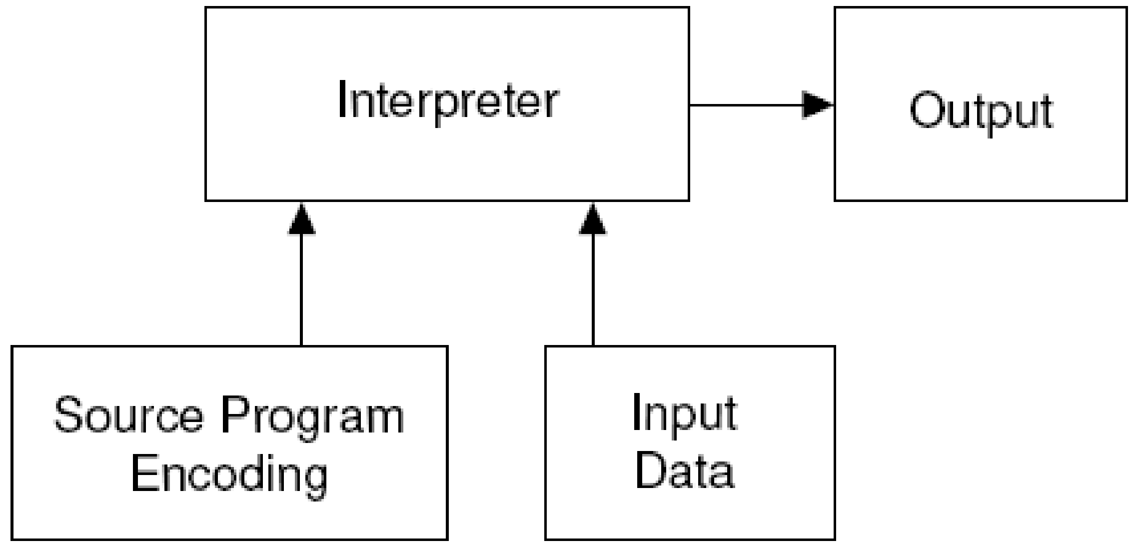
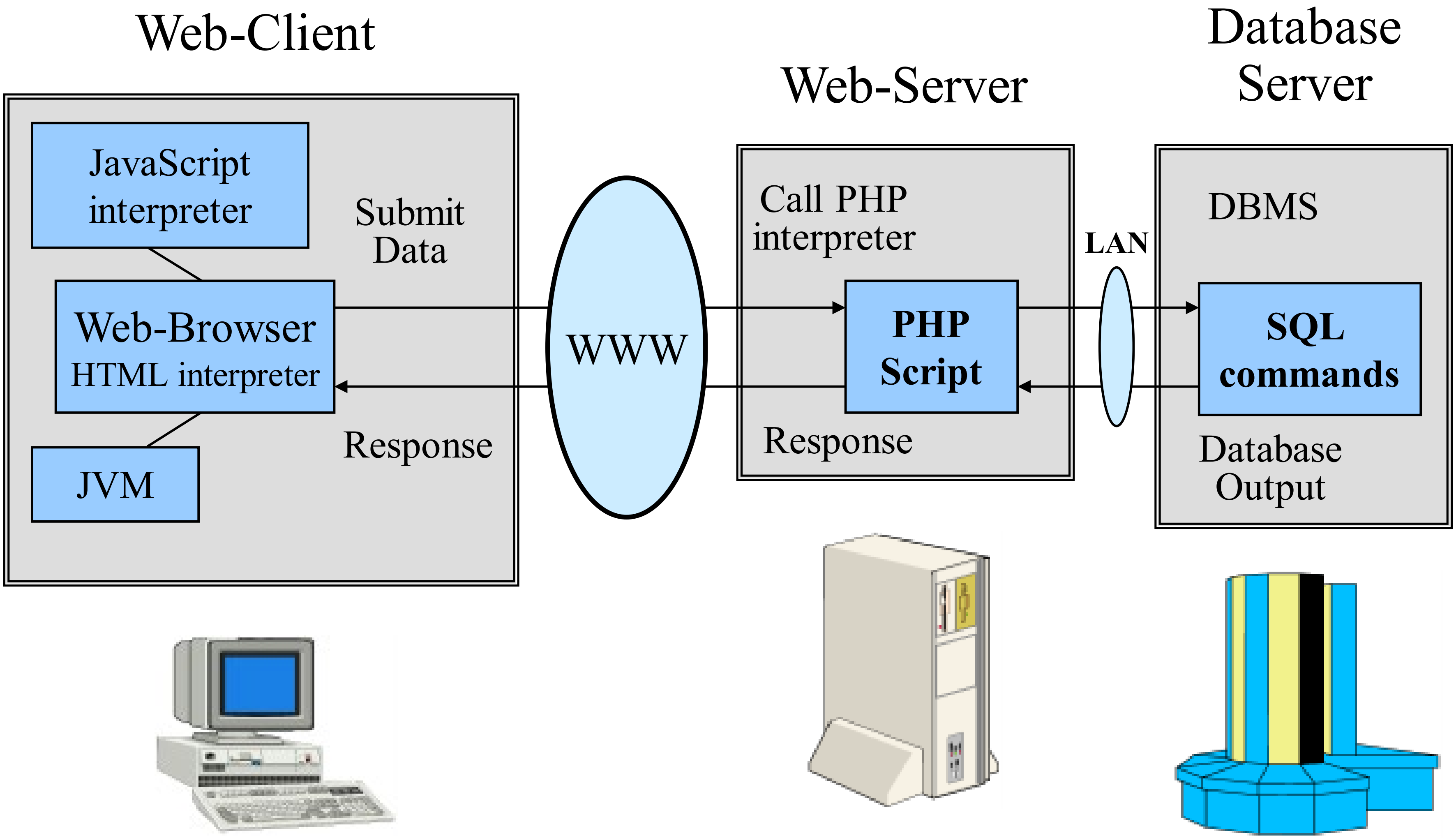


Figure 1.3: An interpreter.

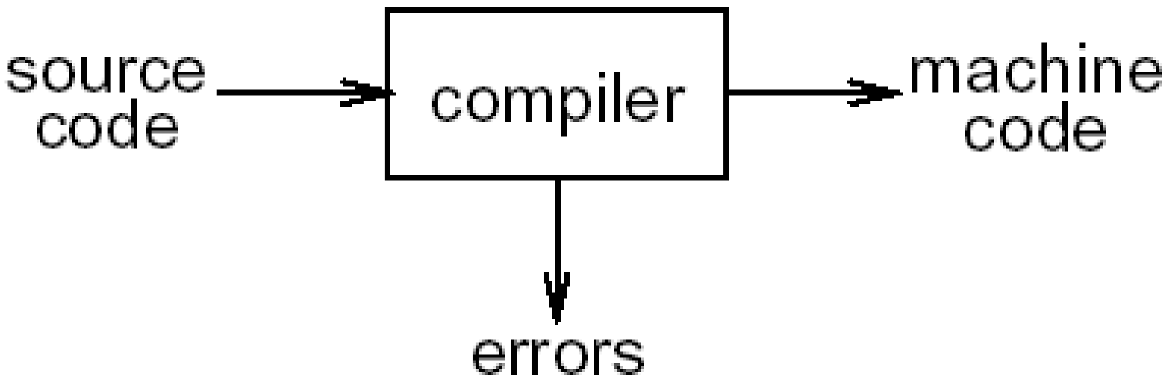
You use lots of interpreters every day!



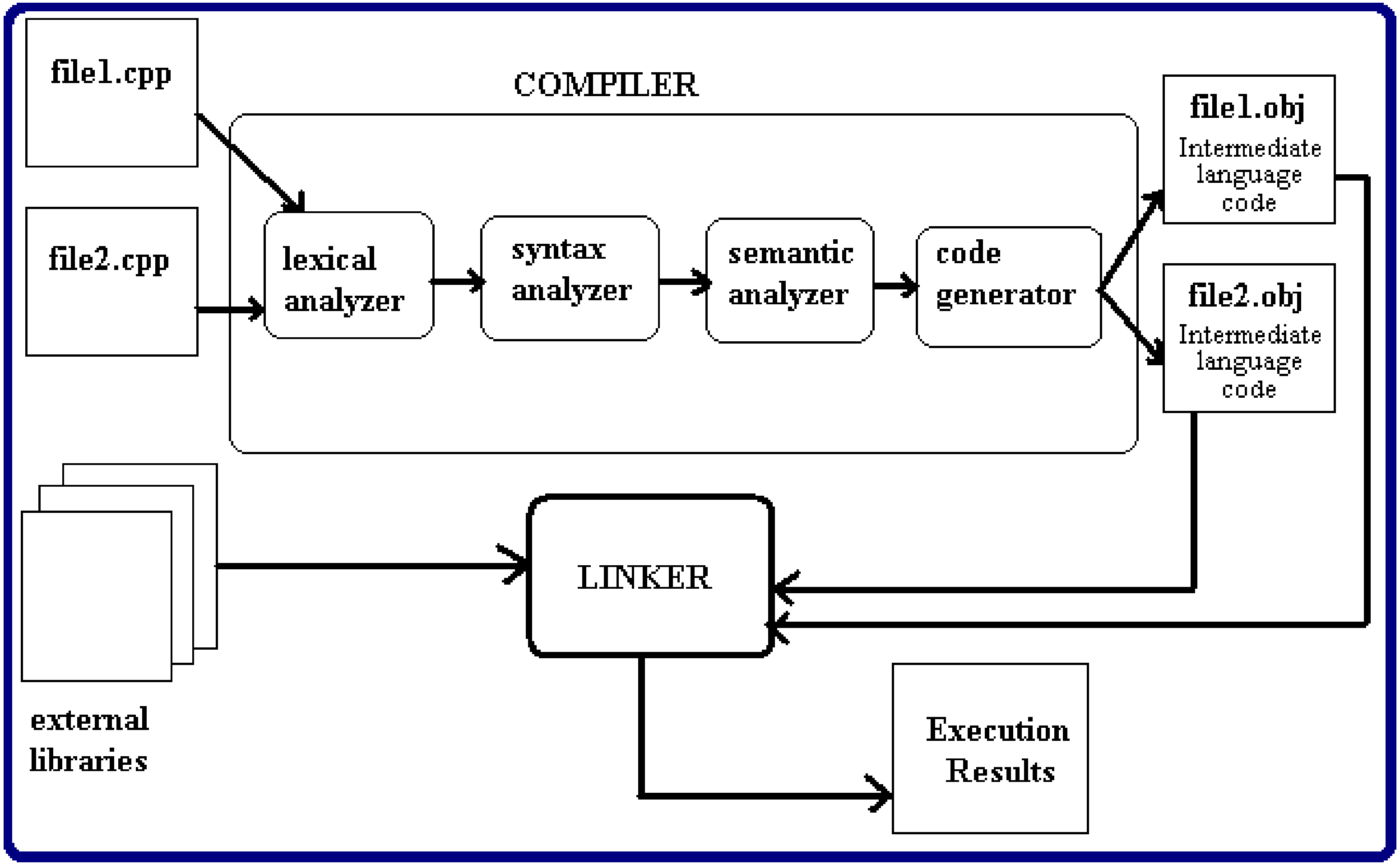
Compilation

- **Compilation** is at least a two-step process, in which the original program (source program) is input to the compiler, and a new program (target program) is output from the compiler. The compilation steps can be visualized as the following.

Compiler (simple view)

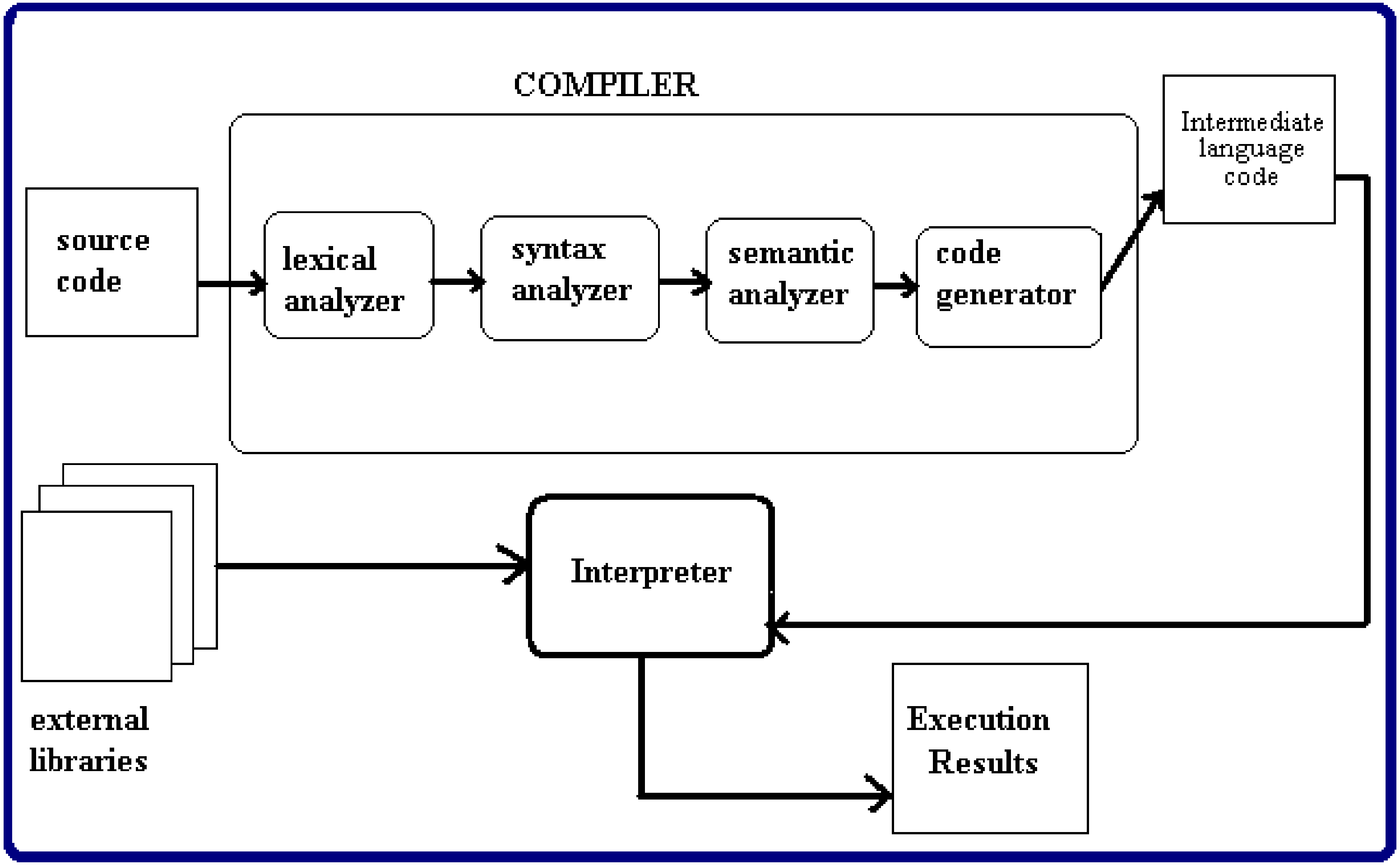


Compiler

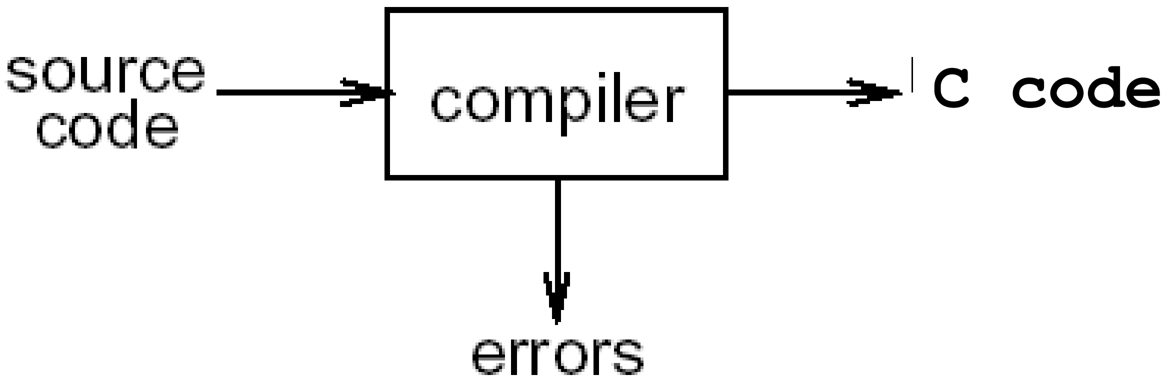


file1.cpp
file2.cpp
COMPILER
lexical analyzer
syntax analyzer
semantic analyzer
code generator
file1.obj
Intermediate language code
file2.obj
Intermediate language code
external libraries
LINKER
Execution Results

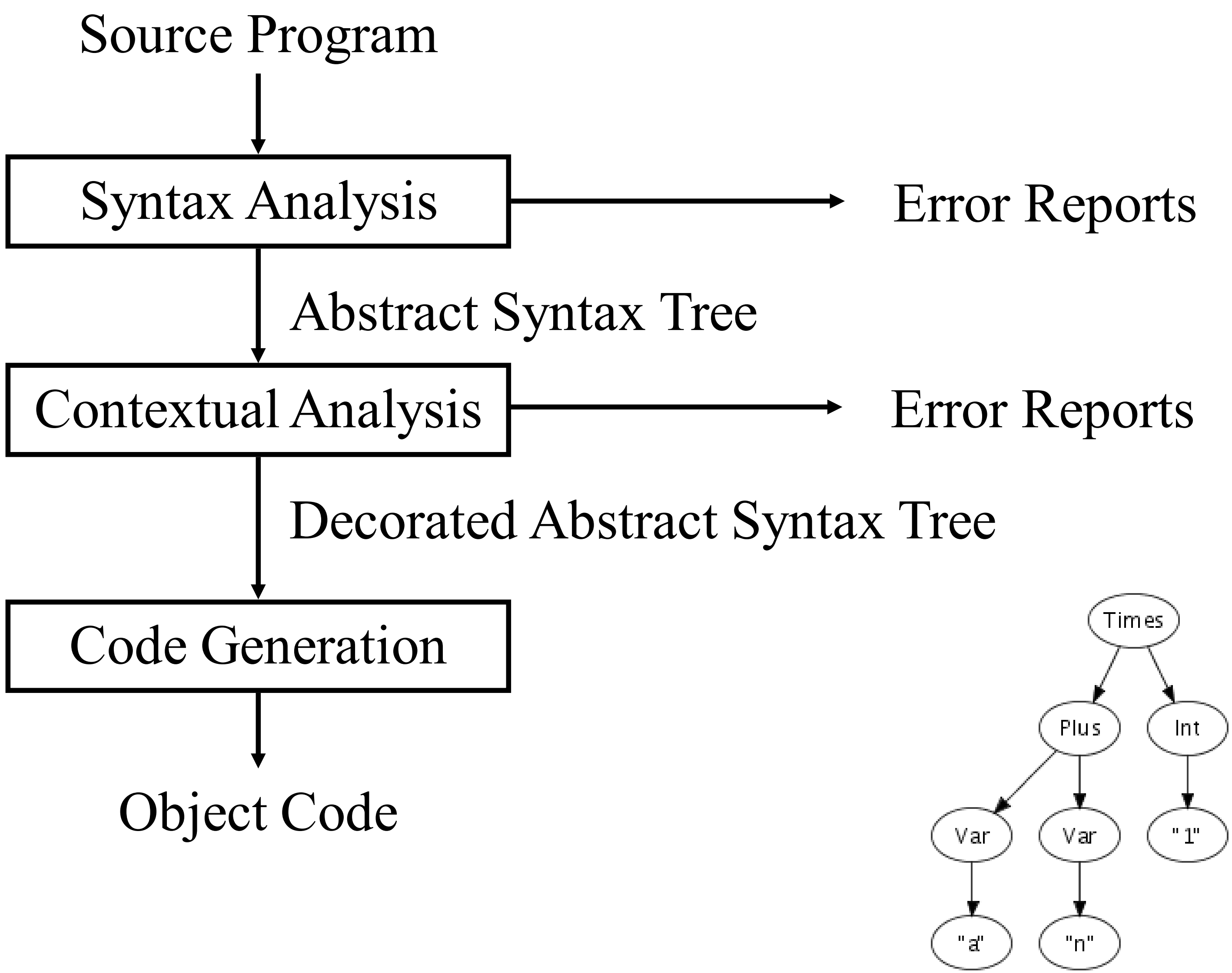
Hybrid compiler / interpreter



Compiler (simple view again)



The Phases of a Compiler



Different Phases of a Compiler

The different phases can be seen as different transformation steps to transform source code into object code.

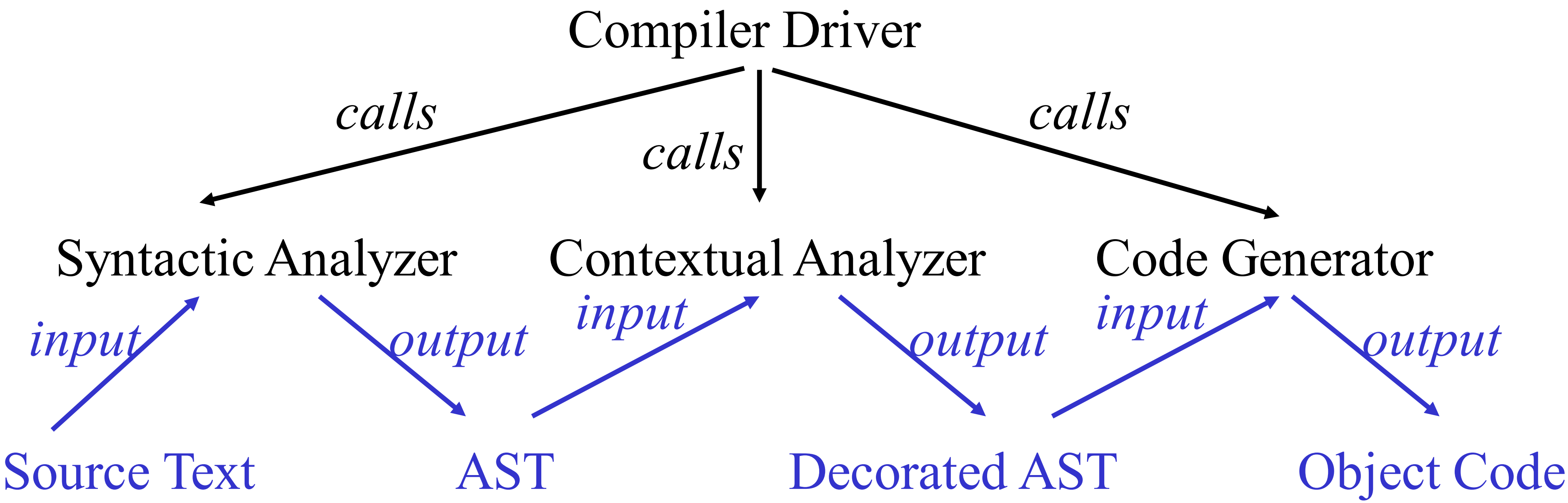
The different phases correspond roughly to the different parts of the language specification:

- Syntax analysis <-> Syntax
- Contextual analysis <-> Contextual constraints
- Code generation <-> Semantics

Multi Pass Compiler

A multi pass compiler makes several passes over the program. The output of a preceding phase is stored in a data structure and used by subsequent phases.

Dependency diagram of a typical Multi Pass Compiler:



Organization of a Compiler

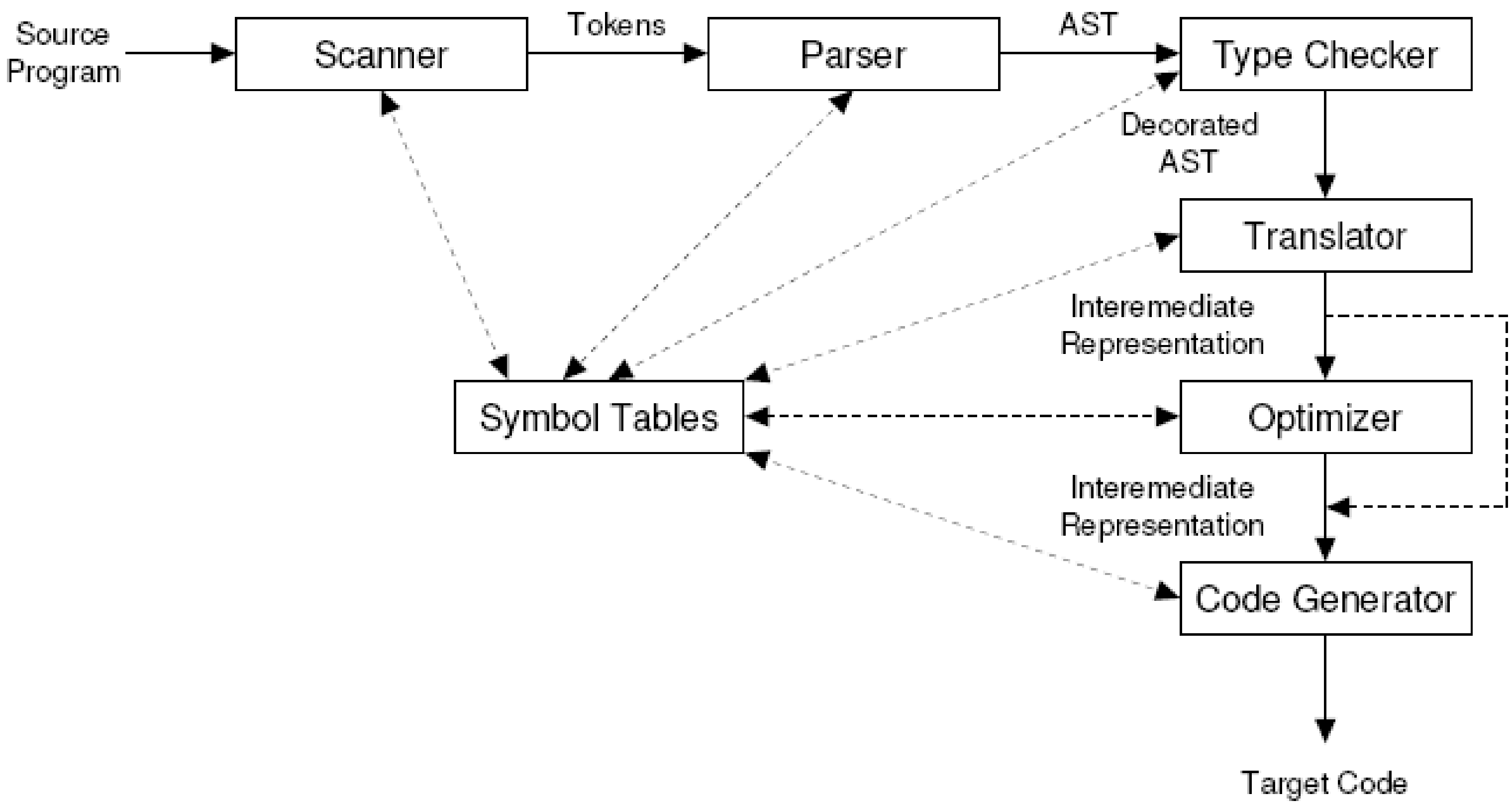
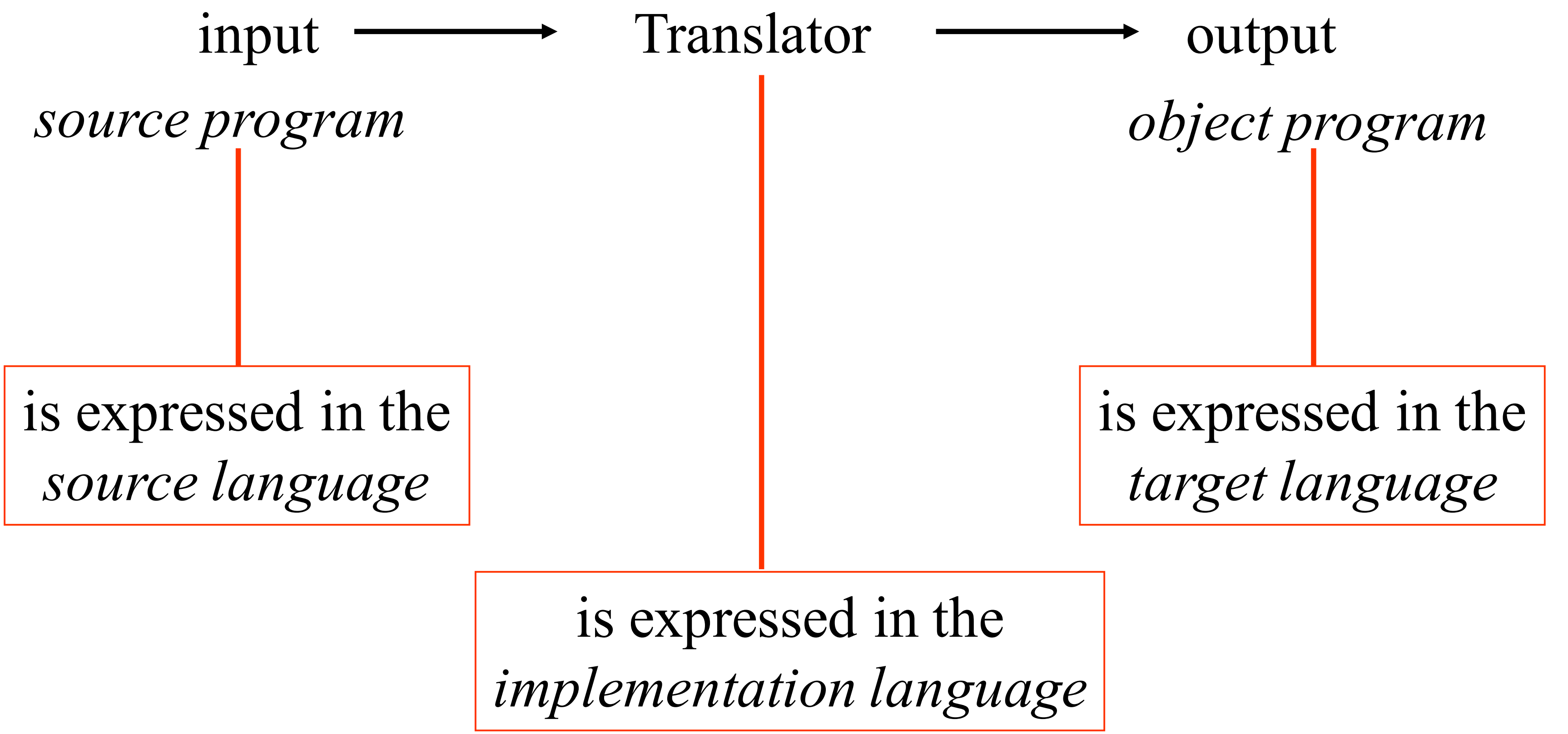


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

Programming Language Implementation

Q: Which programming languages play a role in this picture?

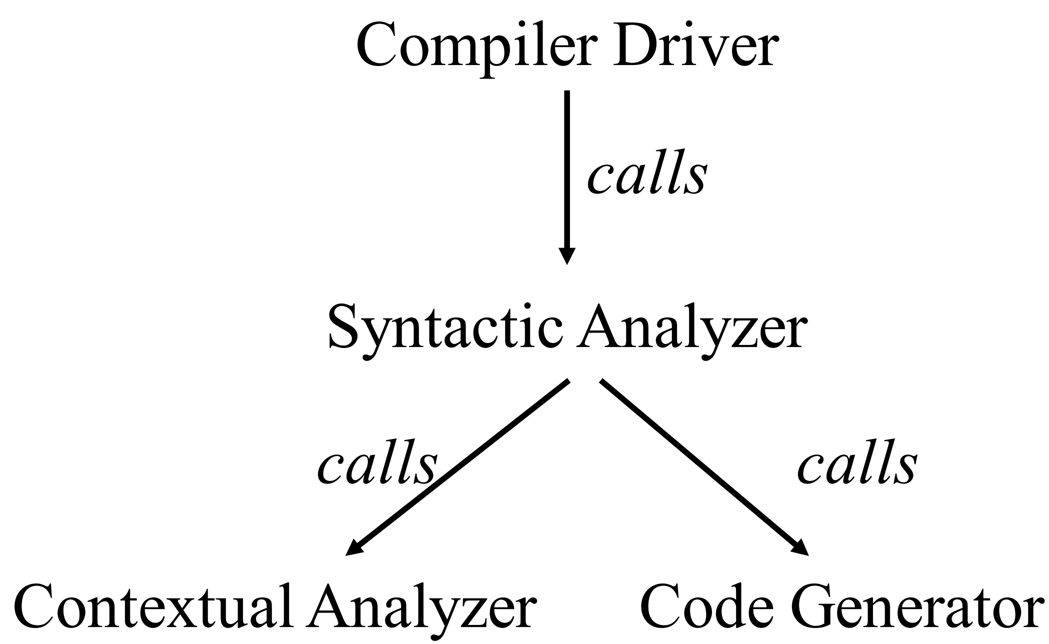


A: All of them!

Single Pass Compiler

A single pass compiler makes a single pass over the source text, parsing, analyzing and generating code all at once.

Dependency diagram of a typical Single Pass Compiler:



Programming Language and Compiler Design

- Many compiler techniques arise from the need to cope with some programming language construct
- The state of the art in compiler design also strongly affects programming language design
- The advantages of a programming language that's easy to compile:
  - Easier to learn, read, understand
  - Have quality compilers on a wide variety of machines
  - Better code will be generated
  - Fewer compiler bugs
  - The compiler will be smaller, cheaper, faster, more reliable, and more widely used
  - Better diagnostic messages and program development tools

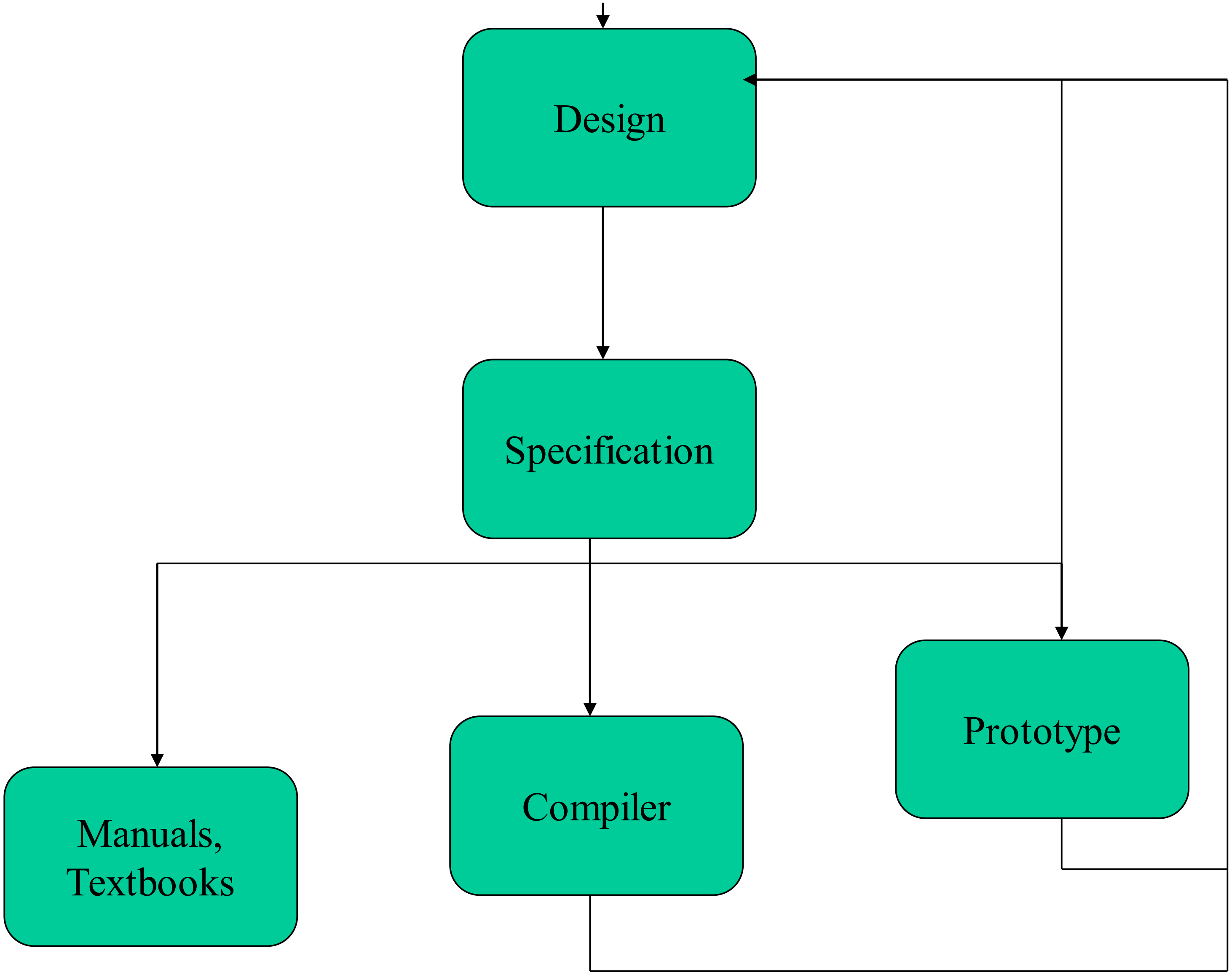
Compiler Writing Tools

- Compiler generators (compiler compilers)
  - Scanner generator
    - JLex (lex, lg)
  - Parser generator
    - JavaCUP (Yacc, pg)
  - Front-end generator
    - SableCC, JavaCC, (COCO/R, ANTLR, ..)
  - Code-generation tools
- Much of the effort in crafting a compiler lies in writing and debugging the semantic phases
  - Usually hand-coded

Programming Language Projects

- A good DAT4/SW4/IT8 project group can
  - Design a language (or language extensions)
  - Define the language syntax using CFG
  - Define the language semantics using SOS
  - Implement a compiler/interpreter
    - in Java (or C/C++, C#, SML, F#, Scala, Kotlin …)
    - Build a recursive decent parser by hand
    - Or using front-end tools such as Lex/Yacc, JavaCC, SableCC, ..
    - Do code generation for abstract machine
      - JVM (PerlVM or .Net CLR) or new VM
    - Or code generation to some high level language
      - C, Java, C#, SQL, XML
    - Or code generation for some hardware platform
      - MIPS, X86, ARM, ATmega, Z80, …
  - (Prove correctness of compiler)
    - Using SOS for Prg. Lang. and VM

Programming Language Life Cycle



Some advice

- A language design and compiler project is easy to structure.
  - Design phase (Lecture 1-5 + 13-14 + 19)
  - Front-end development (Lecture 6-9)
  - Contextual analysis (Lecture 10-12)
  - Code generation or interpretation (Lecture 15-18 + 20)
- You will learn the techniques and tools you need in time for you to apply them in your project

Summary

- Programming Language Design
  - New features
  - Paradigm, Philosophy
- Programming Language Specification
  - Syntax
  - Contextual constraints
  - Meaning (semantics and code generation)
- Programming Language Implementation
  - Compiler
  - Interpreter
  - Hybrid system

Important

- At the end of the course you should …
- Know
  - Which techniques exist
  - Which tools exist
- Be able to choose “the right ones”
  - Objective criteria
  - Subjective criteria
- Be able to argue and justify your choices!

Finally

Keep in mind, the compiler is the program from which all other programs arise. If your compiler is under par, all programs created by the compiler will also be under par. No matter the purpose or use -- your own enlightenment about compilers or commercial applications -- you want to be patient and do a good job with this program; in other words, don't try to throw this together on a weekend.

Asking a computer programmer to tell you how to write a compiler is like saying to Picasso, "Teach me to paint like you."

Sigh Nevertheless, Picasso shall try.

Languages and Compilers
(SProg og Oversættere)

Lecture 2
Programming Language Evolution

Bent Thomsen
Department of Computer Science
Aalborg University

Learning goals

- Introduction to programming language design
- Overview of the evolution of programming languages

Why Are There So Many Programming Languages

- Why does some people speak French?
- Programming languages have evolved over time as better ways have been developed to design them.
  - First programming languages were developed in the 1950s
  - Since then thousands of languages have been developed
- Different programming languages are designed for different types of programs.

Why do people design new programming Languages?

- Most new languages are invented out of frustration!
  - "The decision to create a new programming language or to design an extension of an existing language is often a reaction to some language that the designer knows (and likes or dislikes)"
    - P. Sestoft 2012
- A few languages are created because somebody requested a new language
  - Fortran, C#, Swift, DART
  - All of you, because the study regulations says so ☺

Java	Python

<pre>public class Employee
{
    private String myEmployeeName;
    private int   myTaxDeductions = 1;
    private String myMaritalStatus = "single";
 	
    //--------- constructor #1 -------------
    public Employee(String EmployeName)
    {
        this(employeeName, 1);
    }
 	
    //--------- constructor #2 -------------
    public Employee(String EmployeName, int taxDeductions)
    {
       this(employeeName, taxDeductions, "single");
    }
 	
    //--------- constructor #3 -------------
    public Employee(String EmployeName,
           int taxDeductions,
           String maritalStatus)
    {
       this.employeeName   = employeeName;
       this.taxDeductions   = taxDeductions;
       this.maritalStatus   = maritalStatus;
    }
...</pre>
	<pre>class Employee():
	
	    def __init__(self,
	        employeeName, taxDeductions=1, maritalStatus="single"):
	
	        self.employeeName = employeeName
	        self.taxDeductions = taxDeductions
	        self.maritalStatus = maritalStatus
	...</pre>
	In Python, a class has only one constructor. The constructor method is simply another method of the class, but one that has a special name: `__init__`

Programming Language design

- Designing a new programming language or extending an existing programming language usually follows an iterative approach:

1. Create ideas for the programming language or extensions
2. Describe/define the programming language or extensions
3. Implement the programming language or extensions
4. Evaluate the programming language or extensions
5. If not satisfied, goto 1

Programming Language design

1. Create ideas for the programming language or extensions
   - This subject is almost completely absent from literature!
2. Describe/define the programming language or extensions
   - We will spend quite a bit of time in this course and the SS
3. Implement the programming language or extensions
   - We will spend a lot of time on this subject.
4. Evaluate the programming language or extensions
   - is not usually covered in classic litterature on Programming Languages and Compilers!
   - But you saw Sebesta's Language evaluation criteria in the last lecture
   - We shall see a some more later.

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

How to create ideas for a new programming language or extensions ?

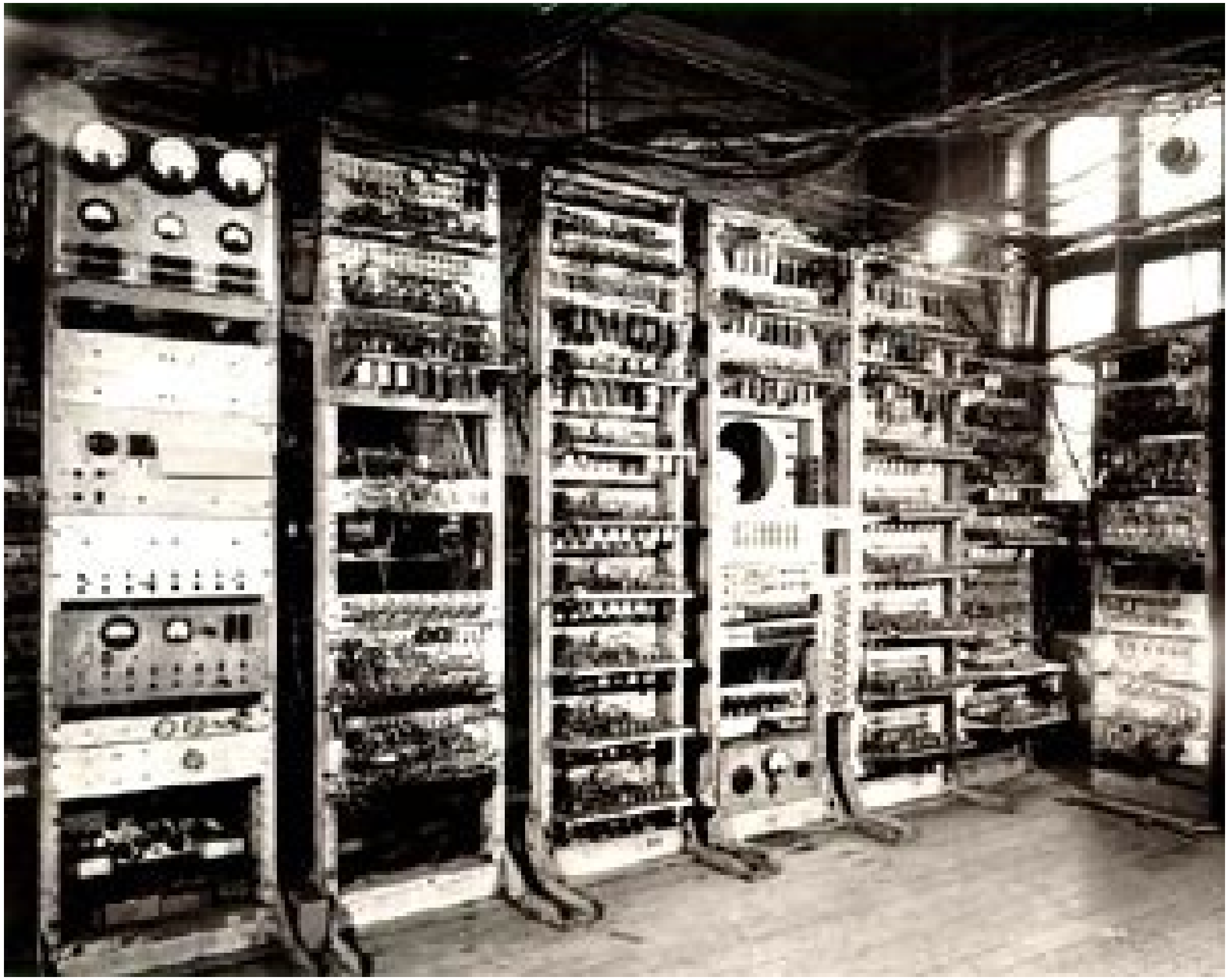
- Do a problem analysis!
  - Who needs the new language?
  - What is the purpose of the new language
  - What type of programs would we like to write?
    - Create some example programs
    - Even before you have defined the language you can create examples of programs as you would like them to look
- Take inspiration from other languages
  - Which langauges do you know?
  - What do you like about these languages?
  - What do you dislike?
  - Look at languages you don't know!
  - Look at the history of programming languages

Programming Language History
1940s

The first electronic computers were monstrous contraptions

- Programmed in binary *machine code* by hand
- Code is not reusable or *relocatable*
  - *Each machine had its own machine language*
- Computation and machine maintenance were difficult:
  - cathode tubes regularly burned out
  - The term ‘‘*bug*’’ originated from a bug that reportedly roamed around in a machine causing short circuits

… in the beginning of time



Programming Language History
Late 1940s early 1950s

- *Assembly languages*
  - invented to allow machine operations to be expressed in mnemonic abbreviations
  - Enables larger, reusable, and re-locatable programs
  - Actual machine code is produced by an *assembler*
  - Early assemblers had a one-to-one correspondence between assembly and machine instructions
  - Later: expansion of *macros* into multiple machine instructions to achieve a form of higher-level programming

Assembly
LOAD x
ADD R1 R2

```
; Hello World for Intel Assembler (MSDOS)

mov ax,cs
mov ds,ax
mov ah,9
mov dx, offset Hello
int 21h
xor ax,ax
int 21h
```

Programming Language History
Mid 1950s

- Fortran , the first higher-level language
  - Now programs could be developed that were machine independent!
  - Main computing activity in the 50s: solve numerical problems in science and engineering
  - Other high-level languages soon followed:
    - Algol 58 is an improvement compared to Fortran
    - Cobol for business computing
    - Lisp for symbolic computing and artificial intelligence
    - BASIC for "beginners"

```
C     Hello World in Fortran

      PROGRAM HELLO
      WRITE (*,100)
      STOP
  100 FORMAT (' Hello World! ' /)
      END
```

```
* Hello World in COBOL

*****************************
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
MAIN SECTION.
DISPLAY "Hello World!"
STOP RUN.
****************************
```

Programming Language History
1960s

- Structured Programming
  - Dijkstra, Dahl, and Hoare.
- Pascal, Niklaus Wirth (ETH, Zurich)
  - Modelled after Algol
  - No GOTO
  - Very strongly typed
  - Procedures nested inside each other
  - Designed for teaching programming
- Simula, Dahl and Nygaard (Norway)
  - The first language with objects, classes, and subclasses

```pascal
{Hello world in Pascal}

program HelloWorld(output);
begin
  WriteLn('Hello World!');
end.
```

Programming Language History
1970s

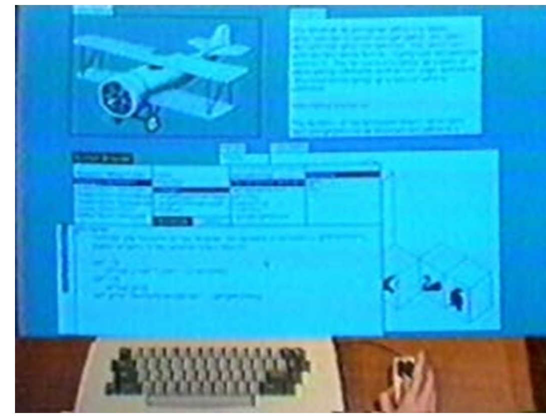
- C, Dennis Ritchie/Ken Thompson (Bell Labs)
  - Successor to B, which was stripped-down BCPL.
  - High-level constructs and low-level power
  - Flat name space for functions/procedures
- Ada, Jean Ichbiah (France)
  - Instigated by the Department of Defense
  - Designed for systems programming, especially embedded systems.

```c
/* Hello World in C, Ansi-style */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  puts("Hello World!");
  return EXIT_SUCCESS;
}
```

```ada
-- Hello World in Ada

with Text_IO;
procedure Hello_World is

begin
  Text_IO.Put_Line("Hello
World!");
end Hello_World;
```

Programming Language History 1970s

- Smalltalk, Alan Kay, Adele Goldberg (Xerox PARC)
  - Graphics-rich
    - GUI
    - Fonts
  - Object-oriented
    - Everything is an object
    - Objects communicate through messages



- Scheme, Gerald Sussman & Guy Steele (MIT)
  - LISP with static scoping
- Prolog, Philippe Roussel (France)
  - Based on rules, facts, and queries.

"Hello World in Smalltalk"

Transcript show: 'Hello World!'.

; Hello World in Scheme

(display "Hello, world!")
(newline)

% Hello World in Prolog

hello :- display('Hello World!') , nl .

Programming Language History 1980s

- Object-oriented programming
  - Important innovation for software development
  - The concept of a class is based on the notion of data type abstraction from Simula 67 , a language for discrete event simulation that has classes but no inheritance
- 1979-1983: C++ Bjarne Stroustrop (Bell Labs)
  - Originally thought of as “C with classes”.
  - First widely-accepted object-oriented language.
  - First implemented as a pre-processor for the C compiler.

```
// Hello World in C++ (pre-ISO)

#include <iostream.h>

main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

Programming Language History
1980s

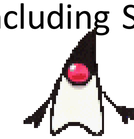
- Functional Programming
  - Extensive list of new concepts
    - Lazy vs. eager evaluation
    - Pure vs. imperative features
    - Parametric polymorphism
    - Type inference
    - (Garbage collection)
  - Hope
  - Clean
  - Haskell
  - SML
  - Caml

Programming Language History 1990s

- HTML, Tim Berners-Lee (CERN)
  - "Hypertext Markup Language"
    - Language of the World Wide Web.
  - A markup language, not a programming language.
- Scripting languages
  - PERL.
    - CGI or Apache module
  - Languages within Web pages
    - JavaScript, VBScript
    - PHP, ASP, JSP
- Java, James Gosling (Sun)

The evolution of Java

- 1993 Oak project at Sun
  - small, robust, architecture independent, Object-Oriented, language to control interactive TV.
  - didn’t go anywhere
- 1995 Oak becomes Java
  - Focus on the web
- 1996 Java 1.0 available
- 1997 (March) Java 1.1 - some language changes, much larger library, new event handling model
- 1997 (September) Java 1.2 beta – huge increase in libraries including Swing, new collection classes, J2EE
- 1998 (October) Java 1.2 final (Java2!)
- 2000 (April) Java 1.3 final
- 2001 Java 1.4 final (assert)
- 2004 Java 1.5 (parameterized types, enum, …)
- 2005 J2EE 1.5
- 2006 Java 6
- 2011 Java 7
- 2014 Java 8 (lambda expressions)
- 2017 Java 9 (expected 23.3.17, but released 21.9.17
- – REPL, process control, collections, streams, …)
- 2018 Java 10 (March – Minor updates, GC interface, parallel GC)
- 2018 Java 11 (September - Local-variable syntax for lambda parameters, ZGC: a scalable low-latency GC)
- 2019 Java 12 (March)
- Java SE 13 (September 17, 2019)
- Java SE 14 (March 17, 2020) – preview of patternmatching
- Java SE 15 (September 15, 2020)



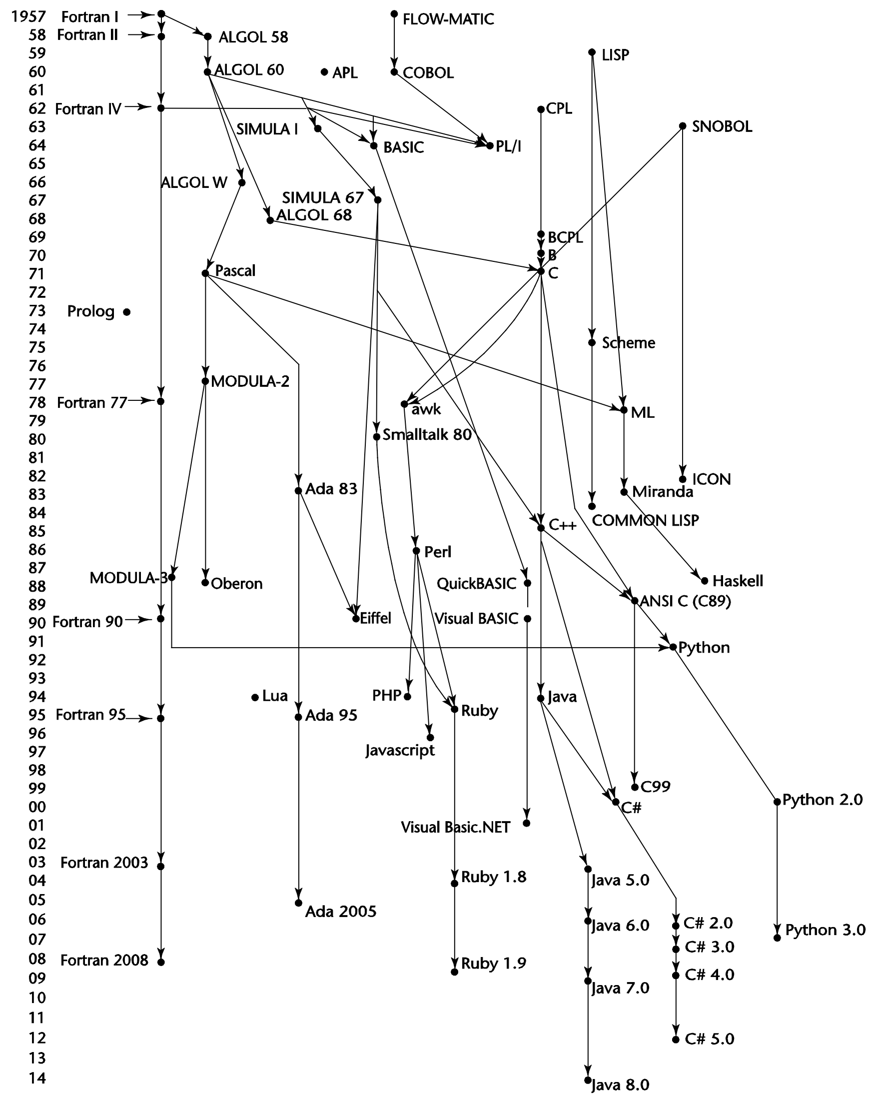
Programming Language History 2000s

- XML
- Microsoft .NET
  - Multiple languages
    - C++
    - C#
    - Visual Basic
    - COBOL
    - Fortran
    - Eiffel
  - Common virtual machine (.Net CLR)
  - Web services

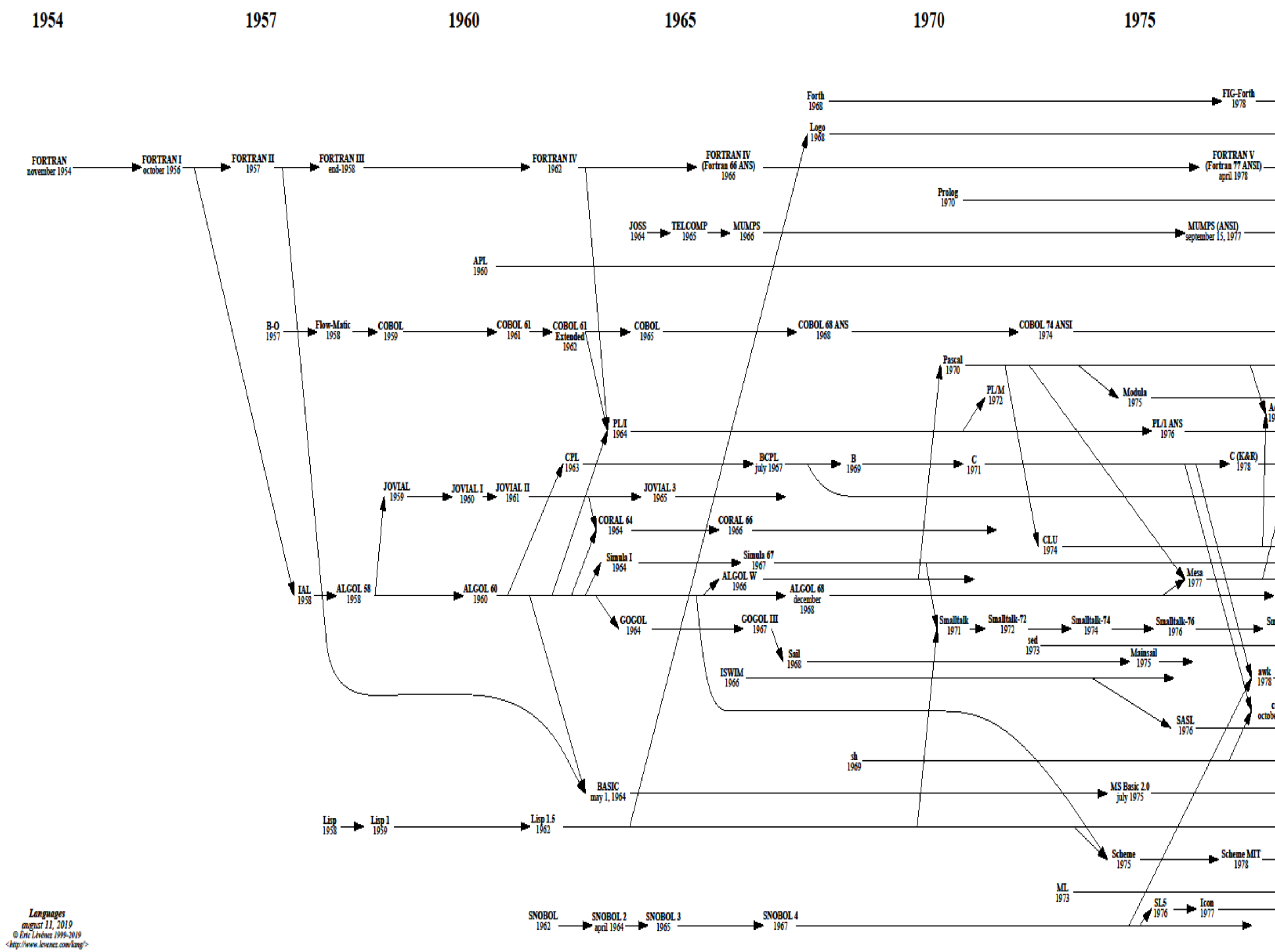
C# History

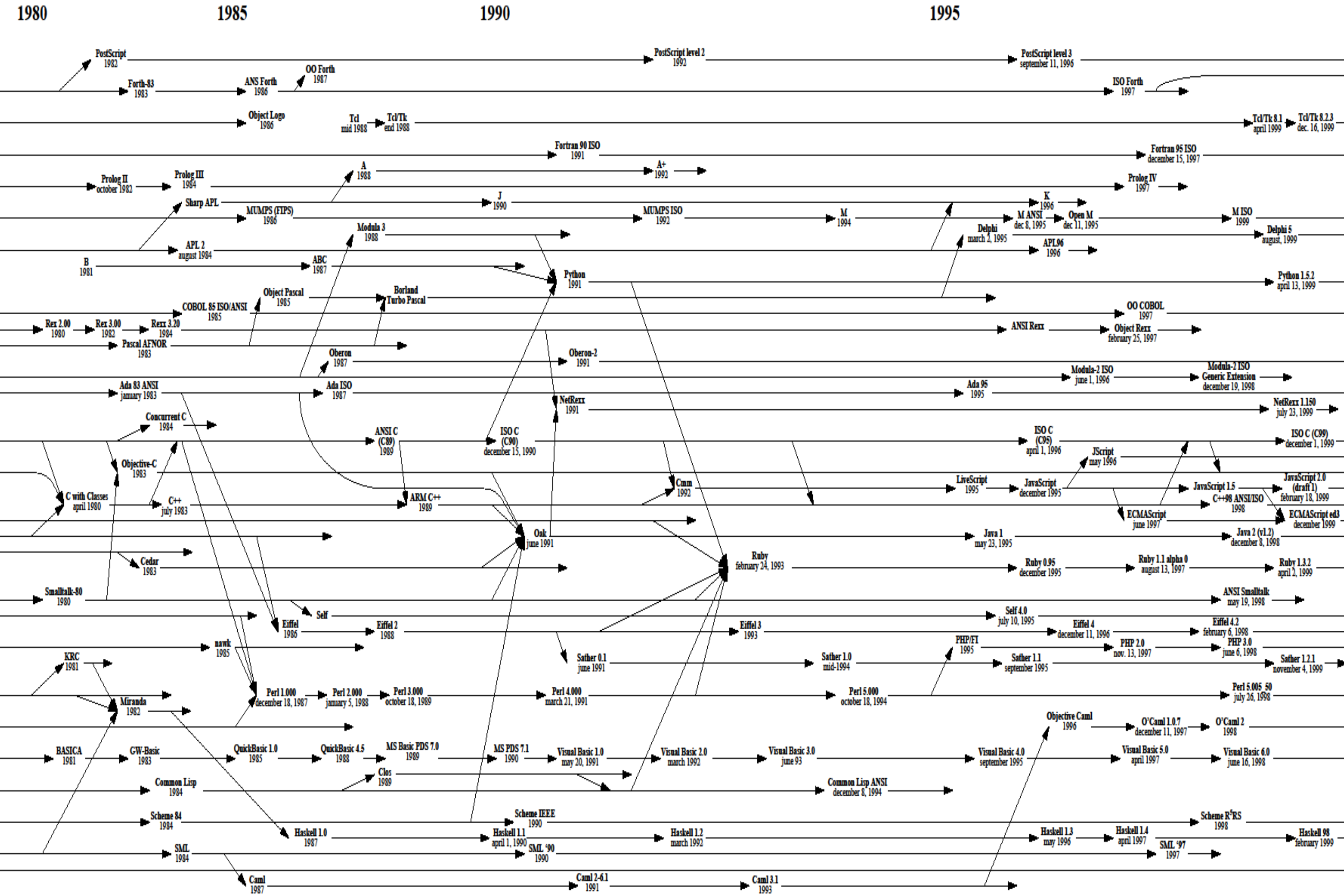
- 12/1998 – COOL project started
- 07/1999 – First internal ports to COOL
- 02/2000 – Named changed to C#
- 07/2000 – First public preview release
- 02/2002 – C# 1.0, VS.NET 2002
- 05/2003 – C# 1.1, VS.NET 2003
- 06/2004 – Beta 1 of C# 2.0 and VS 2005
- 04/2005 – Beta 2 of C# 2.0 and VS 2005
- 11/2005 – C# 2.0 VS 2005, C# 2.0 release
  - Generics, anonymous delegates, nullable types, iterators, partial classes
- 11/2006 – C# 3.0, VS 2008
  - (local type inference, lambdas, expression trees, LINQ)
- 04/2010 – C# 4.0, VS 2010
  - Type dynamics, named+optional parameters, co-/contra variant generics
- 08/2012 – C# 5.0, VS 2012
  - Async methods
- 06/2015 – C# 6.0, VS 2015
  - Await in catch/finally blocks, succinct null checking
- 2017 – C# 7.0,7.1,7.2, VS 2017
  - Pattern matching, Local functions, tuples
- 2018 – C# 7.3
  - Reassigning ref local variables, Using initializers on stackalloc arrays
- 2019 – C# 8
  - readonly struct members, default interface members, switch expressions, Property, Tuple, and positional patterns, using declarations
  - static local functions, Disposable ref struct, Nullable reference types, Indices and Ranges, Null-coalescing assignment,Async Streams
- 2020 – C# 9

Genealogy of Common Languages

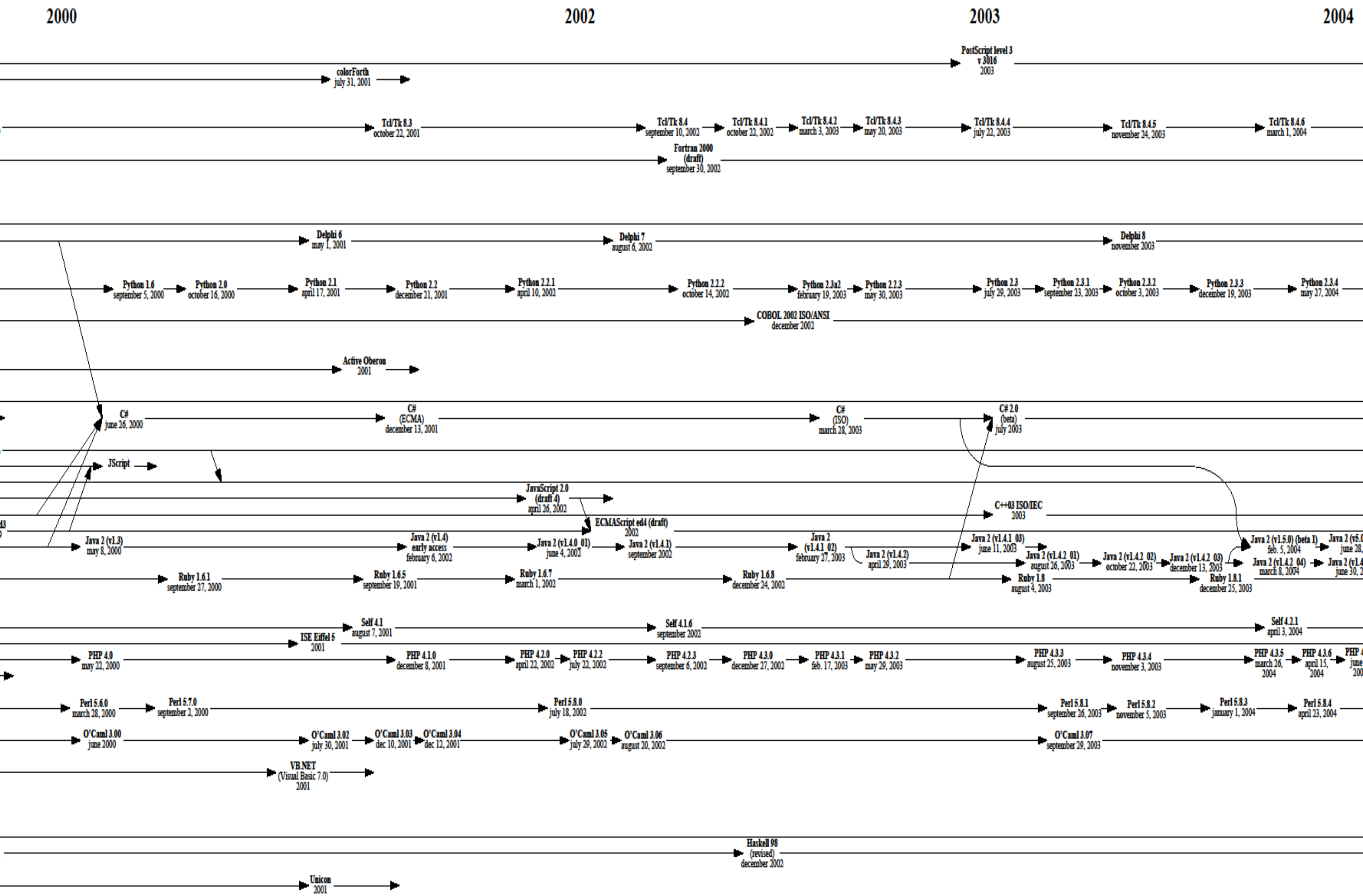


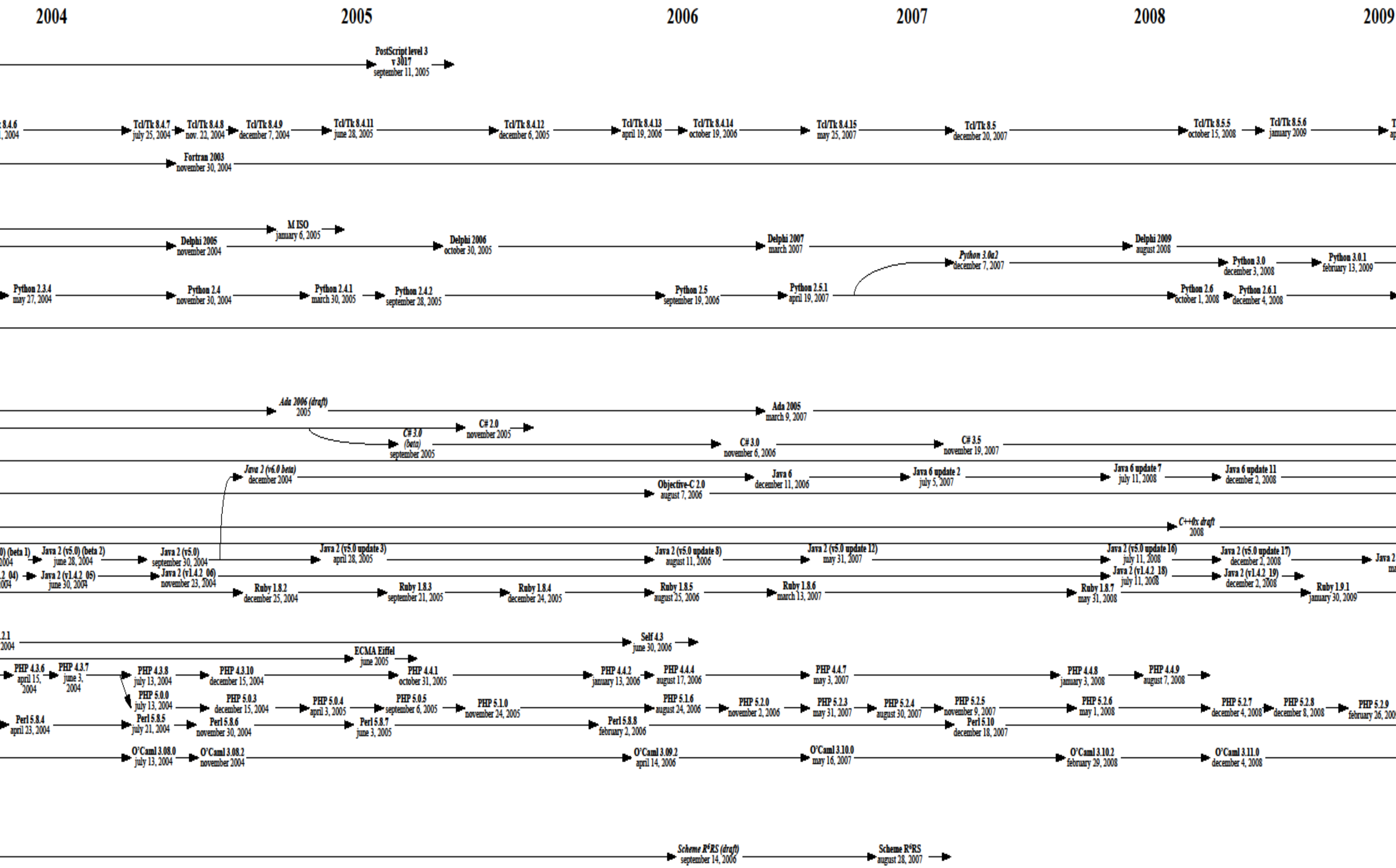
lang.pdf


1954
1957
1960
1965
1970
1975
Forth 1968
FIG-Forth 1978
Logo 1968
FORTRAN november 1954
FORTRAN I october 1956
FORTRAN II 1957
FORTRAN III end-1958
FORTRAN IV 1962
FORTRAN IV (Fortran 66 ANS) 1966
FORTRAN V (Fortran 77 ANSI) april 1978
Prolog 1970
JOSS 1964
TELCOMP 1965
MUMPS 1966
MUMPS (ANSI) september 15, 1977
APL 1960
B-O 1957
Flow-Matic 1958
COBOL 1959
COBOL 61 1961
COBOL 61 Extended 1962
COBOL 1965
COBOL 68 ANS 1968
COBOL 74 ANSI 1974
Pascal 1970
PL/M 1972
Modula 1975
PL/I 1964
PL/1 ANS 1976
CPL 1963
BCPL july 1967
B 1969
C 1971
C (K&R) 1978
JOVIAL 1959
JOVIAL I 1960
JOVIAL II 1961
JOVIAL 3 1965
CORAL 64 1964
CORAL 66 1966
CLU 1974
Simula I 1964
Simula 67 1967
ALGOL W 1966
Mesa 1977
IAL 1958
ALGOL 58 1958
ALGOL 60 1960
ALGOL 68 december 1968
GOGOL 1964
GOGOL III 1967
Smalltalk 1971
Smalltalk-72 1972
Smalltalk-74 1974
Smalltalk-76 1976
sed 1973
Sail 1968
Mainsail 1975
ISWIM 1966
awk 1978
SASL 1976
sh 1969
BASIC may 1, 1964
MS Basic 2.0 july 1975
Lisp 1958
Lisp 1 1959
Lisp 1.5 1962
Scheme 1975
Scheme MIT 1978
ML 1973
SL5 1976
Icon 1977
SNOBOL 1962
SNOBOL 2 april 1964
SNOBOL 3 1965
SNOBOL 4 1967
Languages
august 11, 2019
© Éric Lévénez 1999-2019
<http://www.levenez.com/lang/>

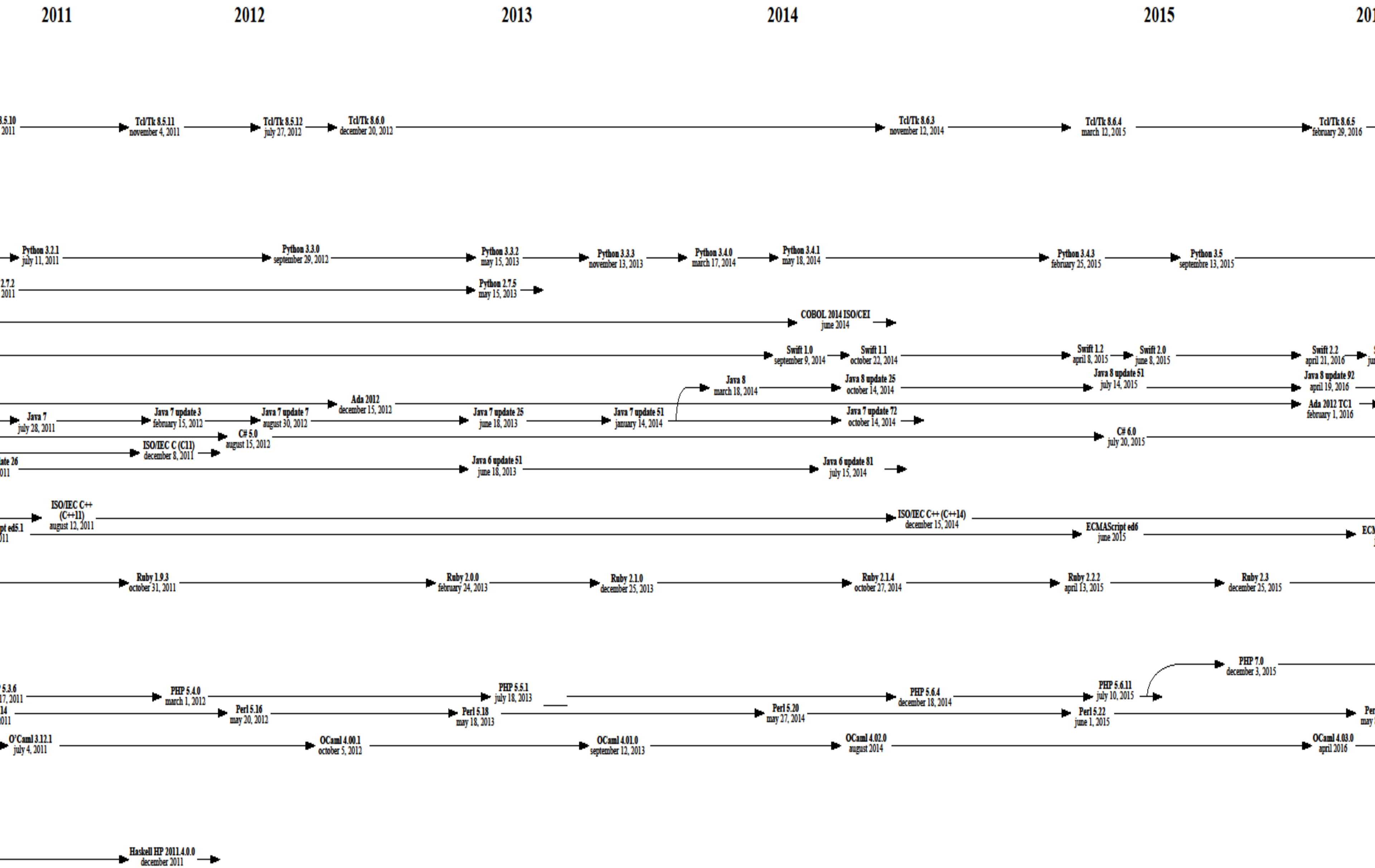


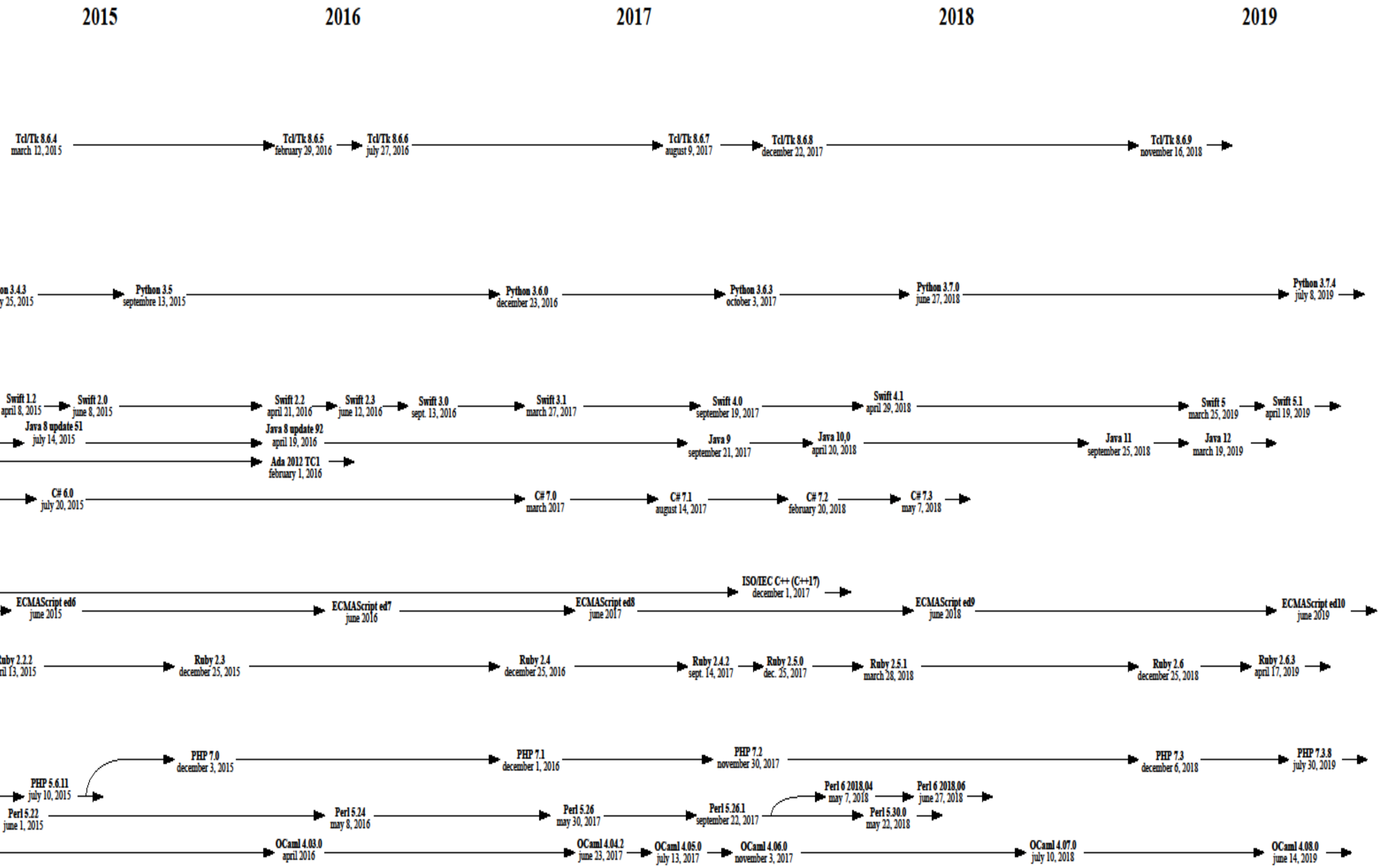
1980
1985
1990
1995
PostScript 1982
PostScript level 2 1992
PostScript level 3 september 11, 1996
Forth-83 1983
ANS Forth 1986
OO Forth 1987
ISO Forth 1997
Object Logo 1986
Tcl mid 1988
Tcl/Tk end 1988
Tcl/Tk 8.1 april 1999
Tcl/Tk 8.2.3 dec. 16, 1999
Fortran 90 ISO 1991
Fortran 95 ISO december 15, 1997
A 1988
A+ 1992
Prolog II october 1982
Prolog III 1984
Prolog IV 1997
Sharp APL
J 1990
K 1996
MUMPS (FIPS) 1986
MUMPS ISO 1992
M 1994
M ANSI dec 8, 1995
Open M dec 11, 1995
M ISO 1999
Modula 3 1988
Delphi march 2, 1995
Delphi 5 august, 1999
APL 2 august 1984
APL96 1996
B 1981
ABC 1987
Python 1991
Python 1.5.2 april 13, 1999
Object Pascal 1985
Borland Turbo Pascal
COBOL 85 ISO/ANSI 1985
OO COBOL 1997
Rex 2.00 1980
Rex 3.00 1982
Rexx 3.20 1984
ANSI Rexx
Object Rexx february 25, 1997
Pascal AFNOR 1983
Oberon 1987
Oberon-2 1991
Modula-2 ISO june 1, 1996
Modula-2 ISO Generic Extension december 19, 1998
Ada 83 ANSI january 1983
Ada ISO 1987
Ada 95 1995
NetRexx 1991
NetRexx 1.150 july 23, 1999
Concurrent C 1984
ANSI C (C89) 1989
ISO C (C90) december 15, 1990
ISO C (C95) april 1, 1996
ISO C (C99) december 1, 1999
JScript may 1996
Objective-C 1983
Cmm 1992
LiveScript 1995
JavaScript december 1995
JavaScript 1.5
JavaScript 2.0 (draft 1) february 18, 1999
C with Classes april 1980
C++ july 1983
ARM C++ 1989
C++98 ANSI/ISO 1998
ECMAScript june 1997
ECMAScript ed3 december 1999
Oak june 1991
Java 1 may 23, 1995
Java 2 (v1.2) december 8, 1998
Cedar 1983
Ruby february 24, 1993
Ruby 0.95 december 1995
Ruby 1.1 alpha 0 august 13, 1997
Ruby 1.3.2 april 2, 1999
Smalltalk-80 1980
ANSI Smalltalk may 19, 1998
Self
Self 4.0 july 10, 1995
Eiffel 1986
Eiffel 2 1988
Eiffel 3 1993
Eiffel 4 december 11, 1996
Eiffel 4.2 february 6, 1998
nawk 1985
PHP/FI 1995
PHP 2.0 nov. 13, 1997
PHP 3.0 june 6, 1998
KRC 1981
Sather 0.1 june 1991
Sather 1.0 mid-1994
Sather 1.1 september 1995
Sather 1.2.1 november 4, 1999
Miranda 1982
Perl 1.000 december 18, 1987
Perl 2.000 january 5, 1988
Perl 3.000 october 18, 1989
Perl 4.000 march 21, 1991
Perl 5.000 october 18, 1994
Perl 5.005_50 july 26, 1998
Objective Caml 1996
O'Caml 1.0.7 december 11, 1997
O'Caml 2 1998
BASICA 1981
GW-Basic 1983
QuickBasic 1.0 1985
QuickBasic 4.5 1988
MS Basic PDS 7.0 1989
MS PDS 7.1 1990
Visual Basic 1.0 may 20, 1991
Visual Basic 2.0 march 1992
Visual Basic 3.0 june 93
Visual Basic 4.0 september 1995
Visual Basic 5.0 april 1997
Visual Basic 6.0 june 16, 1998
Common Lisp 1984
Clos 1989
Common Lisp ANSI december 8, 1994
Scheme 84 1984
Scheme IEEE 1990
Scheme R5RS 1998
Haskell 1.0 1987
Haskell 1.1 april 1, 1990
Haskell 1.2 march 1992
Haskell 1.3 may 1996
Haskell 1.4 april 1997
Haskell 98 february 1999
SML 1984
SML '90 1990
SML '97 1997
Caml 1987
Caml 2-6.1 1991
Caml 3.1 1993


2000
2002
2003
2004
PostScript level 3 v 3016 2003
colorForth july 31, 2001
Tcl/Tk 8.3 october 22, 2001
Tcl/Tk 8.4 september 10, 2002
Tcl/Tk 8.4.1 october 22, 2002
Tcl/Tk 8.4.2 march 3, 2003
Tcl/Tk 8.4.3 may 20, 2003
Tcl/Tk 8.4.4 july 22, 2003
Tcl/Tk 8.4.5 november 24, 2003
Tcl/Tk 8.4.6 march 1, 2004
Fortran 2000 (draft) september 30, 2002
Delphi 6 may 1, 2001
Delphi 7 august 6, 2002
Delphi 8 november 2003
Python 1.6 september 5, 2000
Python 2.0 october 16, 2000
Python 2.1 april 17, 2001
Python 2.2 december 21, 2001
Python 2.2.1 april 10, 2002
Python 2.2.2 october 14, 2002
Python 2.3a2 february 19, 2003
Python 2.2.3 may 30, 2003
Python 2.3 july 29, 2003
Python 2.3.1 september 23, 2003
Python 2.3.2 october 3, 2003
Python 2.3.3 december 19, 2003
Python 2.3.4 may 27, 2004
COBOL 2002 ISO/ANSI december 2002
Active Oberon 2001
C# june 26, 2000
C# (ECMA) december 13, 2001
C# (ISO) march 28, 2003
C# 2.0 (beta) july 2003
JScript
JavaScript 2.0 (draft 4) april 26, 2002
C++03 ISO/IEC 2003
ECMAScript ed4 (draft) 2002
Java 2 (v1.3) may 8, 2000
Java 2 (v1.4) early access february 6, 2002
Java 2 (v1.4.0_01) june 4, 2002
Java 2 (v1.4.1) september 2002
Java 2 (v1.4.1_02) february 27, 2003
Java 2 (v1.4.2) april 29, 2003
Java 2 (v1.4.1_03) june 11, 2003
Java 2 (v1.4.2_01) august 26, 2003
Java 2 (v1.4.2_02) october 22, 2003
Java 2 (v1.4.2_03) december 13, 2003
Java 2 (v1.5.0) (beta 1) feb. 5, 2004
Java 2 (v1.4.2_04) march 8, 2004
Ruby 1.6.1 september 27, 2000
Ruby 1.6.5 september 19, 2001
Ruby 1.6.7 march 1, 2002
Ruby 1.6.8 december 24, 2002
Ruby 1.8 august 4, 2003
Ruby 1.8.1 december 25, 2003
Self 4.1 august 7, 2001
Self 4.1.6 september 2002
Self 4.2.1 april 3, 2004
ISE Eiffel 5 2001
PHP 4.0 may 22, 2000
PHP 4.1.0 december 8, 2001
PHP 4.2.0 april 22, 2002
PHP 4.2.2 july 22, 2002
PHP 4.2.3 september 6, 2002
PHP 4.3.0 december 27, 2002
PHP 4.3.1 feb. 17, 2003
PHP 4.3.2 may 29, 2003
PHP 4.3.3 august 25, 2003
PHP 4.3.4 november 3, 2003
PHP 4.3.5 march 26, 2004
PHP 4.3.6 april 15, 2004
Perl 5.6.0 march 28, 2000
Perl 5.7.0 september 2, 2000
Perl 5.8.0 july 18, 2002
Perl 5.8.1 september 26, 2003
Perl 5.8.2 november 5, 2003
Perl 5.8.3 january 1, 2004
Perl 5.8.4 april 23, 2004
O'Caml 3.00 june 2000
O'Caml 3.02 july 30, 2001
O'Caml 3.03 dec 10, 2001
O'Caml 3.04 dec 12, 2001
O'Caml 3.05 july 29, 2002
O'Caml 3.06 august 20, 2002
O'Caml 3.07 september 29, 2003
VB.NET (Visual Basic 7.0) 2001
Haskell 98 (revised) december 2002
Unicon 2001



2004
2005
2006
2007
2008
2009
PostScript level 3
v 3017
september 11, 2005
Tcl/Tk 8.4.7
july 25, 2004
Tcl/Tk 8.4.8
nov. 22, 2004
Tcl/Tk 8.4.9
december 7, 2004
Tcl/Tk 8.4.11
june 28, 2005
Tcl/Tk 8.4.12
december 6, 2005
Tcl/Tk 8.4.13
april 19, 2006
Tcl/Tk 8.4.14
october 19, 2006
Tcl/Tk 8.4.15
may 25, 2007
Tcl/Tk 8.5
december 20, 2007
Tcl/Tk 8.5.5
october 15, 2008
Tcl/Tk 8.5.6
january 2009
Fortran 2003
november 30, 2004
M ISO
january 6, 2005
Delphi 2005
november 2004
Delphi 2006
october 30, 2005
Delphi 2007
march 2007
Delphi 2009
august 2008
Python 3.0a2
december 7, 2007
Python 3.0
december 3, 2008
Python 3.0.1
february 13, 2009
Python 2.3.4
may 27, 2004
Python 2.4
november 30, 2004
Python 2.4.1
march 30, 2005
Python 2.4.2
september 28, 2005
Python 2.5
september 19, 2006
Python 2.5.1
april 19, 2007
Python 2.6
october 1, 2008
Python 2.6.1
december 4, 2008
Ada 2006 (draft)
2005
Ada 2005
march 9, 2007
C# 2.0
november 2005
C# 3.0
(beta)
september 2005
C# 3.0
november 6, 2006
C# 3.5
november 19, 2007
Java 2 (v6.0 beta)
december 2004
Java 6
december 11, 2006
Java 6 update 2
july 5, 2007
Java 6 update 7
july 11, 2008
Java 6 update 11
december 2, 2008
Objective-C 2.0
august 7, 2006
C++0x draft
2008
Java 2 (v5.0) (beta 2)
june 28, 2004
Java 2 (v5.0)
september 30, 2004
Java 2 (v5.0 update 3)
april 28, 2005
Java 2 (v5.0 update 8)
august 11, 2006
Java 2 (v5.0 update 12)
may 31, 2007
Java 2 (v5.0 update 16)
july 11, 2008
Java 2 (v5.0 update 17)
december 2, 2008
Java 2 (v1.4.2_05)
june 30, 2004
Java 2 (v1.4.2_06)
november 23, 2004
Java 2 (v1.4.2_18)
july 11, 2008
Java 2 (v1.4.2_19)
december 2, 2008
Ruby 1.8.2
december 25, 2004
Ruby 1.8.3
september 21, 2005
Ruby 1.8.4
december 24, 2005
Ruby 1.8.5
august 25, 2006
Ruby 1.8.6
march 13, 2007
Ruby 1.8.7
may 31, 2008
Ruby 1.9.1
january 30, 2009
Self 4.3
june 30, 2006
ECMA Eiffel
june 2005
PHP 4.3.6
april 15, 2004
PHP 4.3.7
june 3, 2004
PHP 4.3.8
july 13, 2004
PHP 4.3.10
december 15, 2004
PHP 4.4.1
october 31, 2005
PHP 4.4.2
january 13, 2006
PHP 4.4.4
august 17, 2006
PHP 4.4.7
may 3, 2007
PHP 4.4.8
january 3, 2008
PHP 4.4.9
august 7, 2008
PHP 5.0.0
july 13, 2004
PHP 5.0.3
december 15, 2004
PHP 5.0.4
april 3, 2005
PHP 5.0.5
september 6, 2005
PHP 5.1.0
november 24, 2005
PHP 5.1.6
august 24, 2006
PHP 5.2.0
november 2, 2006
PHP 5.2.3
may 31, 2007
PHP 5.2.4
august 30, 2007
PHP 5.2.5
november 9, 2007
PHP 5.2.6
may 1, 2008
PHP 5.2.7
december 4, 2008
PHP 5.2.8
december 8, 2008
PHP 5.2.9
february 26, 2009
Perl 5.8.4
april 23, 2004
Perl 5.8.5
july 21, 2004
Perl 5.8.6
november 30, 2004
Perl 5.8.7
june 3, 2005
Perl 5.8.8
february 2, 2006
Perl 5.10
december 18, 2007
O'Caml 3.08.0
july 13, 2004
O'Caml 3.08.2
november 2004
O'Caml 3.09.2
april 14, 2006
O'Caml 3.10.0
may 16, 2007
O'Caml 3.10.2
february 29, 2008
O'Caml 3.11.0
december 4, 2008
Scheme R6RS (draft)
september 14, 2006
Scheme R6RS
august 28, 2007


2011
2012
2013
2014
2015
Tcl/Tk 8.5.11
november 4, 2011
Tcl/Tk 8.5.12
july 27, 2012
Tcl/Tk 8.6.0
december 20, 2012
Tcl/Tk 8.6.3
november 12, 2014
Tcl/Tk 8.6.4
march 12, 2015
Tcl/Tk 8.6.5
february 29, 2016
Python 3.2.1
july 11, 2011
Python 3.3.0
september 29, 2012
Python 3.3.2
may 15, 2013
Python 3.3.3
november 13, 2013
Python 3.4.0
march 17, 2014
Python 3.4.1
may 18, 2014
Python 3.4.3
february 25, 2015
Python 3.5
septembre 13, 2015
Python 2.7.5
may 15, 2013
COBOL 2014 ISO/CEI
june 2014
Swift 1.0
september 9, 2014
Swift 1.1
october 22, 2014
Swift 1.2
april 8, 2015
Swift 2.0
june 8, 2015
Swift 2.2
april 21, 2016
Java 8
march 18, 2014
Java 8 update 25
october 14, 2014
Java 8 update 51
july 14, 2015
Java 8 update 92
april 19, 2016
Ada 2012
december 15, 2012
Ada 2012 TC1
february 1, 2016
Java 7
july 28, 2011
Java 7 update 3
february 15, 2012
Java 7 update 7
august 30, 2012
Java 7 update 25
june 18, 2013
Java 7 update 51
january 14, 2014
Java 7 update 72
october 14, 2014
C# 5.0
august 15, 2012
C# 6.0
july 20, 2015
ISO/IEC C (C11)
december 8, 2011
Java 6 update 51
june 18, 2013
Java 6 update 81
july 15, 2014
ISO/IEC C++
(C++11)
august 12, 2011
ISO/IEC C++ (C++14)
december 15, 2014
ECMAScript ed6
june 2015
Ruby 1.9.3
october 31, 2011
Ruby 2.0.0
february 24, 2013
Ruby 2.1.0
december 25, 2013
Ruby 2.1.4
october 27, 2014
Ruby 2.2.2
april 13, 2015
Ruby 2.3
december 25, 2015
PHP 7.0
december 3, 2015
PHP 5.4.0
march 1, 2012
PHP 5.5.1
july 18, 2013
PHP 5.6.4
december 18, 2014
PHP 5.6.11
july 10, 2015
Perl 5.16
may 20, 2012
Perl 5.18
may 18, 2013
Perl 5.20
may 27, 2014
Perl 5.22
june 1, 2015
O'Caml 3.12.1
july 4, 2011
OCaml 4.00.1
october 5, 2012
OCaml 4.01.0
september 12, 2013
OCaml 4.02.0
august 2014
OCaml 4.03.0
april 2016
Haskell HP 2011.4.0.0
december 2011



2015
2016
2017
2018
2019
Tcl/Tk 8.6.4 march 12, 2015
Tcl/Tk 8.6.5 february 29, 2016
Tcl/Tk 8.6.6 july 27, 2016
Tcl/Tk 8.6.7 august 9, 2017
Tcl/Tk 8.6.8 december 22, 2017
Tcl/Tk 8.6.9 november 16, 2018
Python 3.5 septembre 13, 2015
Python 3.6.0 december 23, 2016
Python 3.6.3 october 3, 2017
Python 3.7.0 june 27, 2018
Python 3.7.4 july 8, 2019
Swift 1.2 april 8, 2015
Swift 2.0 june 8, 2015
Swift 2.2 april 21, 2016
Swift 2.3 june 12, 2016
Swift 3.0 sept. 13, 2016
Swift 3.1 march 27, 2017
Swift 4.0 september 19, 2017
Swift 4.1 april 29, 2018
Swift 5 march 25, 2019
Swift 5.1 april 19, 2019
Java 8 update 51 july 14, 2015
Java 8 update 92 april 19, 2016
Java 9 september 21, 2017
Java 10,0 april 20, 2018
Java 11 september 25, 2018
Java 12 march 19, 2019
Ada 2012 TC1 february 1, 2016
C# 6.0 july 20, 2015
C# 7.0 march 2017
C# 7.1 august 14, 2017
C# 7.2 february 20, 2018
C# 7.3 may 7, 2018
ISO/IEC C++ (C++17) december 1, 2017
ECMAScript ed6 june 2015
ECMAScript ed7 june 2016
ECMAScript ed8 june 2017
ECMAScript ed9 june 2018
ECMAScript ed10 june 2019
Ruby 2.3 december 25, 2015
Ruby 2.4 december 25, 2016
Ruby 2.4.2 sept. 14, 2017
Ruby 2.5.0 dec. 25, 2017
Ruby 2.5.1 march 28, 2018
Ruby 2.6 december 25, 2018
Ruby 2.6.3 april 17, 2019
PHP 7.0 december 3, 2015
PHP 7.1 december 1, 2016
PHP 7.2 november 30, 2017
PHP 7.3 december 6, 2018
PHP 7.3.8 july 30, 2019
PHP 5.6.11 july 10, 2015
Perl 6 2018.04 may 7, 2018
Perl 6 2018.06 june 27, 2018
Perl 5.22 june 1, 2015
Perl 5.24 may 8, 2016
Perl 5.26 may 30, 2017
Perl 5.26.1 september 22, 2017
Perl 5.30.0 may 22, 2018
OCaml 4.03.0 april 2016
OCaml 4.04.2 june 23, 2017
OCaml 4.05.0 july 13, 2017
OCaml 4.06.0 november 3, 2017
OCaml 4.07.0 july 10, 2018
OCaml 4.08.0 june 14, 2019

Programming Language History
2010s

- Multi paradigm integration, especially OO+FP(+concurrency)
  - C#, C++ and Java
  - Python
  - Ruby
  - Groovy
  - Clojure
  - Fortress
  - Scala
  - O’Caml, F#
  - Haskell
  - Erlang
  - Swift, DART, RUST, Kotlin

```
-- Hello World in Haskell

main = putStrLn "Hello World“
```

```
%% Hello World in Erlang

-module(hello).

-export([hello/0]).

hello() ->
   io:format("Hello World!~n", []).
```

```
// Hello world in Swift

println("Hello, world!")
```

```
// Hello world in Dart

main() {
  print('Hello world!');
}
```

```
// Hello world in Kotlin

fun main(args : Array<String>) {
    println("Hello, world!")
}
```

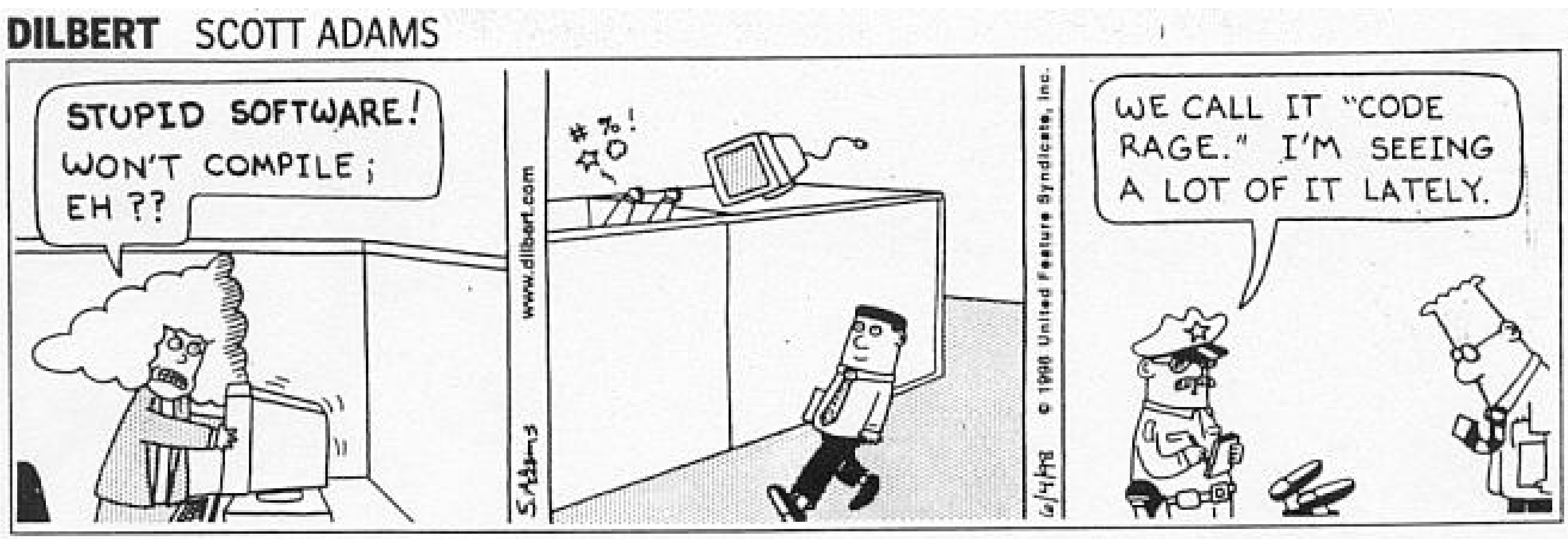
Three Trends

- Declarative programming languages in vogue again
  - Especially functional
- Dynamic Programming languages gained momentum, but …
- Concurrent Programming languages came back on the agenda
  - Reactive programming
    - (a special kind of concurrent programming)

So what can you do in your projects?

- Look at code in the languages you know
- Use Sebesta's Language Evalualtion criteria to those languages
- Look at code in languages you do not know
- Make a list of language features you like
- Make a list of language features you dislike
- Creat some example programs

So how would you like to programme in 20 years?



Languages and Compilers
(SProg og Oversættere)

Lecture 2
Tombstone Diagrams

Bent Thomsen

Department of Computer Science

Aalborg University

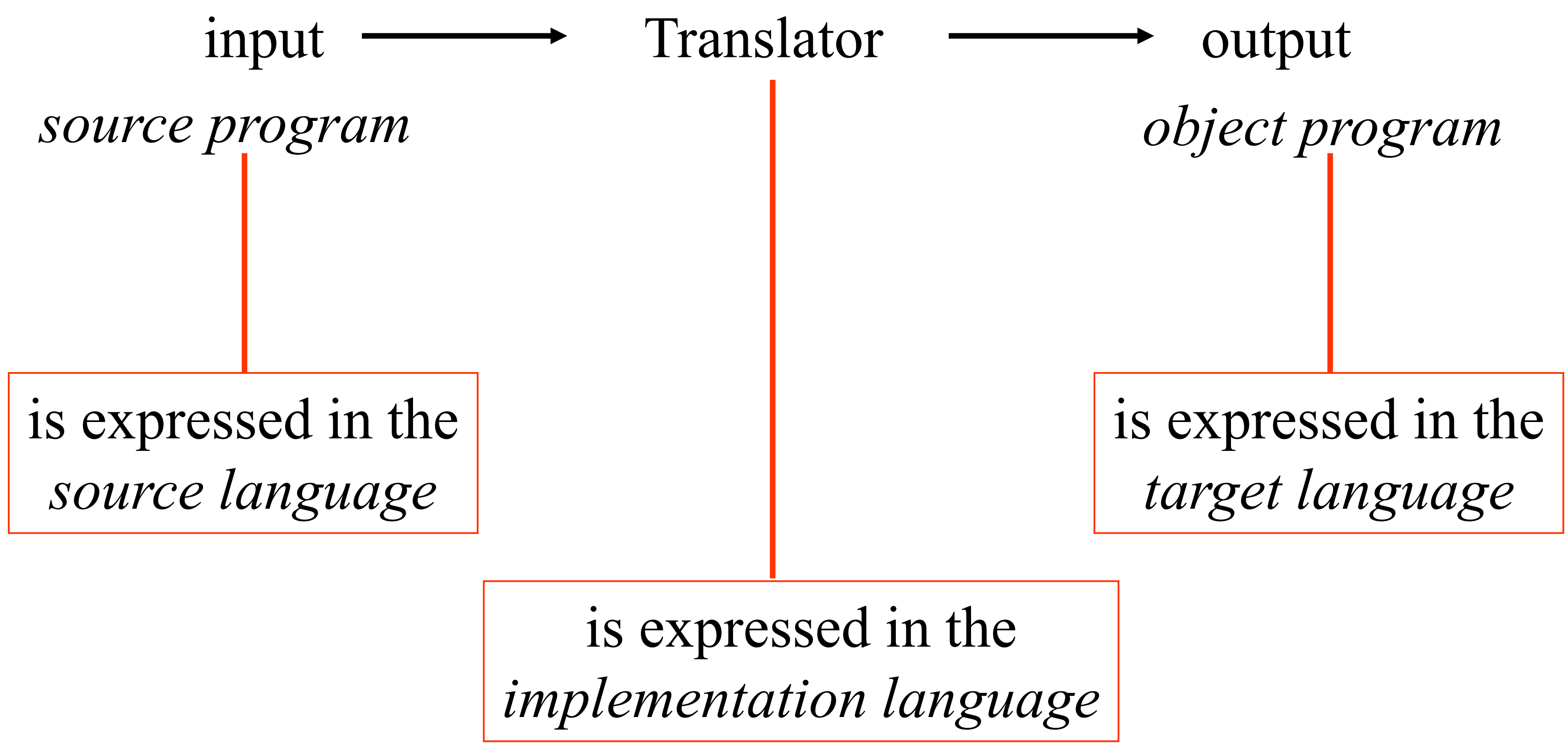
With acknowledgement to Norm Hutchinson whose slides this lecture is based on.

Learning goals

- Knowledge of compilers and interpreters as programs
- Knowledge of tombstone diagrams
- Introduction to Cross compilation
- Introduction to Two stage compiling
- Reasoning about Portability
- Introduction to bootstrapping

Terminology

Q: Which programming languages play a role in this picture?



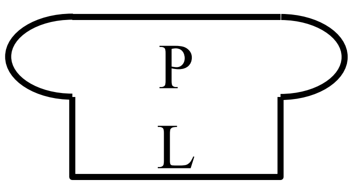
A: All of them!

Tombstone Diagrams

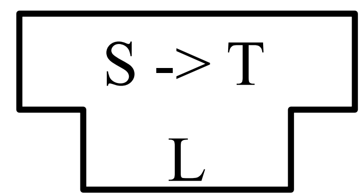
What are they?

– diagrams consisting out of a set of “puzzle pieces” we can use to reason about language processors and programs
– different kinds of pieces
– combination rules (not all diagrams are “well formed”)

Program P implemented in L

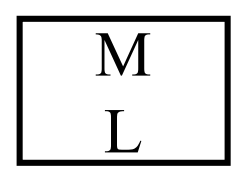


Translator implemented in L

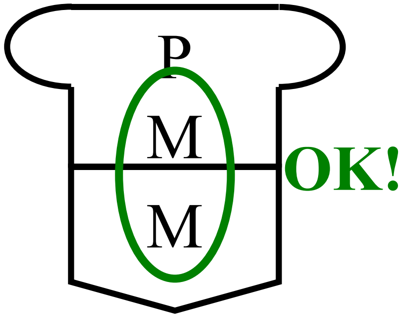


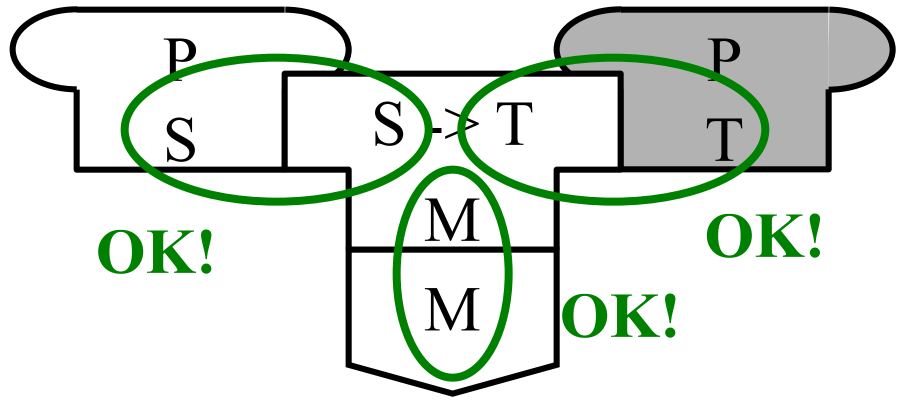
Machine implemented in hardware

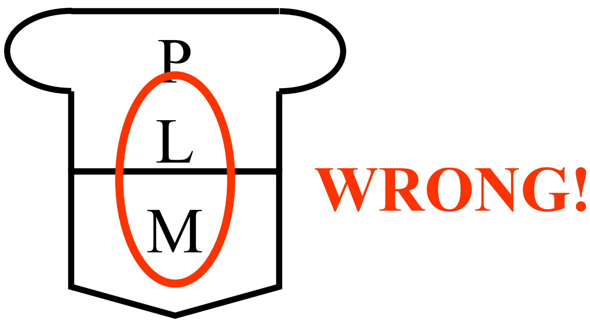
Language interpreter in L

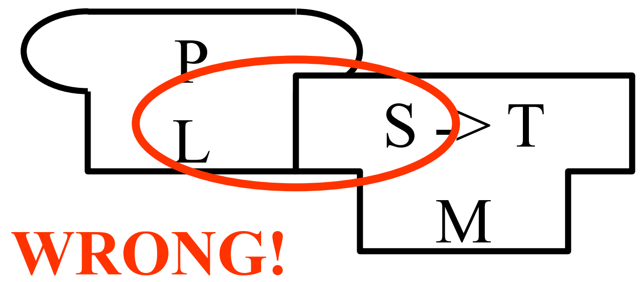


Tombstone diagrams: Combination rules



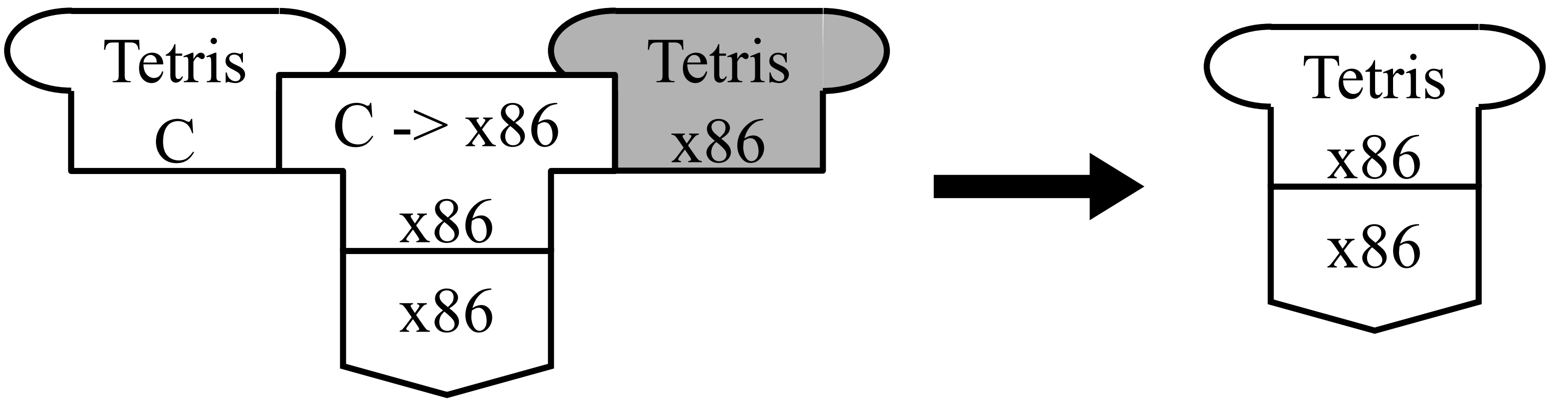






Compilation

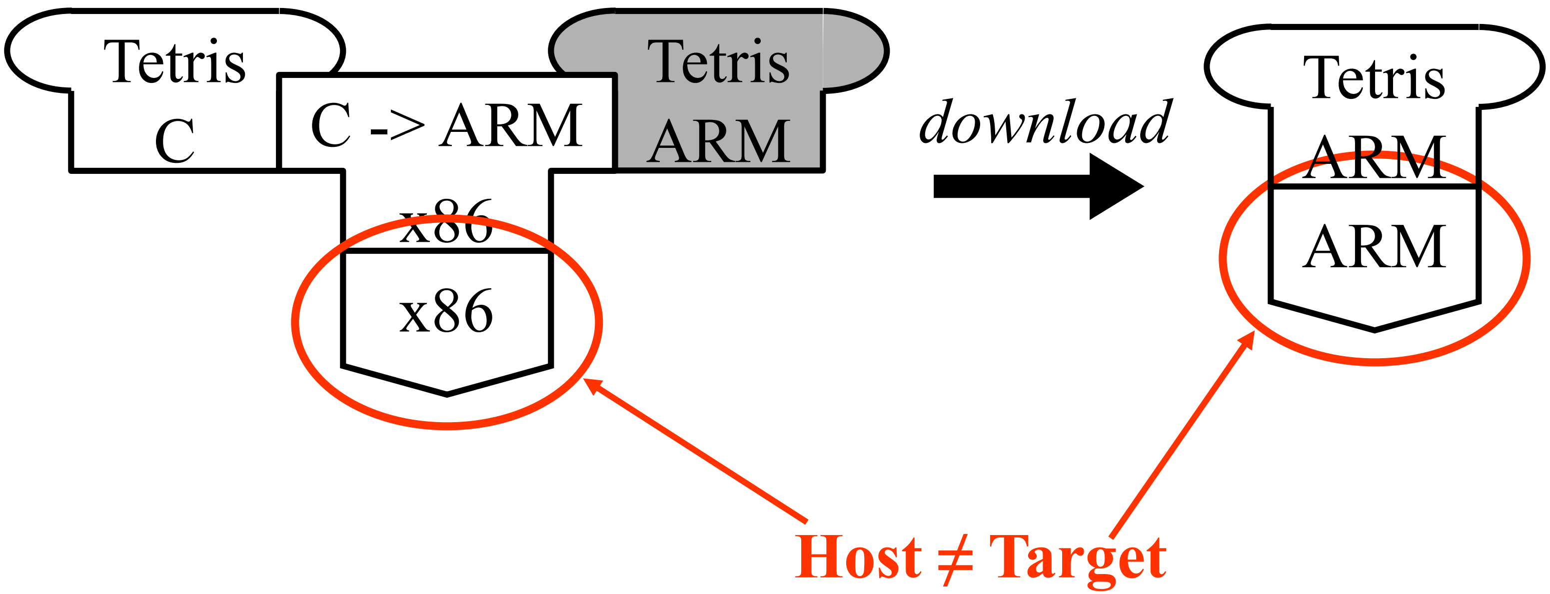
Example: Compilation of C programs on an x86 machine



Cross compilation

Example: A C "cross compiler" from x86 to ARM

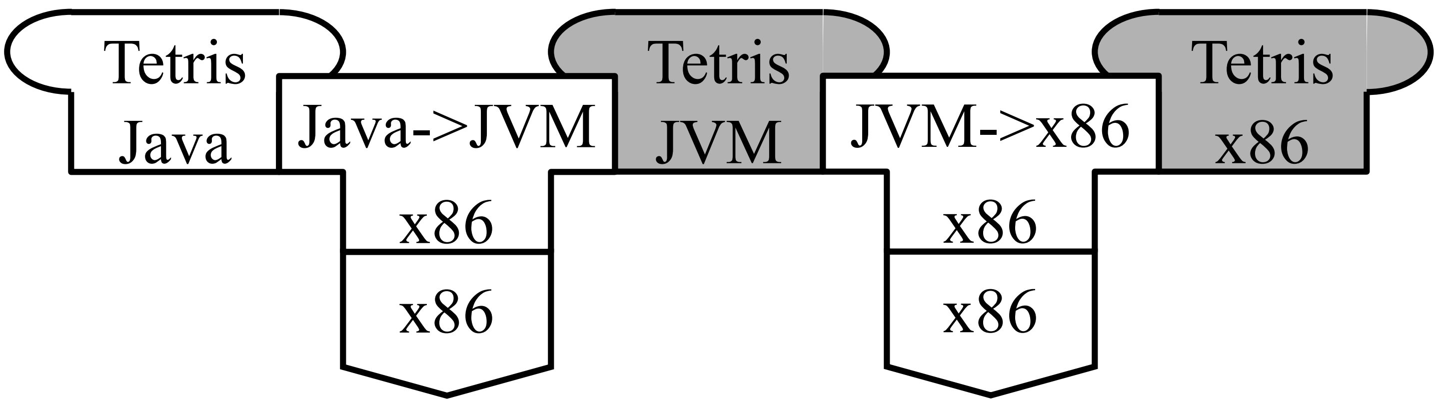
A *cross compiler* is a compiler which runs on one machine (the *host machine*) but emits code for another machine (the *target machine*).



Q: Are cross compilers useful? Why would/could we use them?

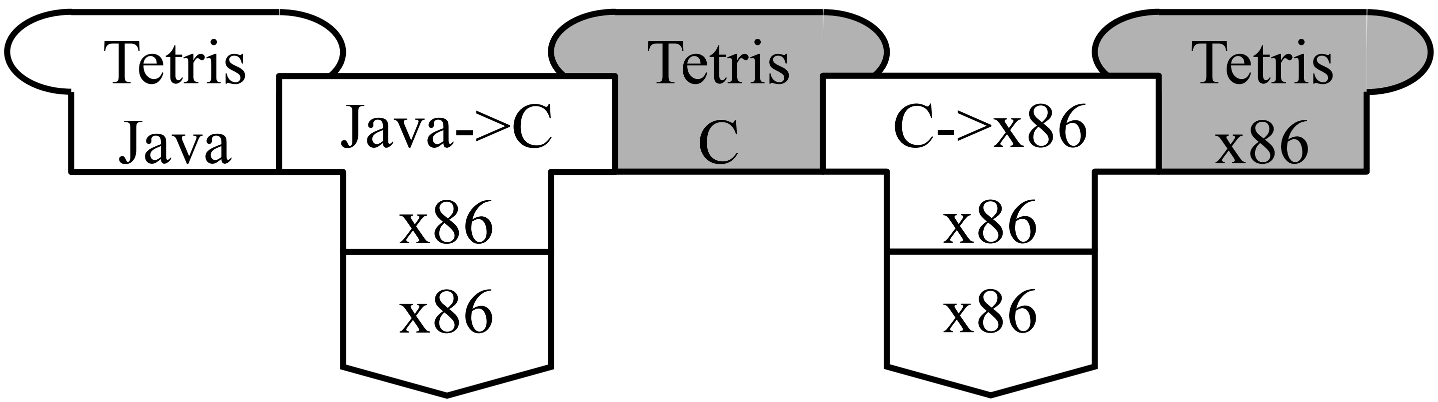
Two Stage Compilation

A *two-stage translator* is a composition of two translators. The output of the first translator is provided as input to the second translator.



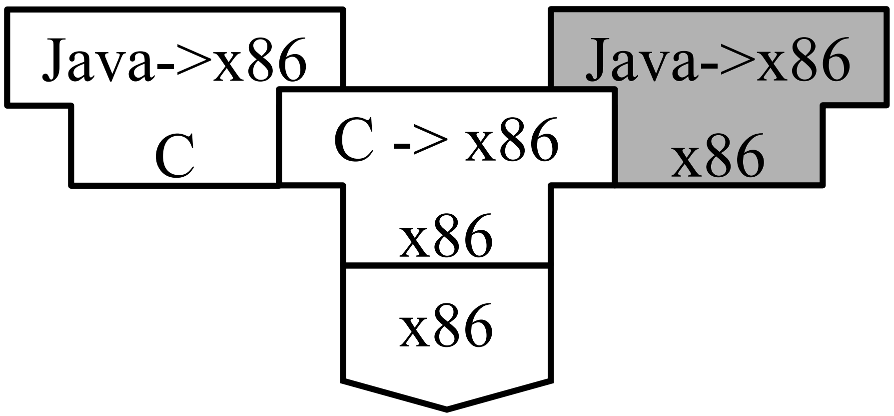
Two Stage Compilation (via C)

A *two-stage translator* is a composition of two translators. The output of the first translator is provided as input to the second translator.



Compiling a Compiler

Observation: A compiler is a program!
Therefore it can be provided as input to a language processor.
Example: compiling a compiler.

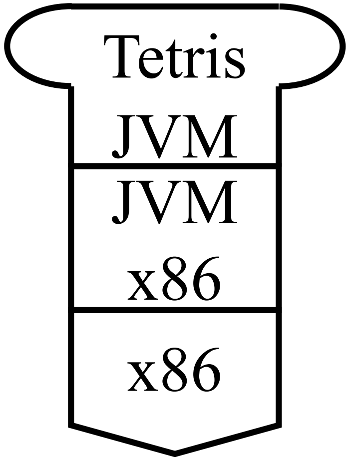


Interpreters

An *interpreter* is a language processor implemented in software, i.e. as a program.

Terminology: *abstract (or virtual) machine* versus *real machine*

Example: The Java Virtual Machine

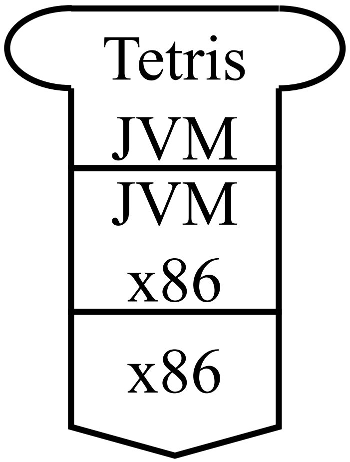


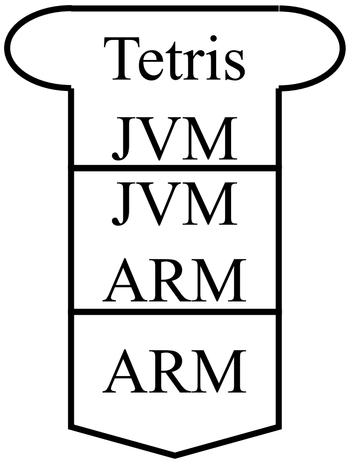
Q: Why are abstract machines useful?

Interpreters

Q: Why are abstract machines useful?

1) Abstract machines provide better platform independence



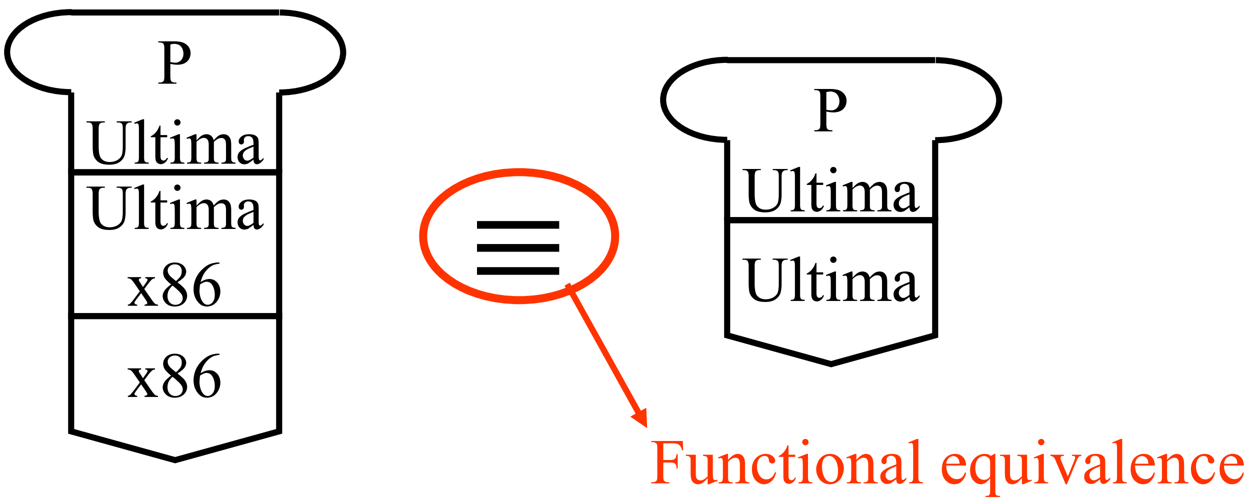


Interpreters

Q: Why are abstract machines useful?

2) Abstract machines are useful for testing and debugging.

Example: Testing the "Ultima" processor using *hardware emulation*



Note: we don't have to implement Ultima emulator in x86 we can use a high-level language and compile it.

Interpreters versus Compilers

Q: What are the tradeoffs between compilation and interpretation?

Compilers typically offer more advantages when

- programs are deployed in a production setting
- programs are “repetitive”
- the instructions of the programming language are complex

Interpreters typically are a better choice when

- we are in a development/testing/debugging stage
- programs are run once and then discarded
- the instructions of the language are simple
- the execution speed is overshadowed by other factors
  - e.g. on a web server where communications costs are much higher than execution speed

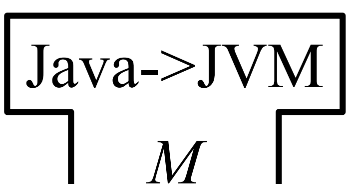
Interpretive Compilers

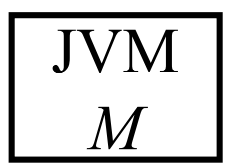
Why?
A tradeoff between fast(er) compilation and a reasonable runtime performance.

How?
Use an "intermediate language"

- more high-level than machine code => easier to compile to
- more low-level than source language => easy to implement as an interpreter

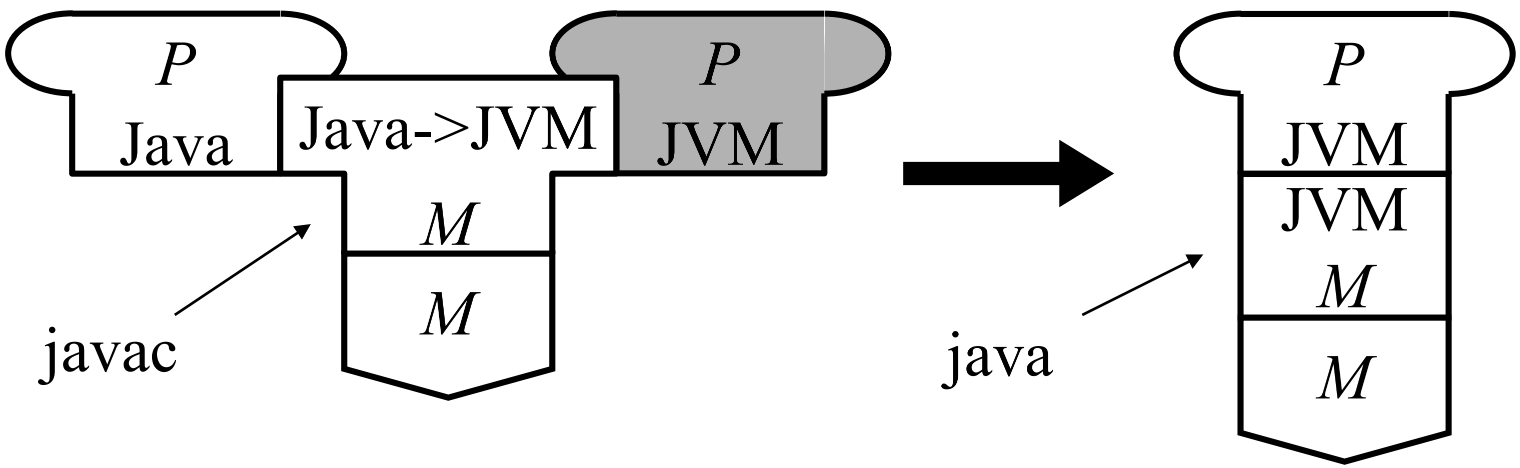
Example: A "Java Development Kit" for machine *M*





Interpretive Compilers

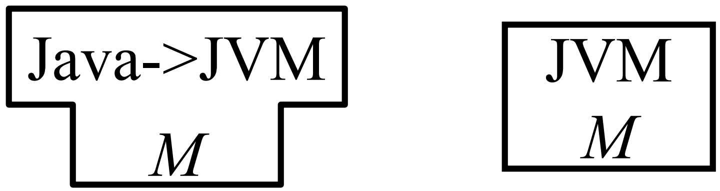
Example: Here is how we use our "Java Development Kit" to run a Java program *P*



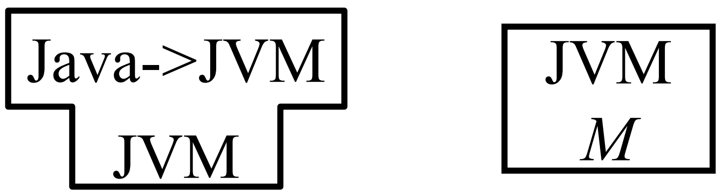
Portable Compilers

Example: Two different “Java Development Kits”

Kit 1:



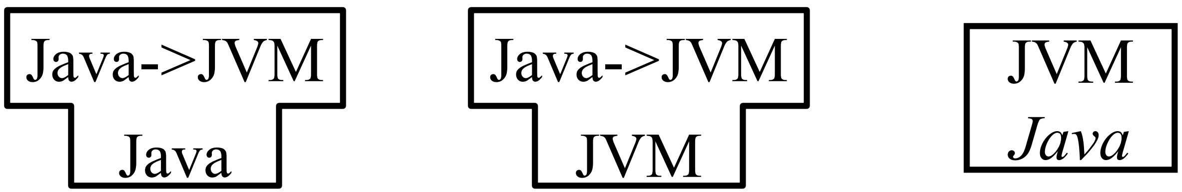
Kit 2:



Q: Which one is “more portable”?

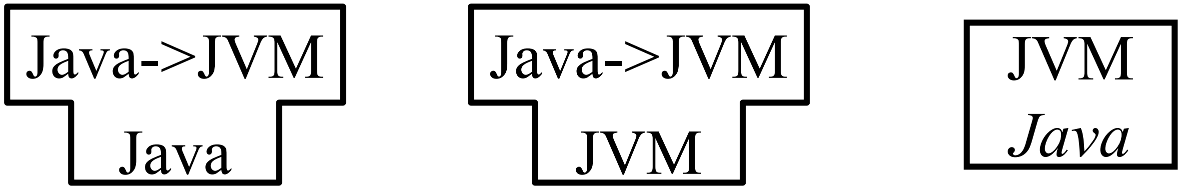
Example: a "portable" compiler kit

Portable Compiler Kit:

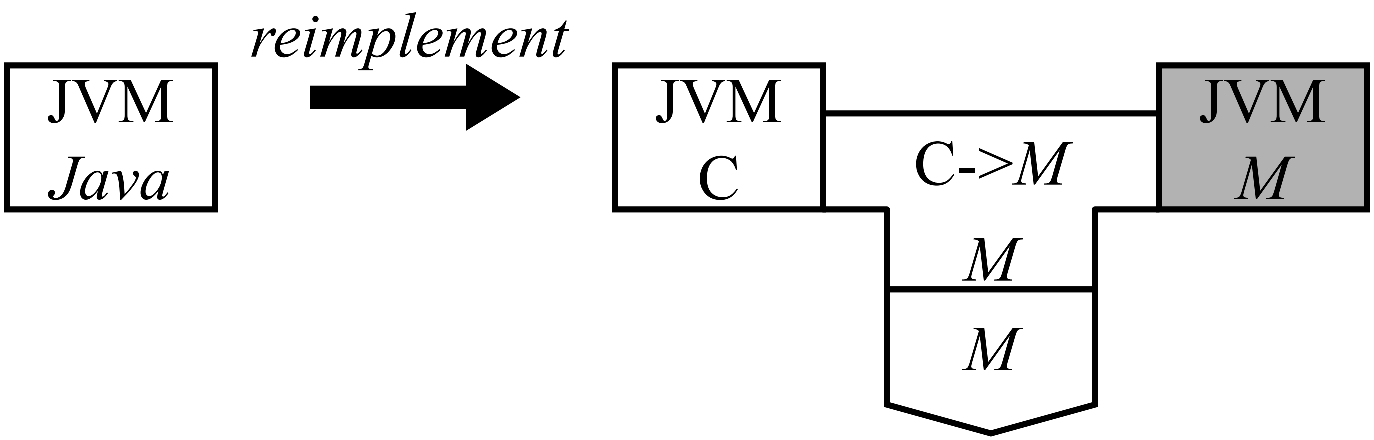


Q: Suppose we want to run this kit on some machine *M*. How could we go about realizing that goal? (with the least amount of effort)

Example: a "portable" compiler kit

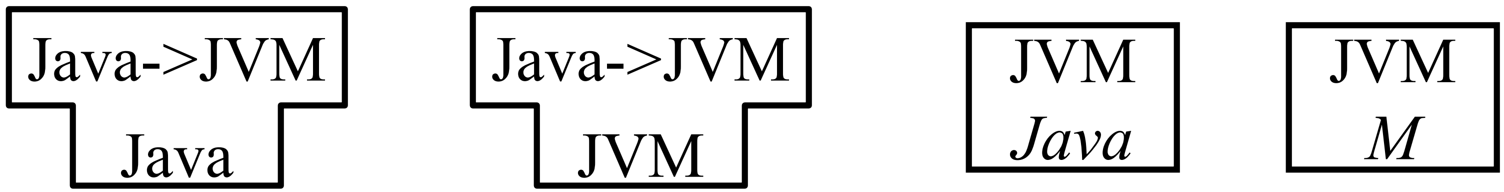


Q: Suppose we want to run this kit on some machine *M*. How could we go about realizing that goal? (with the least amount of effort)

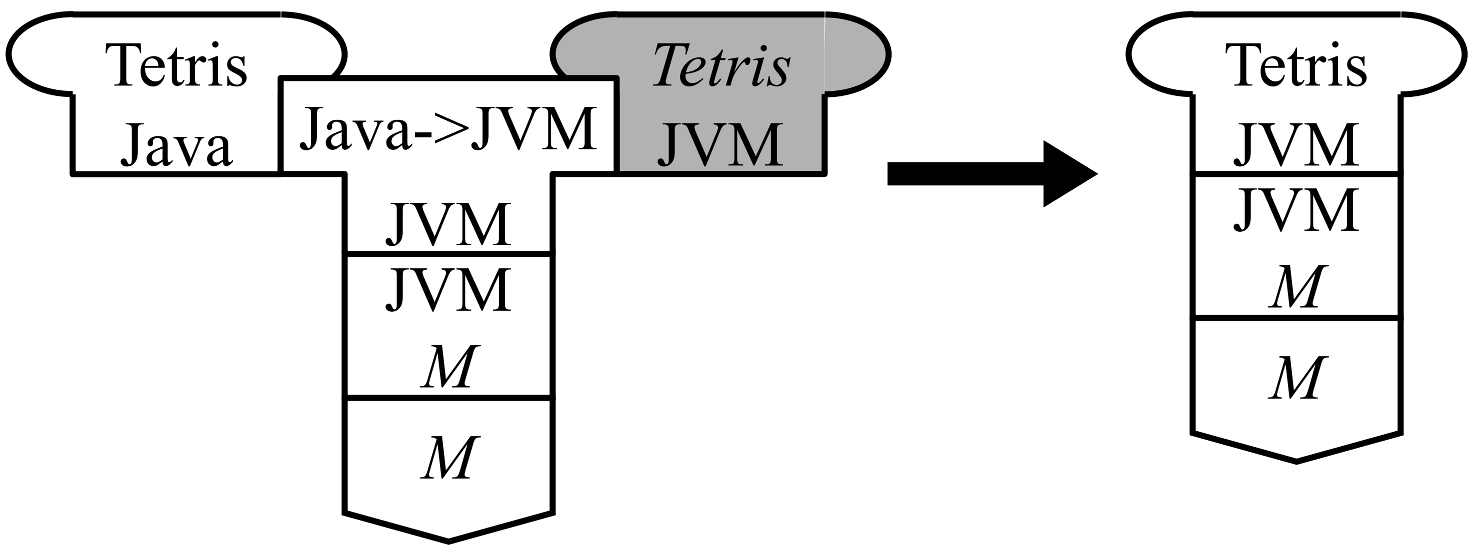


Example: a "portable" compiler kit

This is what we have now:

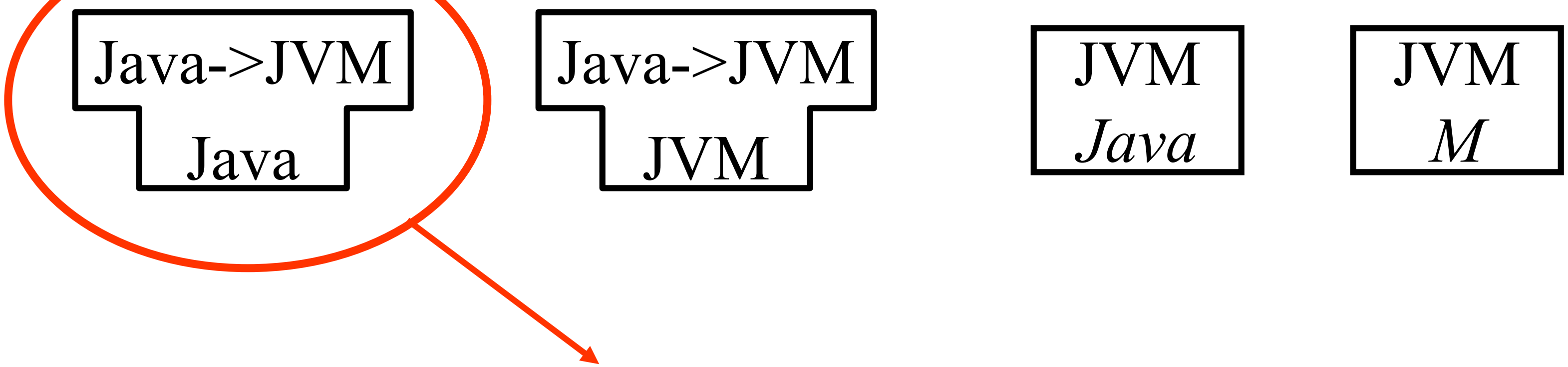


Now, how do we run our Tetris program?

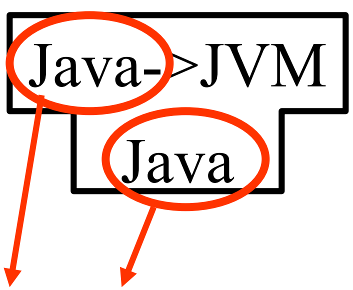


Bootstrapping

Remember our “portable compiler kit”:



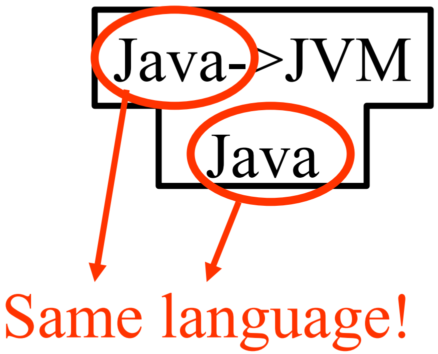
We haven’t used this yet!



Same language!

Q: What can we do with a compiler written in itself? Is that useful at all?

Bootstrapping



Q: What can we do with a compiler written in itself? Is that useful at all?

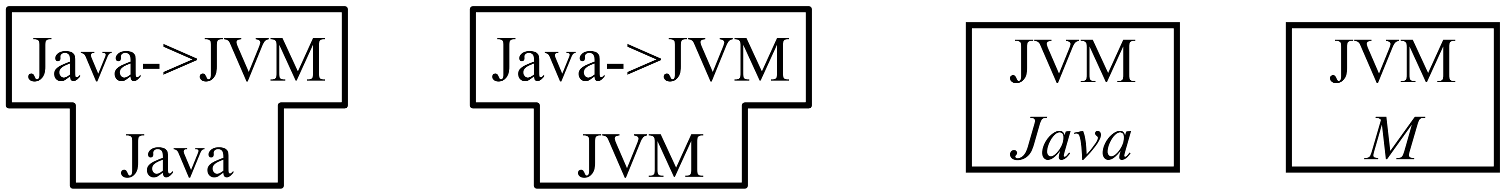
- By implementing the compiler in (a subset of) its own language, we become less dependent on the target platform => more portable implementation.
- But… "chicken and egg problem"? How do to get around that?

=> BOOTSTRAPPING: requires some work to make the first "egg".

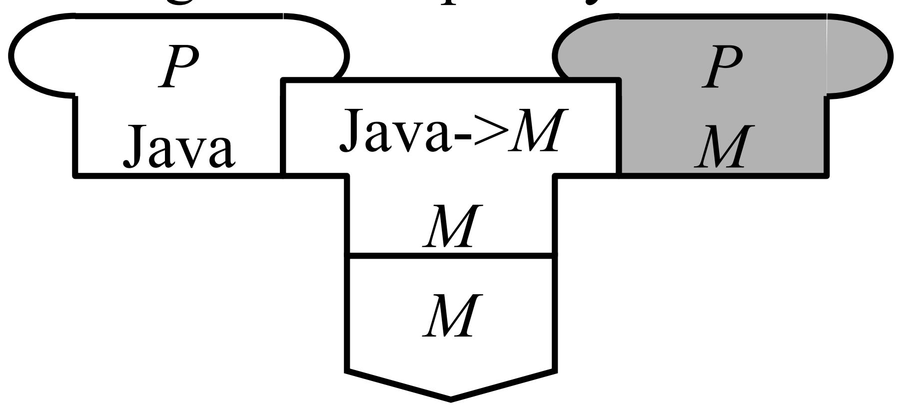
There are many possible variations on how to bootstrap a compiler written in its own language.

Bootstrapping an Interpretive Compiler to Generate *M* code

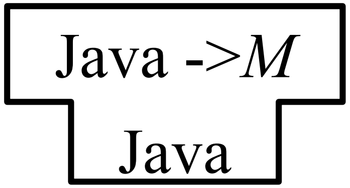
Our "portable compiler kit":

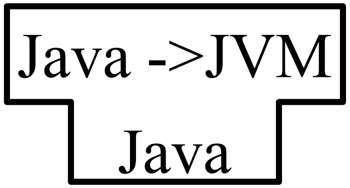


Goal: we want to get a "completely native" Java compiler on machine *M*

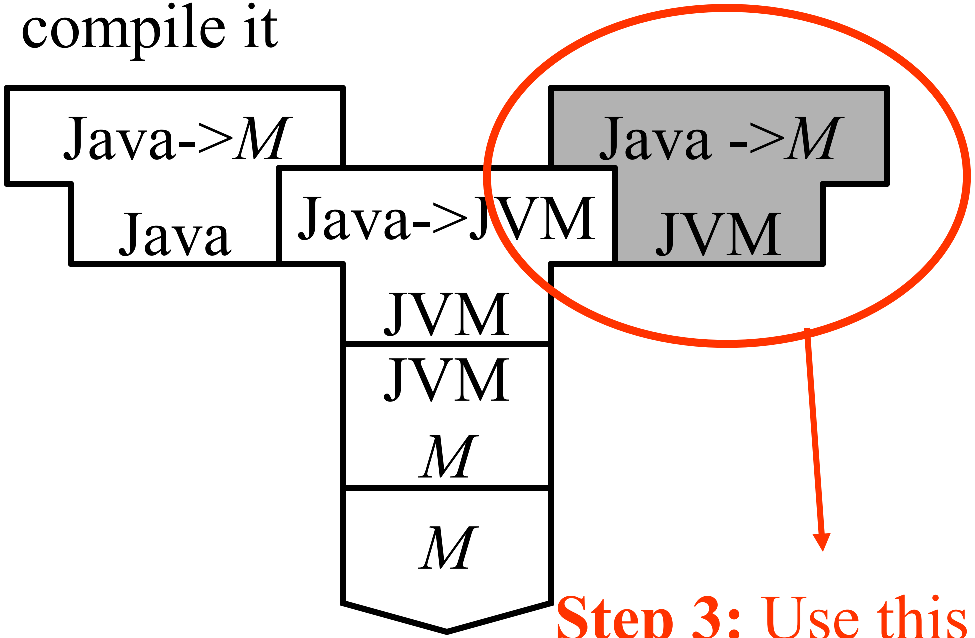


Bootstrapping an Interpretive Compiler to Generate *M* code (first approach)

Step 1: implement 

by rewriting 

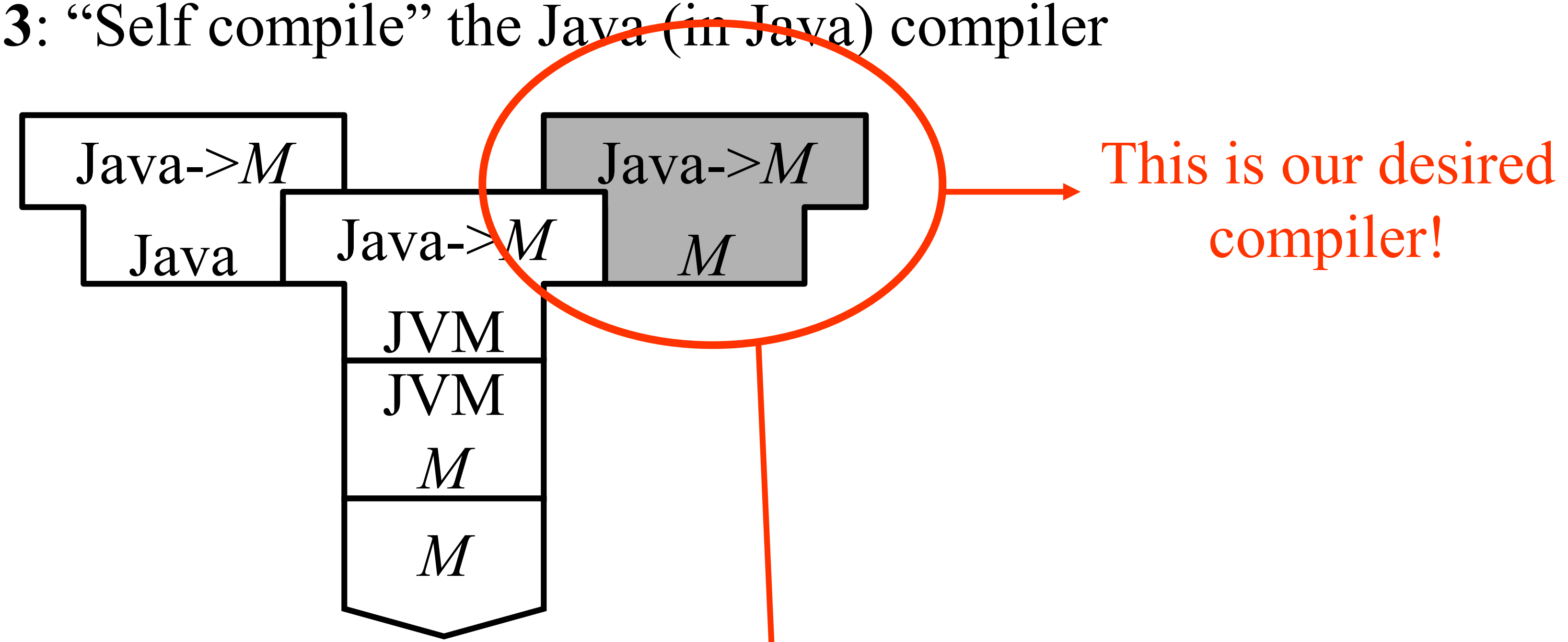
Step 2: compile it



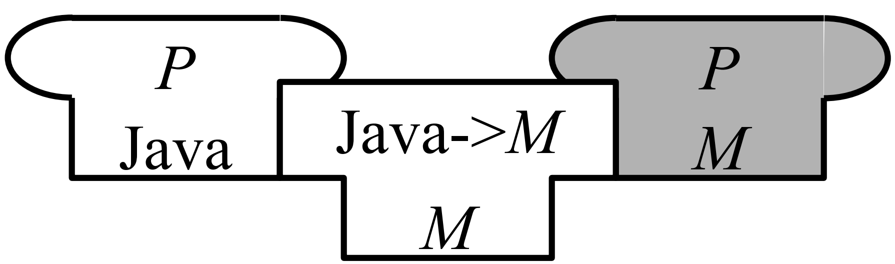
Step 3: Use this to compile again

Bootstrapping an Interpretive Compiler to Generate *M* code (first approach)

Step 3: "Self compile" the Java (in Java) compiler

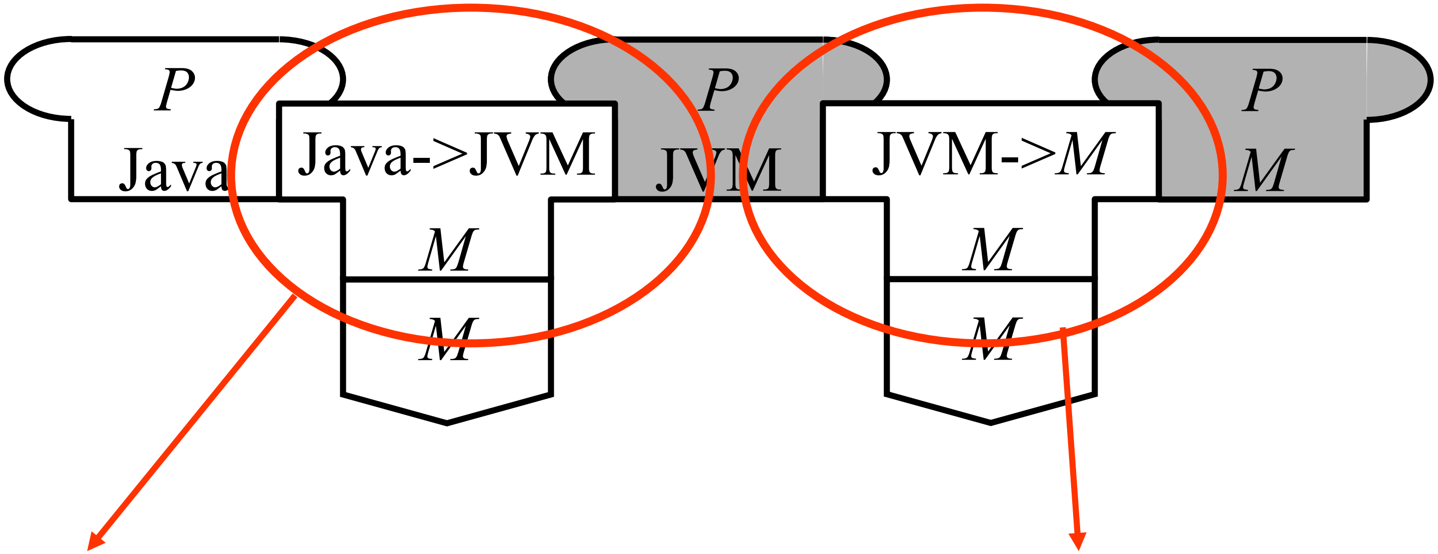


Step 4: use this to compile the P program

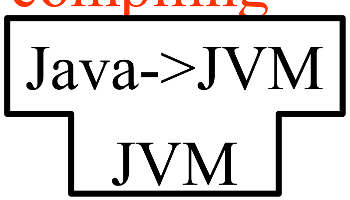


Bootstrapping an Interpretive Compiler to Generate *M* code (second approach)

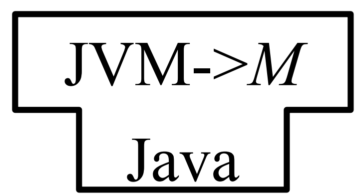
Idea: we will build a two-stage Java -> *M* compiler.



We will make this by compiling



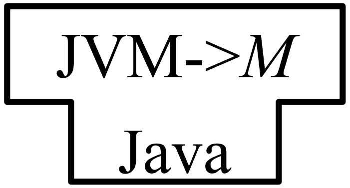
To get this we implement



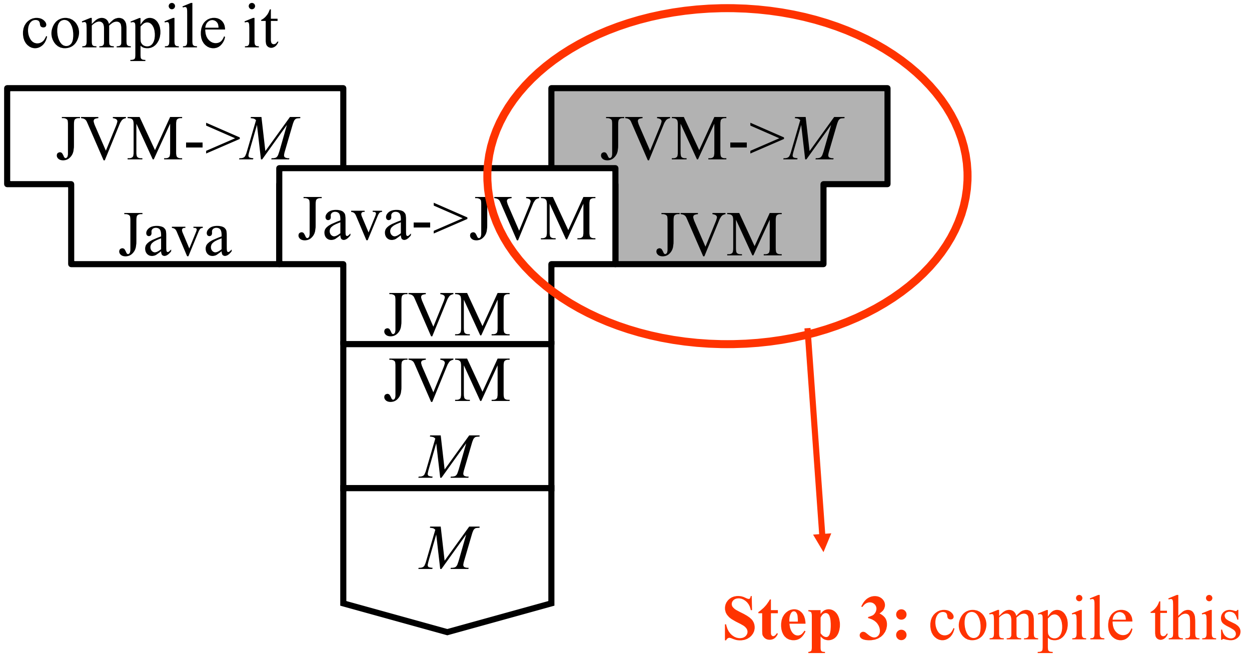
and compile it

Bootstrapping an Interpretive Compiler to Generate *M* code (second approach)

Step 1: implement

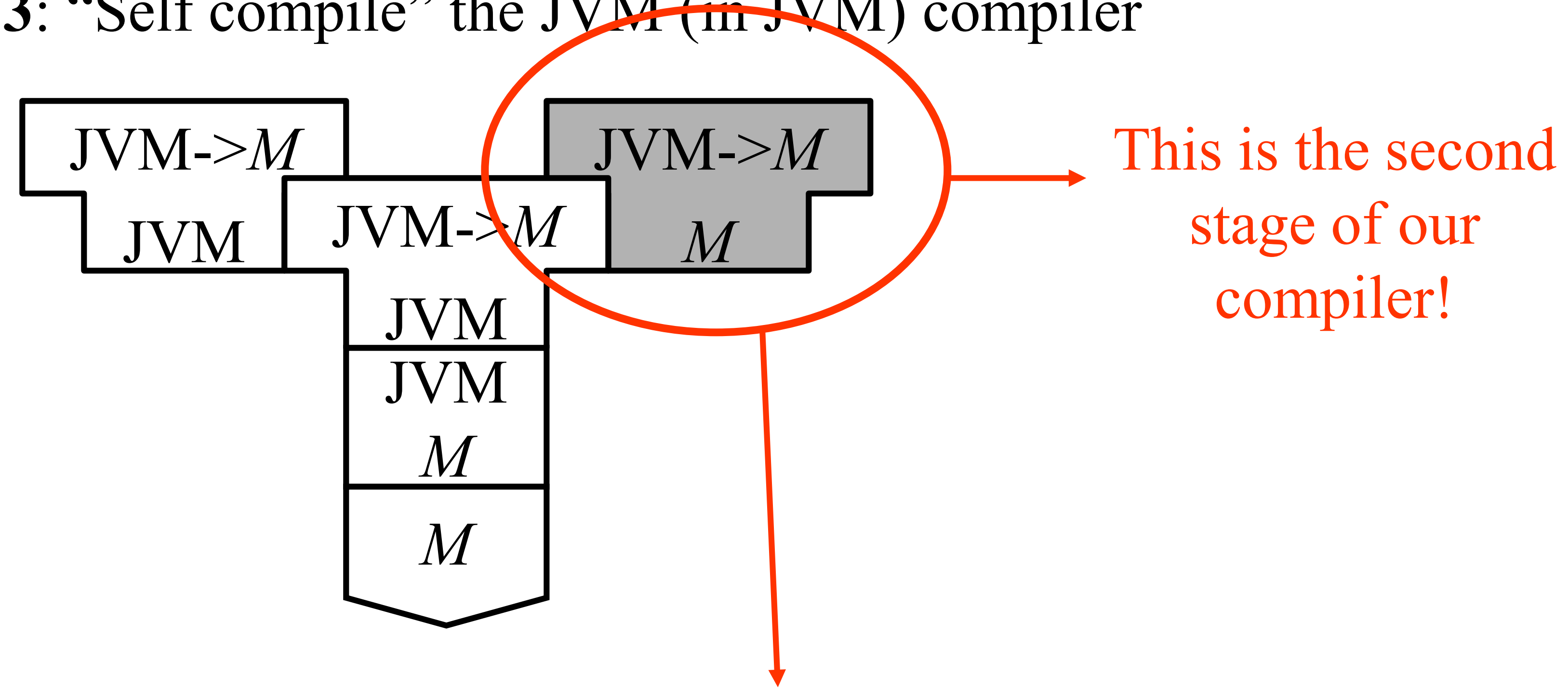


Step 2: compile it



Bootstrapping an Interpretive Compiler to Generate *M* code (second approach)

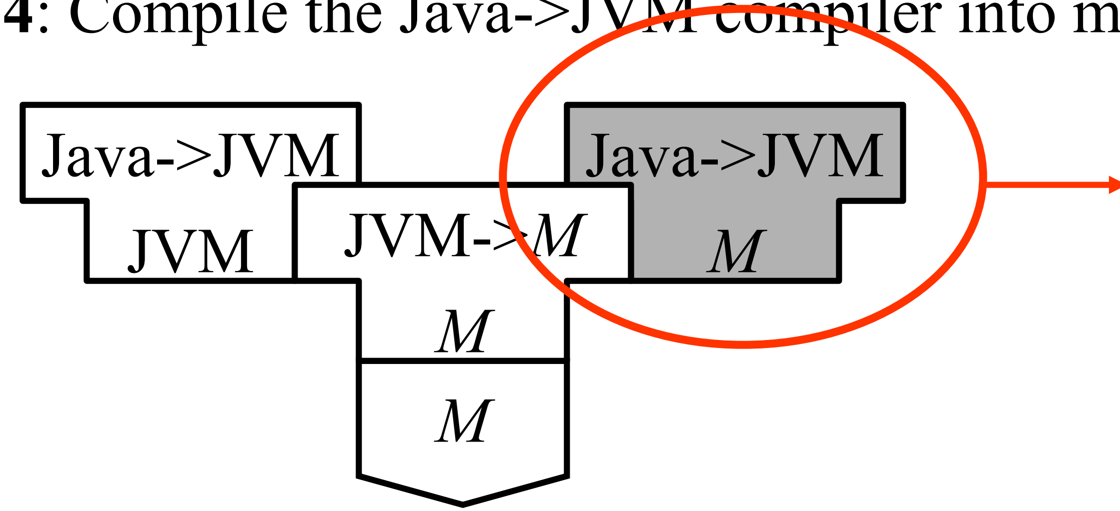
Step 3: "Self compile" the JVM (in JVM) compiler



Step 4: use this to compile the Java compiler

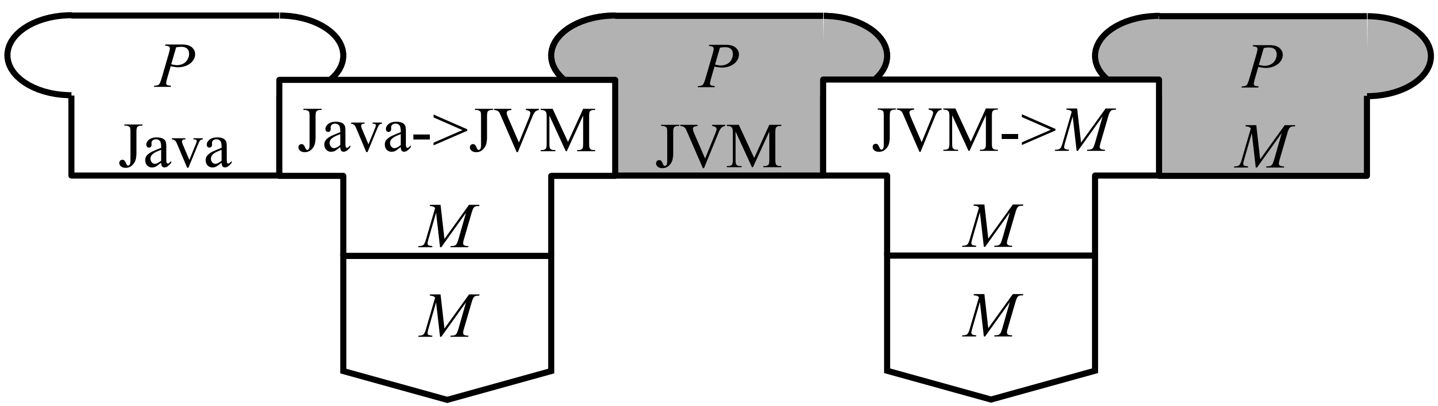
Bootstrapping an Interpretive Compiler to Generate *M* code

Step 4: Compile the Java->JVM compiler into machine code



The first stage of our compiler!

We are DONE!



Bootstrapping to Improve Efficiency

The efficiency of programs and compilers:

Efficiency of programs:

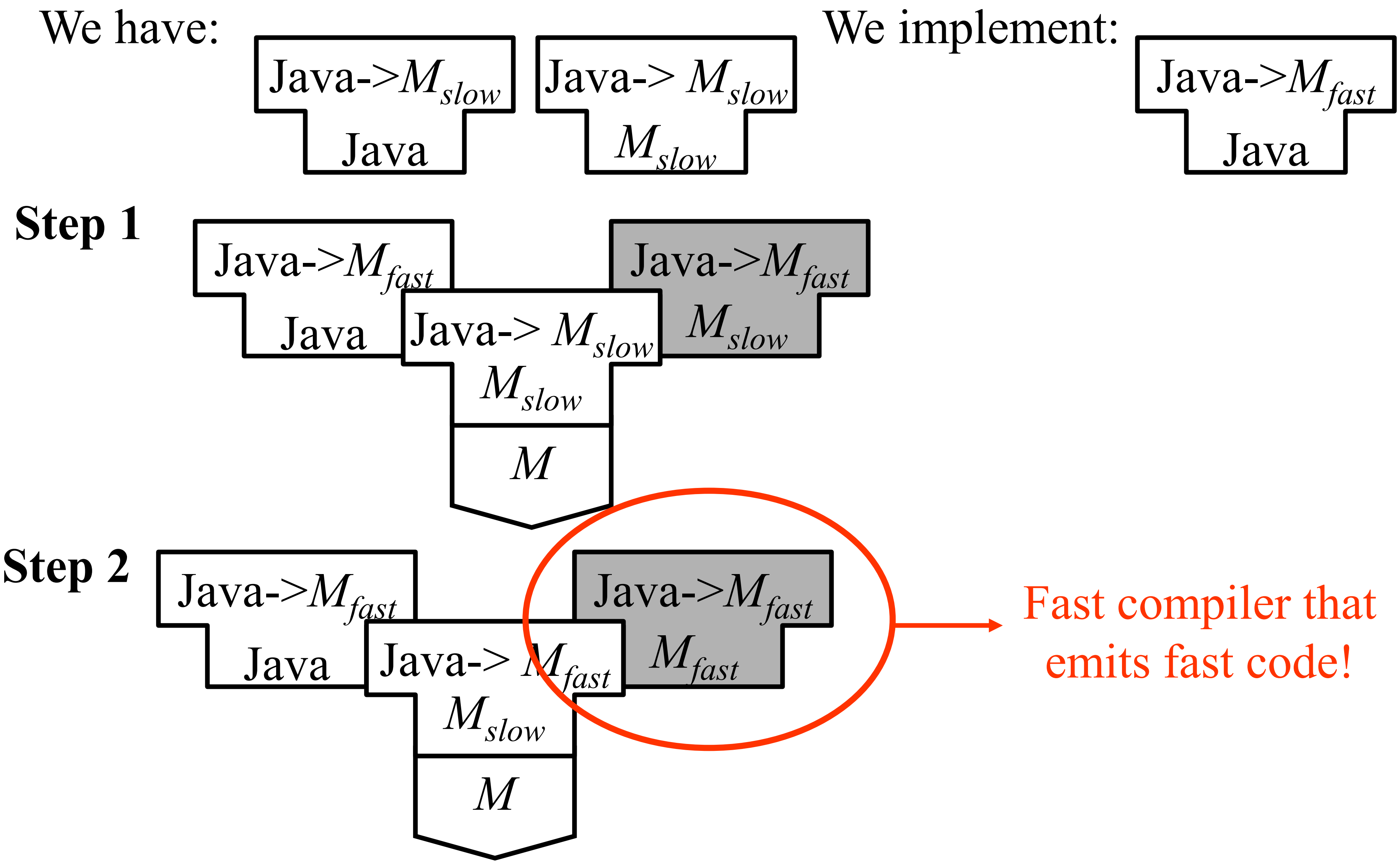
- memory usage
- runtime

Efficiency of compilers:

- Efficiency of the compiler itself
- Efficiency of the emitted code

Idea: We start from a simple compiler (generating inefficient code) and develop more sophisticated versions of it. We can then use bootstrapping to improve performance of the compiler.

Bootstrapping to Improve Efficiency



Conclusion

- To write a good compiler you may be writing several simpler ones first
- You have to think about the source language, the target language and the implementation language.
- Strategies for implementing a compiler
  1. Write it in machine code
  2. Write it in a lower level language and compile it using an existing compiler
  3. Write it in the same language that it compiles and bootstrap
- The work of a compiler writer is never finished, there is always version 1.x and version 2.0 and …

AtoCC Demo

Languages and Compilers
(SProg og Oversættere)

Lecture 3
The ac language and compiler

Bent Thomsen

Department of Computer Science

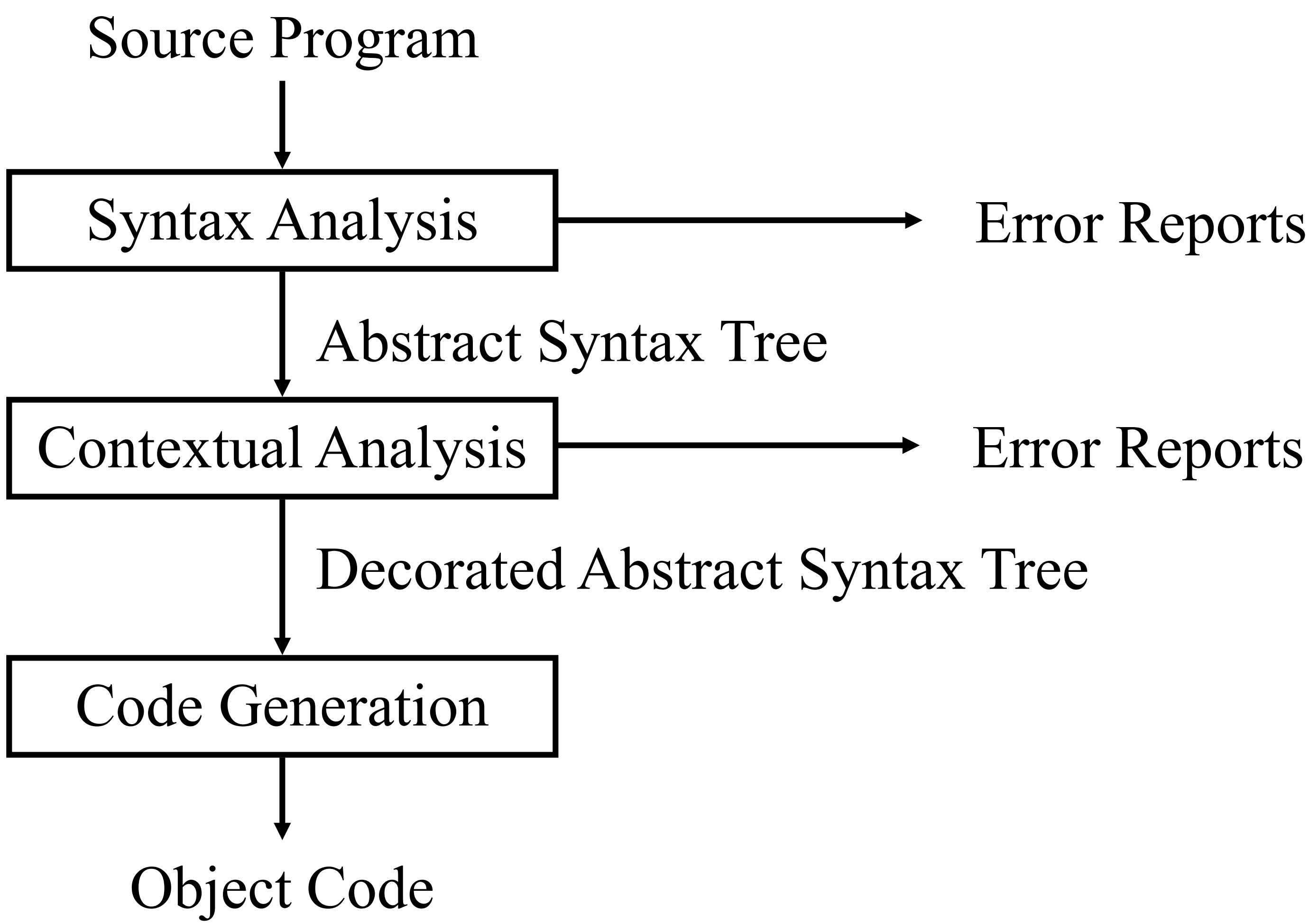
Aalborg University

With acknowledgement to H. J. Wang whose slides this lecture is based on.

Learning goals

- Get an overview of a simple language (ac)
- Get an introduction to language definition
- Get an overview of the compilation process for a simple language
- Get a quick overview of a compiler's phases and their associated data structures

The “Phases” of a Compiler



Different Phases of a Compiler

The different phases can be seen as different transformation steps to transform source code into object code.

The different phases correspond roughly to the different parts of the language specification:

- Syntax analysis <-> Syntax
    - Lexical analysis <-> Regular Expressions
    - Parsing <-> Context Free Grammar
- Contextual analysis <-> Contextual constraints
    - Scope checking <-> Scope rules (static semantics)
    - Type checking <-> Type rules (static semantics)
- Code generation <-> Semantics (dynamic semantics)

Organization of a Compiler

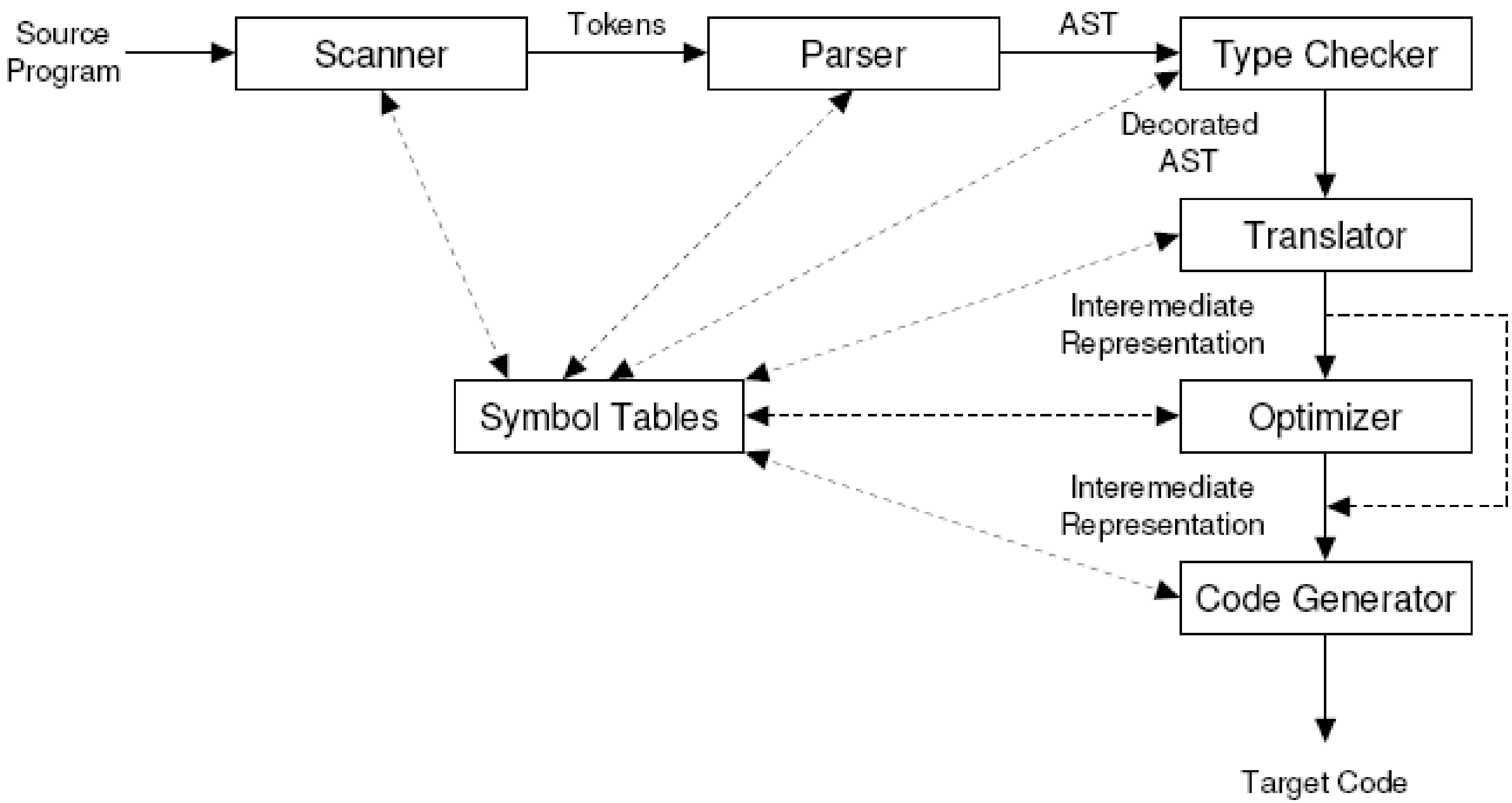


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

Phases of a Simple Compiler

- Scanner: source program -> tokens
  - Part of Syntax analysis phase
  - Fischer et. Al. Chap. 3
- Parser: tokens -> abstract syntax tree (AST)
  - Part of Syntax analysis phase
  - Fischer et. Al. Chap. 5 & 6
- Symbol table: created from AST
  - Part of contextual analysis phase
  - Fischer et. Al. Chap. 8
- Semantic analysis: AST decoration
  - Part of contextual analysis phase
  - Fischer et. Al. Chap. 9
- Translation (Code generation)
  - Part of code generation phase
  - Fischer et. Al. Chap. 11 and Chap 13.

An Informal Definition of the ac Language

- *ac*: adding calculator
- Types
  - integer
  - float: allows 5 fractional digits after the decimal point
  - Automatic type conversion from integer to float
- Keywords
  - f: float
  - i: integer
  - p: print
- Variables
  - 23 names from lowercase Roman alphabet except the three reserved keywords f, i, and p
- Monolitic scope, i.e. names are visible in the program when they are declared
  - Note more complex languages may have nested scopes
    - e.g. in C we can write { int x; … { int x; … x =5; … } … x =x +1; …}
- Target of translation: *dc* (desk calculator)
  - Reverse Polish notation (RPN)

Example Program

```
f  b                //declare variable b as float
i  a                //declare variable a as int
a  =   5            //assign a the value 5
b  =   a + 3.2      //assign b the result of
                    //calculating a + 3.2
p  b                //print the content of b
```

An Example ac Program

- Example ac program:
  - f b
    i a
    a = 5
    b = a + 3.2
    p b
- Corresponding dc code
  - 5
    sa
    la
    3.2
    +
    sb
    lb
    p

Note that DC is a stack machine just like the JVM, CLR and PostScript

Formal Definition of ac

- Syntax specification:
  - context-free grammar (CFG)
  - (Chap. 4)
- Token specification:
  - Regular Expressions (RE)
  - (Sec. 3.2)

- Note no formal definition of Type Rules or Runtime semantics (in Fischer et. Al.)

A sketch SOS for ac

$P \rightarrow Dcl\ Stm$

$Dcl \rightarrow \text{floatdcl}\ id\ Dcl$
$\quad \mid \text{int}\ id\ Dcl$
$\quad \mid \varepsilon$

$Stm \rightarrow id\ \text{assign}\ Exp$
$\quad \mid \text{print}\ id$
$\quad \mid stm\ stm$
$\quad \mid \text{skip}$

$Exp \rightarrow Exp_1 + Exp_2$
$\quad \mid Exp_0 - Exp_2$
$\quad \mid id$
$\quad \mid inum$
$\quad \mid fnum$

$$\frac{Env \vdash E_1 : int \quad Env \vdash E_2 : int}{Env \vdash E_1 + E_2 : int}$$

$$\frac{Env \vdash E_1 : float \quad Env \vdash E_2 : float}{Env \vdash E_1 + E_2 : float}$$

$$\frac{Env \vdash E : int}{Env \vdash E : float}$$

$E \vdash id : t$ where $Env(id) = t$

$E \vdash \varepsilon : ok$

$$\frac{E[x \rightarrow t] \vdash Dcl : ok}{E \vdash t\ x\ Dcl : ok}$$

$$\frac{S \vdash Exp_1 \rightarrow v_1 \quad S \vdash Exp_2 \rightarrow v_2}{S \vdash Exp_1 + Exp_2 \rightarrow v}$$ where $v = v_1 + v_2$

$S \vdash fn \rightarrow v$ if $N[fn] = v$

$S \vdash x \rightarrow v$ if $S(x) = v$

$\langle x = E, s, o \rangle \rightarrow (s[x \mapsto v], o)$ if $S \vdash E \rightarrow v$

$\langle p\ x, s, o \rangle \rightarrow (s, s(x) : o)$

$$\frac{\langle S_1, s, o \rangle \rightarrow (s', o') \quad \langle S_2, s', o' \rangle \rightarrow (s'', o'')}{\langle S_1\ S_2, s, o \rangle \rightarrow (s'', o'')}$$

$(skip, s, o) \rightarrow (s, o)$

Syntax Specification

1	Prog	→	Dcls Stmts $
2	Dcls	→	Dcl Dcls
3		|	λ
4	Dcl	→	floatdcl id
5		|	intdcl id
6	Stmts	→	Stmt Stmts
7		|	λ
8	Stmt	→	id assign Val Expr
9		|	print id
10	Expr	→	plus Val Expr
11		|	minus Val Expr
12		|	λ
13	Val	→	id
14		|	inum
15		|	fnum

Figure 2.1: Context-free grammar for ac.

Context Free Grammar

- CFG:
  - A set of productions or rewriting rules
  - E.g.: Stmt → id assign Val Expr
    | print id
  - Two kinds of symbols
    - Terminals: cannot be rewritten
      - E.g.: id, assign, print
      - Empty or null string: λ - some references use ε for empty string
      - End of input stream or file: $
    - Nonterminals:
      - E.g.: Val, Expr
      - Start symbol: Prog
  - Left-hand side (LHS)
  - Right-hand side (RHS)

Example Program

```
f  b               //declare variable b as float
i  a               //declare variable a as int
a  =   5           //assign a the value 5
b  =   a + 3.2     //assign b the result of
                   //calculating a + 3.2
p  b               //print the content of b
$                  //symbol used to signal
                   //end of input
```

Step	Sentential Form	Production Number
1	⟨Prog⟩	
2	⟨Dcls⟩ Stmts $	1
3	⟨Dcl⟩ Dcls Stmts $	2
4	floatdcl id ⟨Dcls⟩ Stmts $	4
5	floatdcl id ⟨Dcl⟩ Dcls Stmts $	2
6	floatdcl id intdcl id ⟨Dcls⟩ Stmts $	5
7	floatdcl id intdcl id ⟨Stmts⟩ $	3
8	floatdcl id intdcl id ⟨Stmt⟩ Stmts $	6
9	floatdcl id intdcl id id assign ⟨Val⟩ Expr Stmts $	8
10	floatdcl id intdcl id id assign inum ⟨Expr⟩ Stmts $	14
11	floatdcl id intdcl id id assign inum ⟨Stmts⟩ $	12
12	floatdcl id intdcl id id assign inum ⟨Stmt⟩ Stmts $	6
13	floatdcl id intdcl id id assign inum id assign ⟨Val⟩ Expr Stmts $	8
14	floatdcl id intdcl id id assign inum id assign id ⟨Expr⟩ Stmts $	13
15	floatdcl id intdcl id id assign inum id assign id plus ⟨Val⟩ Expr Stmts $	10
16	floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Expr⟩ Stmts $	15
17	floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Stmts⟩ $	12
18	floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Stmt⟩ Stmts $	6
19	floatdcl id intdcl id id assign inum id assign id plus fnum print id ⟨Stmts⟩ $	9
20	floatdcl id intdcl id id assign inum id assign id plus fnum print id $	7

1	Prog	→	Dcls Stmts $
2	Dcls	→	Dcl Dcls
3		|	λ
4	Dcl	→	floatdcl id
5		|	intdcl id
6	Stmts	→	Stmt Stmts
7		|	λ
8	Stmt	→	id assign Val Expr
9		|	print id
10	Expr	→	plus Val Expr
11		|	minus Val Expr
12		|	λ
13	Val	→	id
14		|	inum
15		|	fnum

f b i a a = 5 b = a + 3.2 p b $

Figure 2.2: Derivation of an ac program using the grammar in Figure 2.1.

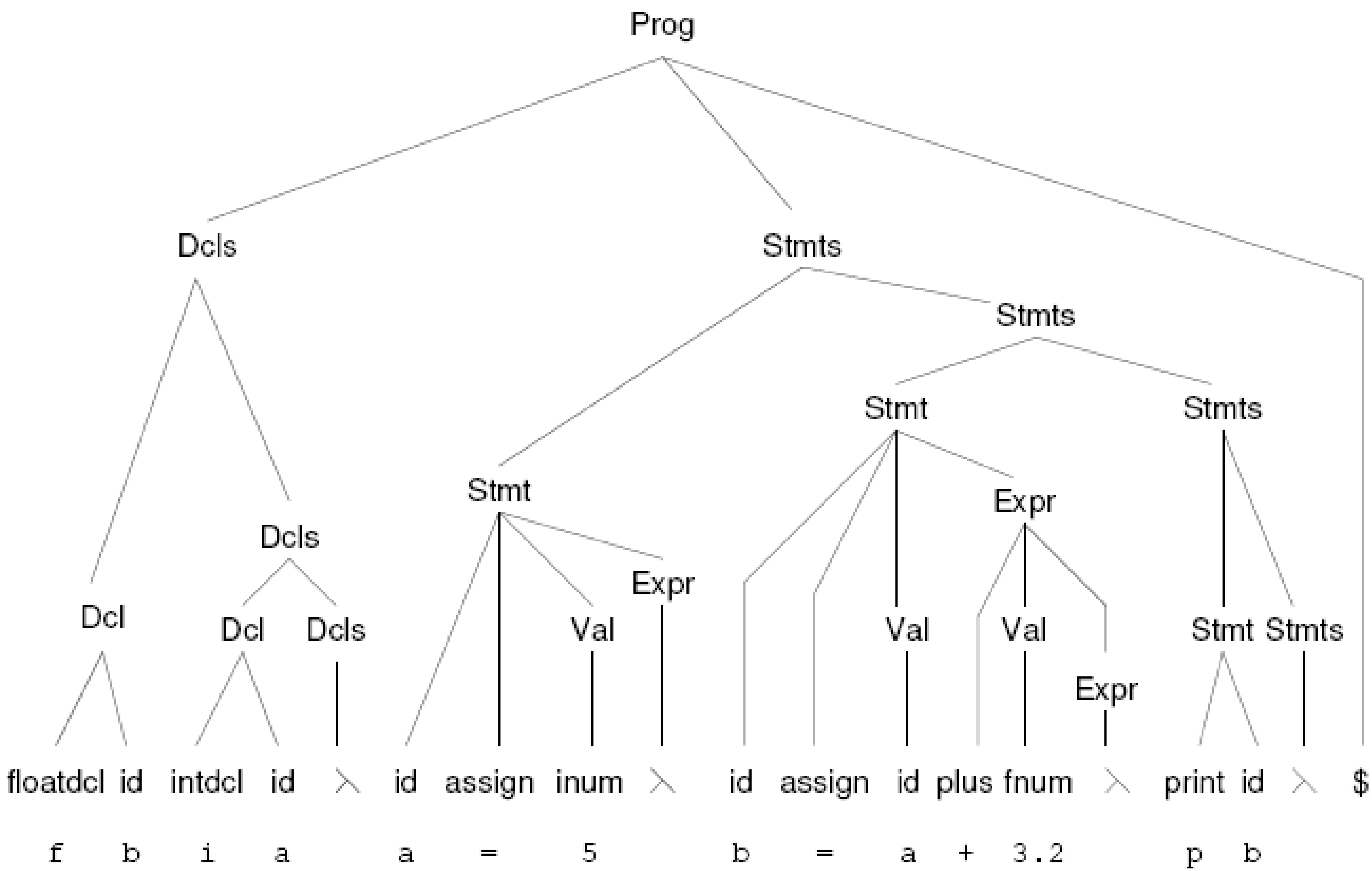


Figure 2.4: An ac program and its parse tree.

Definition of ac language

Regular expression specifies **Token**

- The actual input characters that correspond to each terminal symbol (called token) are specified by regular expression.

- For example:
  - **assign** symbol as a terminal, which appears in the input stream as "=" character.
  - The terminal **id (identifier)** could be any alphabetic character except f, i, or p, which are reserved for special use in ac. It is specified as [a-e] | [g-h]] | [j-o] | [q-z]

- Regular expression will be covered in Ch. 3.
- Also need to specify which symbols to ignore
  - E.g. blanks, tabs, comments (sometimes called Ignore Tokens)

Token Specification for ac

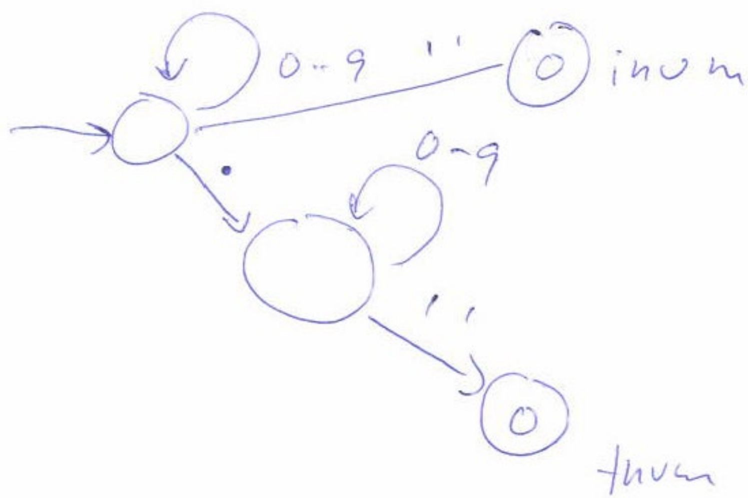
Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	[a – e] | [g – h] | [j – o] | [q – z]
assign	"="
plus	"+"
minus	"-"
inum	$[0-9]^+$
fnum	$[0-9]^+.[0-9]^+$
blank	$(" ")^+$

Figure 2.3: Formal definition of ac tokens.

Note: In most languages id is a sequence of letters and numbers starting
With a letter defined as [a-z]([a-z]|[0-9])*

Tokens and FSA

inum	$[0-9]^+$
fnum	$[0-9]^*.[0-9]^+$
blank	$(" ")^+$



Phases of an ac compiler

- Scanning/lexing
  - The **scanner** reads a source **ac** program as a text file and produces a stream of tokens.

  - Fig. 2.5 shows a scanner that finds all tokens for ac.
  - Fig. 2.6 shows scanning a number token.

  - Each token has the two components:

    1)Token type explains the token's category. (e.g., id)

    2)Token value provides the string value of the token. (e.g., "b")

  - Automatic construction of scanners: Chap.3

Scanning: Divide Input into Tokens

An example ac source program:

```
f b
i a
a = 5
b = a + 3.2
p b
```

Lexems are "words" in the input, for example keywords, operators, identifiers, literals, etc.

Tokens is a datastructure for lexems and additional information

scanner

floatdl	*id*	*intdcl*	*id*	*id*	*assign*	*inum*	
f	b	i	a	a	=	5	…

	assign	*id*	*plus*	*fnum*	*print*	*id*	*eot*
…	=	a	+	3.2	p	b	

```
function SCANNER( ) returns Token
    while s.PEEK( ) = blank do call s.ADVANCE( )
    if s.EOF( )
    then ans.type ← $
    else
        if s.PEEK( ) ∈ { 0, 1, . . . , 9 }
        then ans ← SCANDIGITS( )
        else
            ch ← s.ADVANCE( )
            switch (ch)
                case { a, b, . . . , z } − { i, f, p }
                    ans.type ← id
                    ans.val ← ch
                case f
                    ans.type ← floatdcl
                case i
                    ans.type ← intdcl
                case p
                    ans.type ← print
                case =
                    ans.type ← assign
                case +
                    ans.type ← plus
                case -
                    ans.type ← minus
                case default
                    call LEXICALERROR( )
    return (ans)
end
```

Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	[a − e] | [g − h] | [j − o] | [q − z]
assign	"="
plus	"+"
minus	"-"
inum	$[0-9]^+$
fnum	$[0-9]^+ . [0-9]^+$
blank	$(" \ ")^+$

Figure 2.5: Scanner for the ac language. The variable *s* is an input stream of characters.

```java
/**
 * Figure 2.5 code, processes the input stream looking
 *   for the next Token.
 * @return the next input Token
 */
public static Token Scanner() {
    Token ans;
    while (s.peek() == BLANK)
        s.advance();
    if (s.EOF())
        ans = new Token(EOF);
    else {
        if (isDigit(s.peek()))
            ans = ScanDigits();
        else {
            char ch = s.advance();

            switch(representativeChar(ch)) {
            case 'a':  // matches {a, b, ..., z} - {f, i, p}
                ans = new Token(ID, ""+ch); break;
            case 'f':
                ans = new Token(FLTDCL);  break;
            case 'i':
                ans = new Token(INTDCL);    break;
            case 'p':
                ans = new Token(PRINT);     break;
            case '=':
                ans = new Token(ASSIGN);    break;
            case '+':
                ans = new Token(PLUS);      break;
            case '-':
                ans = new Token(MINUS);     break;
            default:
                throw new Error("Lexical error on character with decimal value: " + (int)ch);

            }
        }

    }
    return ans;
}

/**
```

function ScanDigits( ) **returns** *token*
    tok.val ← " "
    while s.peek( ) ∈ { 0, 1, . . . , 9 } **do**
        tok.val ← *tok.val* + s.advance( )
    if s.peek( ) ≠ "."
    then *tok.type* ← inum
    else
        tok.type ← fnum
        tok.val ← *tok.val* + s.advance( )
        while s.peek( ) ∈ { 0, 1, . . . , 9 } **do**
            tok.val ← *tok.val* + s.advance( )
    return (*tok*)
end

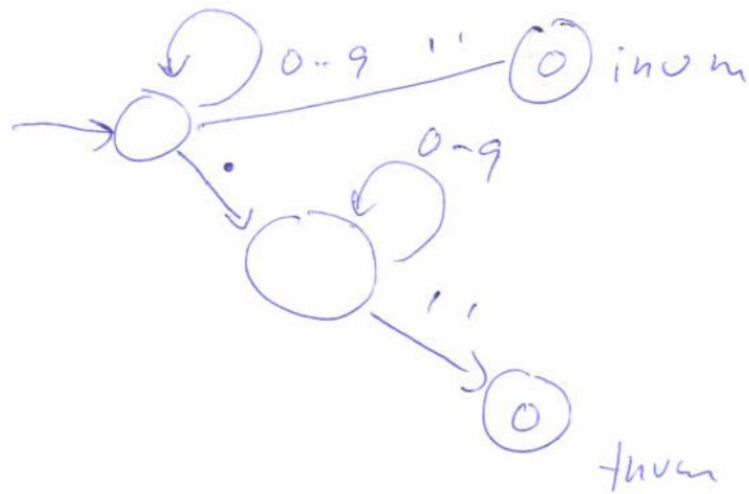


Figure 2.6: Finding inum or fnum tokens for the ac language.

```java
/**
 * Figure 2.6 code, processes the input stream to form
 *     a float or int constant.
 * @return the Token representing the discovered constant
 */

private static Token ScanDigits() {
    String val = "";
    int    type;
    while (isDigit(s.peek())) {
        val = val + s.advance();
    }
    if (s.peek() != '.')
        type = INUM;
    else {
        type = FNUM;
        val = val + s.advance();
        while (isDigit(s.peek())) {
            val = val + s.advance();
        }
    }
    return new Token(type, val);
}
```

Pause

Parsing

- To determine if the stream of tokens conforms to the language's grammar specification
  - Chap. 4, 5, 6
  - For ac, a simple parsing technique called *recursive descent* is used
    - "Mutually recursive parsing routines that descend through a derivation tree"
    - Each nonterminal has an associated parsing procedure for determining if the token stream contains a sequence of tokens derivable from that nonterminal
    - Examine the next input token to predict which production should be applied, e.g:
      - » Stmt → id assign Val Expr
      - » Stmt → print id
    - Predict set
      - » {id} [1]
      - » {print} [6]

```
procedure STMT ( )                                   Stmt → id assign Val Expr
    if ts.PEEK( ) = id                                                        (1)
    then
        call MATCH (ts, id )                                                  (2)
        call MATCH (ts, assign )                                              (3)
        call VAL ( )                                                          (4)
        call EXPR( )                                                          (5)
    else
        if ts.PEEK( ) = print                        Stmt → print id          (6)
        then
            call MATCH (ts, print )
            call MATCH (ts, id )
        else
            call ERROR( )                                                     (7)
end
```

Figure 2.7: Recursive-descent parsing procedure for Stmt. The variable *ts* is an input stream of tokens.

- Consider the productions for Stmts
  - Stmts → Stmt Stmts
  - Stmts → λ
- The predict sets
  - {id, print} [8]
  - {$} [11]

```
procedure STMTS( )
    if ts.PEEK( ) = id or ts.PEEK( ) = print                     (8)
    then
        call STMT( )                                             (9)
        call STMTS( )                                            (10)
    else
        if ts.PEEK( ) = $                                        (11)
        then
            /*   do nothing for λ-production                 */  (12)
        else   call ERROR( )
end
```

Figure 2.8: Recursive-descent parsing procedure for Stmts.

```java
/**
 * Figure 2.7 code
 */
public void Stmts() {
    if (ts.peek() == ID || ts.peek() == PRINT) {
        Stmt();
        Stmts();
    }
    else if (ts.peek() == EOF) {
        // Do nothing for lambda-production
    }
    else error("expected id, print, or eof");

}

public void Stmt() {
    if (ts.peek() == ID) {
        expect(ID);
        expect(ASSIGN);
        Val();
        Expr();
    }
    else if (ts.peek() == PRINT) {
        expect(PRINT);
        expect(ID);
    }
    else error("expected id or print");

}
```

The result of parsing

- If all of the tokens are processed, an **abstract syntax tree (AST)** will be generated**.**
  - An example is shown in fig 2.9.
  - Actually the AST is produced during the process

- **AST** serves as a representation of a program for all phases
  after **syntax analysis**.

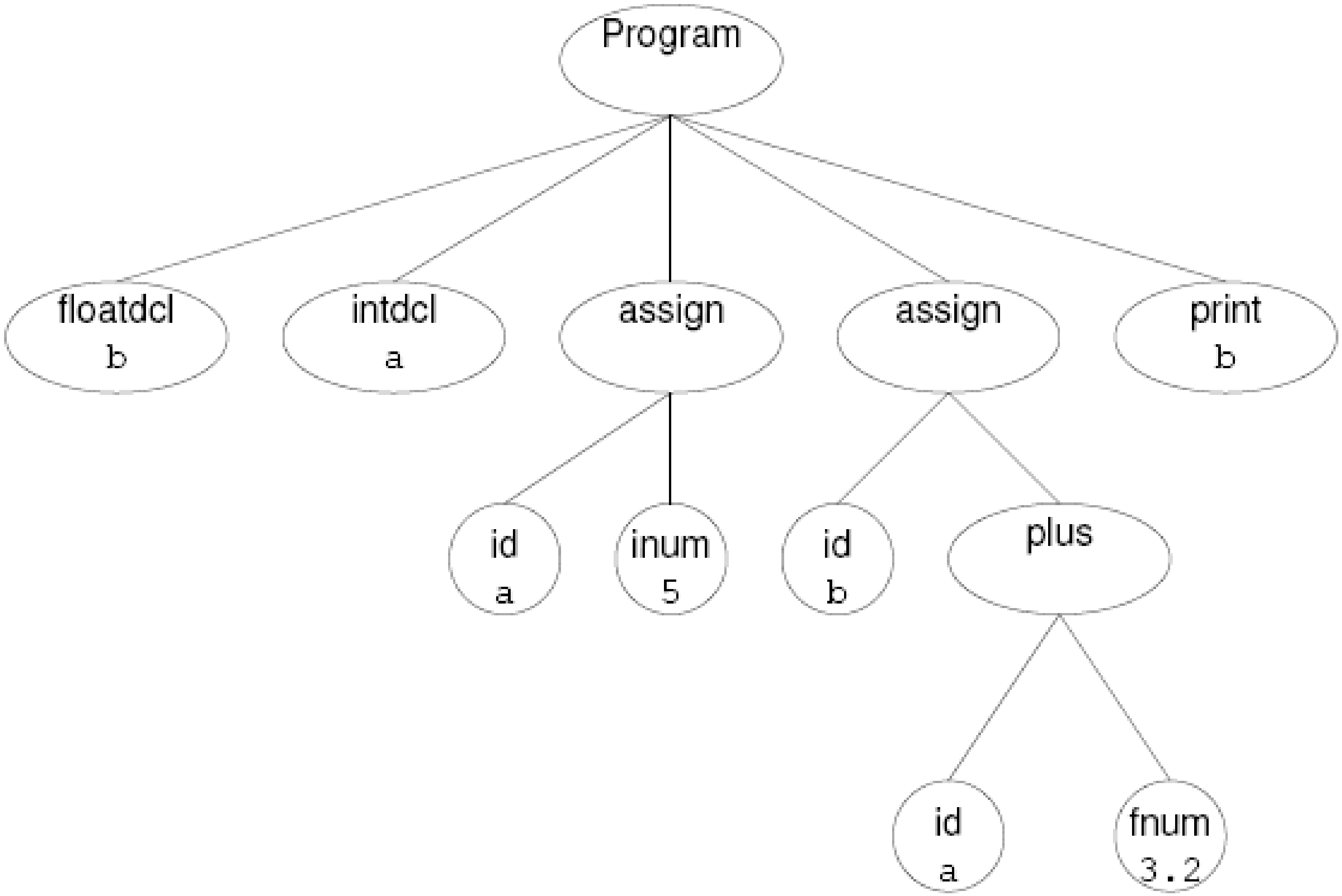


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Abstract Syntax Trees

- Parse trees are large and unnecessarily detailed (Fig. 2.4)
  - Abstract syntax tree (AST) (Fig. 2.9)
    - Inessential punctuation and delimiters are not included
  - A common intermediate representation for all phases after syntax analysis
    - Declarations need not be in source form
    - Order of executable statements explicitly represented
    - Assignment statement must retain identifier and expression
    - Nodes representing computation: operation and operands
    - Print statement must retain name of identifier

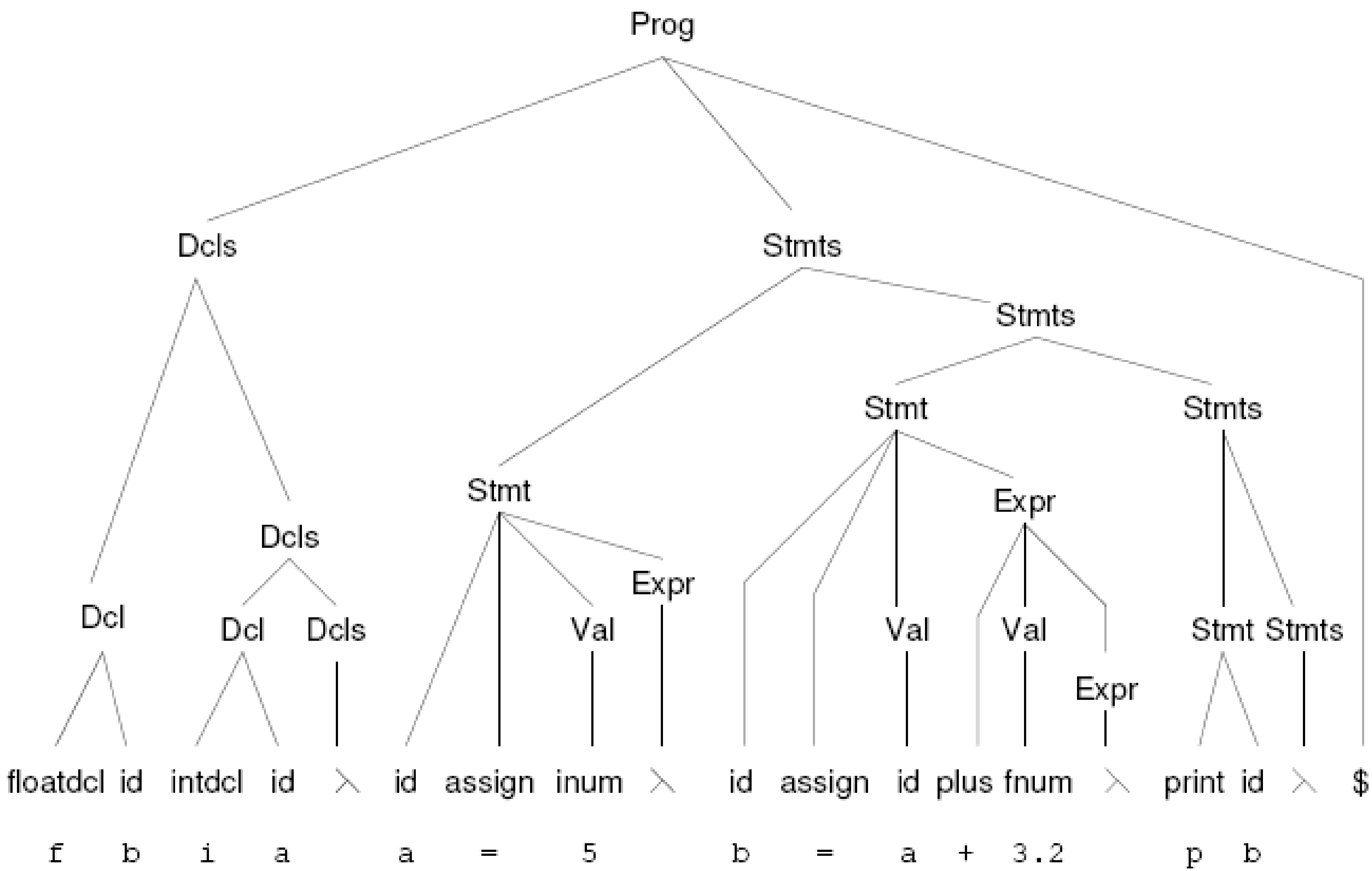


Figure 2.4: An ac program and its parse tree.

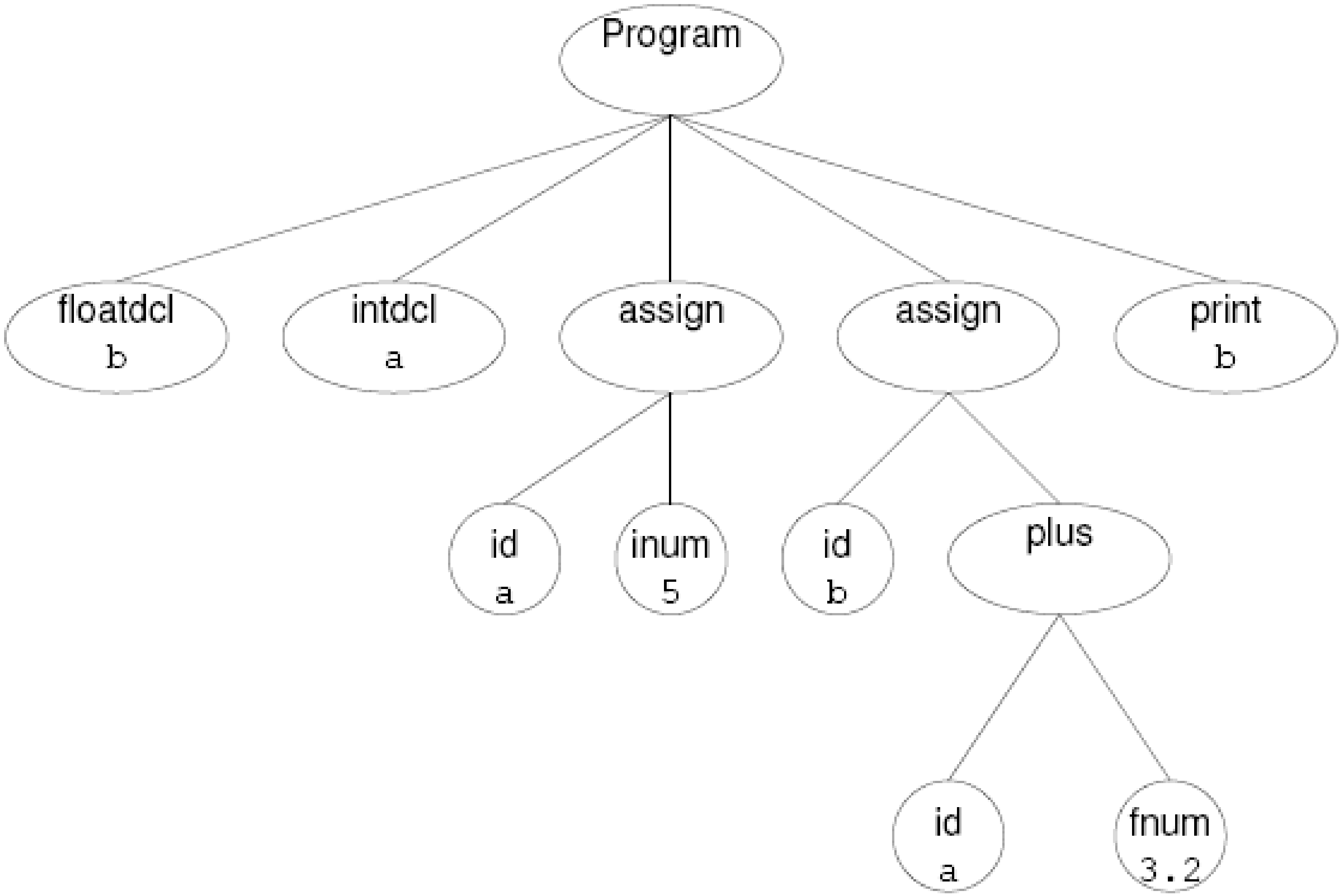


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Contextual Analysis

- Aspects of compilation that can be difficult to perform during syntax analysis
  - Some aspects of language cannot be specified in a CFG
    - Symbol usage consistency with type declaration
    - Scope/visibility of variables
    - In Java: x.y.z
      - Package x, class y, static field z
      - Variable x, field y, another field z
    - Operator overloading
      - +: numerical addition or appending of strings
  - Separation into phases makes the compiler much easier to write and maintain

Semantic Analysis

- Example processing
  - Declarations and name scopes are processed to construct a symbol table
  - Type consistency
  - Make type-dependent behavior explicit

Symbol Tables

- To record all identifiers and their types
  - 23 entries for 23 distinct identifiers in ac (Fig. 2.11)
    - Type info.: integer, float, unused (null)
    - Attributes: scope, storage class, protection properties
  - Symbol table construction (Fig. 2.10)
    - Symbol declaration nodes call VISIT(SymDeclaring n)
    - ENTERSYMBOL checks the given symbol has not been previously declared

```
/*  Visitor methods                                                    */
procedure VISIT( SymDeclaring n )
    if n.GETTYPE( ) = floatdcl
    then  call ENTERSYMBOL(n.GETID( ), float)
    else  call ENTERSYMBOL(n.GETID( ), integer)
end

/*  Symbol table management                                            */
procedure ENTERSYMBOL(name, type)
    if SymbolTable[name] = null
    then  SymbolTable[name] ← type
    else  call ERROR( "duplicate declaration" )
end

function LOOKUPSYMBOL(name) returns type
    return (SymbolTable[name])
end
```

Figure 2.10: Symbol table construction for ac.

Symbol	Type	Symbol	Type	Symbol	Type
a	integer	k	null	t	null
b	float	l	null	u	null
c	null	m	null	v	null
d	null	n	null	w	null
e	null	o	null	x	null
g	null	q	null	y	null
h	null	r	null	z	null
j	null	s	null		

Figure 2.11: Symbol table for the ac program from Figure 2.4.

Type Checking

- Only two types in ac
  - Integer
  - Float
- Type hierarchy
  - Float wider than integer
  - Automatic widening (or casting)
    - integer -> float
- All identifiers must be type-declared in a program before they can be used
- This process walks the AST bottom-up from its leaves toward its root.

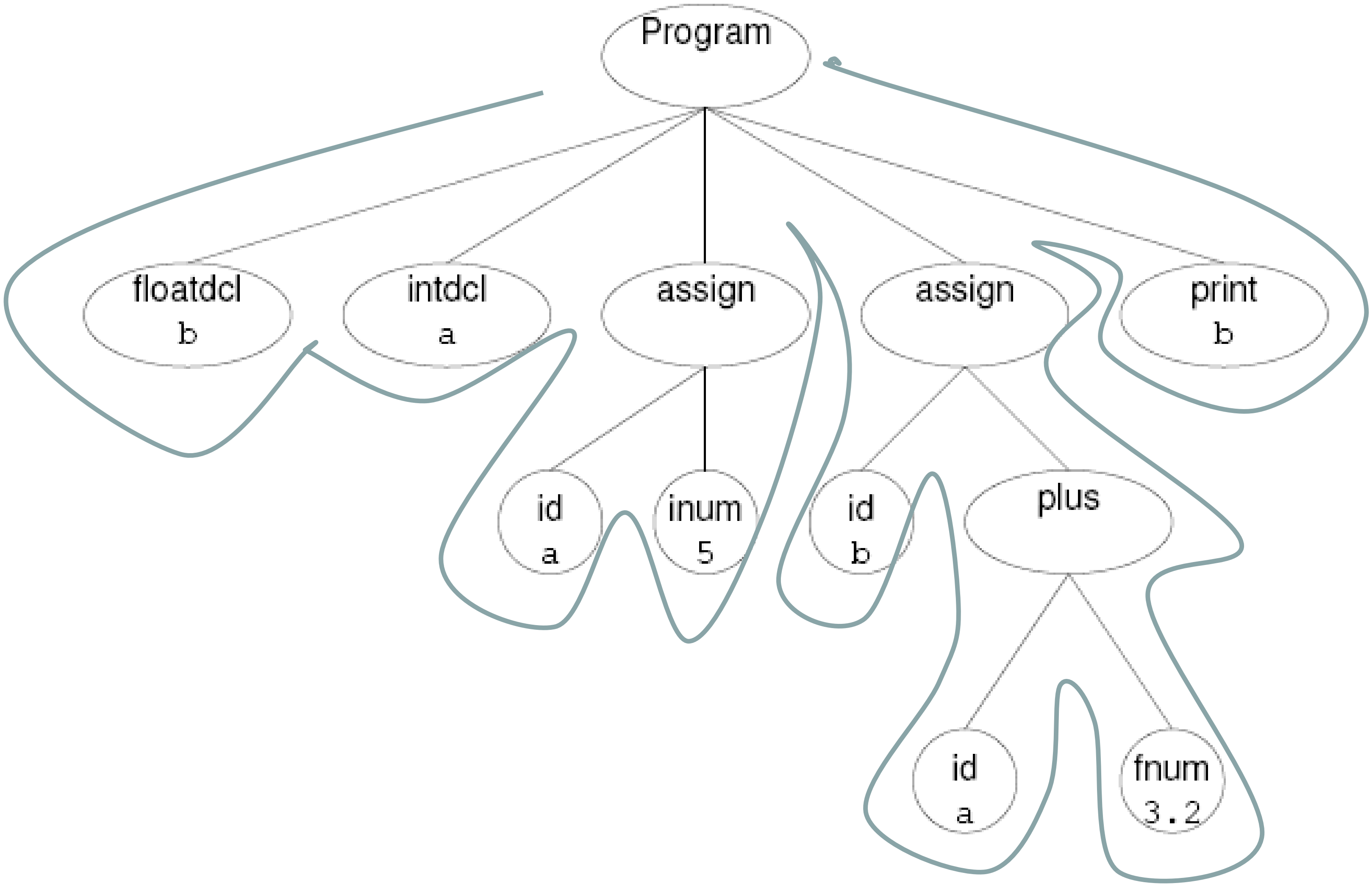


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Phases of an ac compiler (Cont.)

- At each node, appropriate analysis is applied:
  - For constants and symbol references, the visitor methods simple set the supplied node's type based on the node's contents.
  - For nodes that compute value, such as **plus** and **minus**, the appropriate type is computed by calling the utility methods.
  - For an assignment operation, the visitor makes certain that the value computed by the second child is of the same type as the assigned identifier (the first child).

The results of applying semantic analysis to the AST of fig 2.9 are shown in fig 2.13.

Type Checking

```
/*   Visitor methods                                              */
procedure VISIT( Computing n )
    n.type ← CONSISTENT(n.child1, n.child2 )
end
procedure VISIT( Assigning n )
    n.type ← CONVERT(n.child2, n.child1.type )
end
procedure VISIT( SymReferencing n )
    n.type ← LOOKUPSYMBOL(n.id )
end
procedure VISIT( IntConsting n )
    n.type ← integer
end
procedure VISIT( FloatConsting n )
    n.type ← float
end
```

```java
package acASTVisitor;

public class TypeChecker extends Visitor {

    @Override
    void visit(Assigning n) {
        // TODO Auto-generated method stub
        n.child1.accept(this);
        int m = AST.SymbolTable.get(n.id);
        int t = generalize(n.child1.type,m);
        n.child1 = convert(n.child1,m);
        n.type = t;
    }

    @Override
    void visit(Computing n) {
        // TODO Auto-generated method stub
        n.child1.accept(this);
        n.child2.accept(this);
        int m = generalize(n.child1.type,n.child2.type);
        n.child1 = convert(n.child1,m);
        n.child2 = convert(n.child2,m);
        n.type = m;
    }

    void visit(ConvertingToFloat n){
        n.child.accept(this);
        n.type = AST.FLTTYPE;
    }

    @Override
    void visit(FloatConsting n) {
        // TODO Auto-generated method stub
        n.type = AST.FLTTYPE;

    }

    @Override
    void visit(IntConsting n) {
        // TODO Auto-generated method stub
        n.type = AST.INTTYPE;

    }
```

```
/*   Type-checking utilities                                          */
function CONSISTENT( c1, c2 ) returns type
    m ← GENERALIZE( c1.type, c2.type )
    call CONVERT( c1, m )
    call CONVERT( c2, m )
    return (m)
end
function GENERALIZE( t1, t2 ) returns type
    if t1 = float or t2 = float
    then  ans ← float
    else  ans ← integer
    return (ans)
end
procedure CONVERT( n, t )
    if n.type = float and t = integer
    then  call ERROR( "Illegal type conversion" )
    else
        if n.type = integer and t = float
        then
            /*   replace node n by convert-to-float of node n     */ (13)
        else   /* nothing needed */
end
```

```java

}*/

private int generalize(int t1, int t2){
    if (t1 == AST.FLTTYPE || t2 == AST.FLTTYPE) return AST.FLTTYPE; else return AST.INTTYPE;
}

private AST convert(AST n, int t){
    if (n.type == AST.FLTTYPE && t == AST.INTTYPE) error("Illegal type conversion");
    else if (n.type == AST.INTTYPE && t == AST.FLTTYPE) return new ConvertingToFloat(n);
    return n;
}
```

- Type checking
  - Constants and symbol reference: simply set the node’s type based on the node’s contents
  - Computation nodes: CONSISTENT(n.c1, n.c2)
  - Assignment operation: CONVERT(n.c2, n.c1.type)
- CONSISTENT()
  - GENERALIZE(): determines the least general type
  - CONVERT(): checks whether conversion is necessary

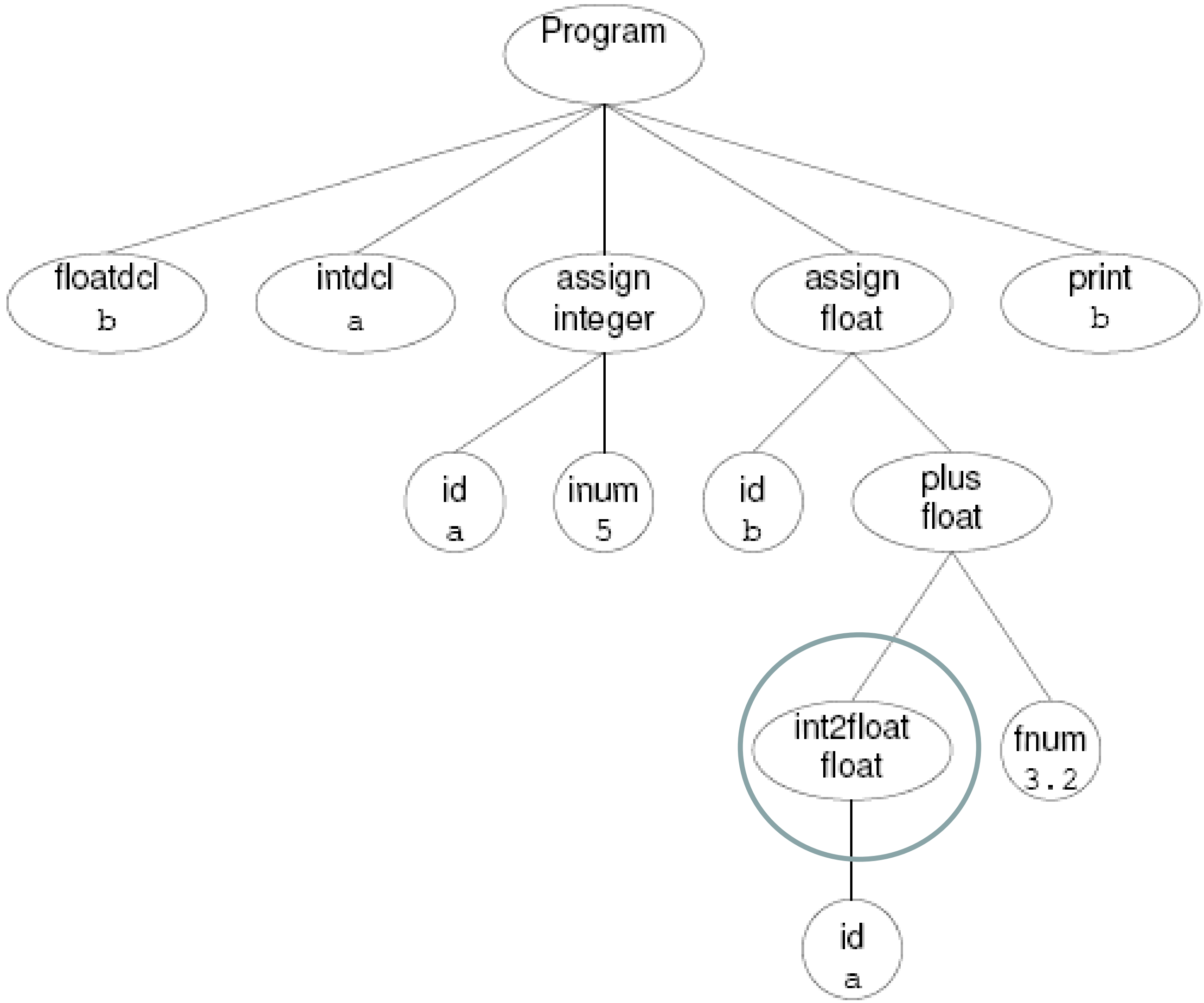


Figure 2.13: AST after semantic analysis.

Code Generation

- The formulation of target-machine instructions that faithfully represent the semantics of the source program
  - Chap. 11 & 13
  - dc: stack machine model
  - Code generation proceeds by traversing the AST, starting at its root
    - VISIT (Computing n)
    - VISIT (Assigning n)
    - VISIT (SymReferencing n)
    - VISIT (Printing n)
    - VISIT (Converting n)

```
procedure visit( Assigning n )
    call CodeGen( n.child2 )
    call Emit( "s" )
    call Emit( n.child1.id )
    call Emit( "0 k" )                    (14)
end
procedure visit( Computing n )
    call CodeGen( n.child1 )
    call CodeGen( n.child2 )
    call Emit( n.operation )              (15)
end
procedure visit( SymReferencing n )
    call Emit( "l" )
    call Emit( n.id )
end
procedure visit( Printing n )
    call Emit( "l" )
    call Emit( n.id )
    call Emit( "p" )
    call Emit( "si" )                     (16)
end
procedure visit( Converting n )
    call CodeGen( n.child )
    call Emit( "5 k" )                    (17)
end
procedure visit( Consting n )
    call Emit( n.val )
end
```

Figure 2.14: Code generation for ac

```java
package acASTVisitor;

public class CodeGenerator extends Visitor {

    String code = "";

    public void emit(String c){
        code = code + c;
    }

    @Override
    void visit(Assigning n) {
        n.child1.accept(this);
        emit(" s");
        emit(n.id);
        emit(" 0 k ");
    }

    @Override
    void visit(Computing n) {
        n.child1.accept(this);
        n.child2.accept(this);
        emit(n.operation);
    }

    void visit(ConvertingToFloat n){
        n.child.accept(this);
        emit(" 5 k ");
    }

    @Override
    void visit(FloatConsting n) {
        emit(" " + n.val + " ");
    }

    @Override
```

```java
    @Override
    void visit(Printing n) {
        emit("l");
        emit(n.id);
        emit(" p ");
        emit("si ");
    }

    @Override
    void visit(Prog n) {
        for(AST ast : n.prog){
            ast.accept(this);
        };
        System.out.println(code);
    }

    @Override
    void visit(SymDeclaring n) {
    }

    @Override
    void visit(FloatDcl n) {
    }

    @Override
    void visit(IntDcl n) {
    }

    @Override
    void visit(SymReferencing n) {
        emit("l");
        emit(n.id + " ");
    }


}
```

Code	Source	Comments
5	a = 5	Push 5 on stack
sa		Pop the stack, storing (s) the popped value in register a
0 k		Reset precision to integer
la	b = a + 3.2	Load (l) register a, pushing its value on stack
5 k		Set precision to float
3.2		Push 3.2 on stack
+		Add: 5 and 3.2 are popped from the stack and their sum is pushed
sb		Pop the stack, storing the result in register b
0 k		Reset precision to integer
lb	p b	Push the value of the b register
p		Print the top-of-stack value
si		Pop the stack by storing into the i register

Figure 2.15: Code generated for the AST shown in Figure 2.9.

- That’s it !!
- At least for ac on dc

Some advice

- A language design and compiler project follows an iterative approach
- but each iteration is easy to structure:
  - Design phase (Lecture 1-5 + 13-14 + 19)
  - Front-end development (Lecture 6-9)
  - Contextual analysis (Lecture 10-12)
  - Code generation or interpretation (Lecture 15-18 + 20)
  - If not happy start again
- You will learn the techniques and tools you need in time for you to apply them in your project

Choosing the impl. language

Q: Which programming languages play a role in this picture?

input → Translator → output

source program — is expressed in the *source language*

Translator — is expressed in the *implementation language*

object program — is expressed in the *target language*

A: All of them!

What can we do now in our projects?

- Write programs!
- Imagine that you have already designed your language – how would programs look?
- Serves as outset for discussions about your language design
  - Especially token and grammer design
- Write lots of programs – they will serve as test case for your compiler later
- Start thinking about implementation language

Languages and Compilers
(SProg og Oversættere)

Lecture 4
Language specifications

Bent Thomsen

Department of Computer Science

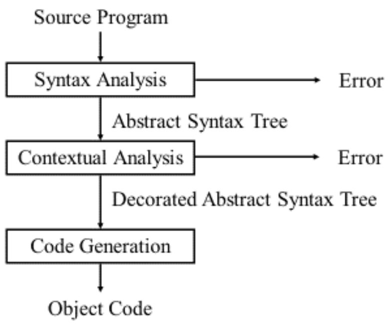
Aalborg University

Learning goals

- A deeper understanding of programming language specifications
- Introduction to context free grammars
- Introduction to BNF and EBNF
- Overview of formal specifications notations

Programming Language Specification

- Why?
  - A communication device between people who need to have a common understanding of the PL:
    - language designer, language implementor, language user
- What to specify?
  - Specify what is a 'well formed' program
    - syntax
    - contextual constraints (also called static semantics):
      - scope rules
      - type rules
  - Specify what is the meaning of (well formed) programs
    - semantics (also called runtime semantics)



Programming Language Specification

- Why?
- What to specify?
- How to specify ?
  - Formal specification: use some kind of precisely defined formalism
  - Informal specification: description in English.

  - Usually a mix of both (e.g. Java specification)
    - Syntax => formal specification using RE and CFG
    - Contextual constraints and semantics => informal
    - Formal semantics has been retrofitted though
  - But trend towards more formality (C#, Fortress)
    - fortress.pdf
    - Ecma-334.pdf

Fortress definition p. 71 and p. 181

13.4 Dotted Method Invocations

Syntax:

Primary	::=	*Primary . Id StaticArgs? ParenthesisDelimited*
ParenthesisDelimited	::=	*Parenthesized*
	|	*ArgExpr*
	|	( )
Parenthesized	::=	(*Expr*)
ArgExpr	::=	*TupleExpr*
	|	((*Expr* ,)* *Expr* ...)
TupleExpr	::=	((*Expr* ,)+ *Expr*)

A *dotted method invocation* consists of a subexpression (called the receiver expression), followed by '.', followed by an identifier, an optional list of static arguments (described in Chapter 9) and a subexpression (called the *argument expression*). Unlike in function calls (described in Section 13.6), the argument expression must be parenthesized, even if it is not a tuple. There must be no whitespace on the left-hand side of the '.' and the left-hand side of the left parenthesis of the argument expression. The receiver expression evaluates to the receiver of the invocation (bound to the self parameter (discussed in Section 10.2) of the method). A method invocation may include explicit instantiations of static parameters but most method invocations do not include them.

The receiver and arguments of a method invocation are each evaluated in parallel in a separate implicit thread (see Section 5.4). After this thread group completes normally, the body of the method is evaluated with the parameter of the method bound to the value of the argument expression (thus evaluation of the body occurs after evaluation of the receiver and arguments in dynamic program order). The value and the type of a dotted method invocation are the value and the type of the method body.

We say that methods or functions (collectively called as *functionals*) may be *applied to* (also "*invoked on*" or "*called with*") an argument. We use "call", "invocation", and "application" interchangeably.

$$[\text{R-METHOD}] \quad \frac{\texttt{object}\ O\ \_\ (\overrightarrow{x{:}\_})\ \_\ \texttt{end} \in p \qquad mbody_p(f[\![\overrightarrow{\tau'}]\!], O[\![\overrightarrow{\tau}]\!]) = \{(\overrightarrow{x'}) \to e\}}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).f[\![\overrightarrow{\tau'}]\!](\overrightarrow{v'})] \longrightarrow E[[\overrightarrow{v}/\overrightarrow{x}][O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x'}]e]}$$

The C89 standard – 519 pages

6.8 Statements and blocks

Syntax

1 *statement:*
  labeled-statement
  compound-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement

Semantics

2 A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence.

3 A *block* allows a set of declarations and statements to be grouped into one syntactic unit. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.

4 A *full expression* is an expression that is not part of another expression or of a declarator. Each of the following is a full expression: an initializer; the expression in an expression statement; the controlling expression of a selection statement (`if` or `switch`); the controlling expression of a `while` or `do` statement; each of the (optional) expressions of a `for` statement; the (optional) expression in a `return` statement. The end of a full expression is a sequence point.

Forward references: expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the `return` statement (6.8.6.4).

6.8.3 Expression and null statements

Syntax

1 *expression-statement:*
 $expression_{opt}$ **;**

Semantics

2 The expression in an expression statement is evaluated as a void expression for its side effects.[134)]

3 A *null statement* (consisting of just a semicolon) performs no operations.

4 EXAMPLE 1 If a function call is evaluated as an expression statement for its side effects only, the discarding of its value may be made explicit by converting the expression to a void expression by means of a cast:

```
int p(int);
/* ... */
(void)p(0);
```

134) Such as assignments, and function calls which have side effects.

5 EXAMPLE 2 In the program fragment

```
char *s;
/* ... */
while (*s++ != '\0')
        ;
```

a null statement is used to supply an empty loop body to the iteration statement.

6 EXAMPLE 3 A null statement may also be used to carry a label just before the closing } of a compound statement.

```
while (loop1) {
        /* ... */
        while (loop2) {
                /* ... */
                if (want_out)
                        goto end_loop1;
                /* ... */
        }
        /* ... */
end_loop1: ;
}
```

Forward references: iteration statements (6.8.5).

Programming Language Specification

A language specification need to address:

- Syntax
  - Token grammar: Regular Expressions
  - Context Free Grammar: BNF or EBNF
- Contextual constraints
  - Scope rules (static semantics)
    - Often informal, but can be formalized
  - Type rules (static semantics)
    - Informal or Formal
- Semantics (dynamic semantics)
  - Informal or Formal

Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

The General Problem of Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`)
- A *token* is a category of lexemes (e.g., identifier)

Definition of Tokens/lexemes

- Tokens are often specified using regular expressions
- Remember:

Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	[a − e] | [g − h] | [j − o] | [q − z]
assign	"="
plus	"+"
minus	"-"
inum	$[0-9]^+$
fnum	$[0-9]^+ . [0-9]^+$
blank	$(" \; ")^+$

Figure 2.3: Formal definition of ac tokens.

Note: In most languages id is a sequence of letters and numbers starting
With a letter defined as [a-z]([a-z]|[0-9])*

Formal Definition of Languages

- **Generators**
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator
- **Recognizers**
  - A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler

BNF and Context-Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages

- Backus-Naur Form (1959)
  - Invented by John Backus to describe Algol 58
  - Modified by Peter Naur to describe Algol 60
  - BNF is equivalent to context-free grammars

Syntax Specification

Syntax is specified using “Context Free Grammars”:

- A finite set of **terminal symbols** (or tokens)
- A finite set of **non-terminal symbols**
- A **start symbol**
- A finite set of **production rules**

A CFG defines a set of strings

- This is called the language of the CFG.

Backus-Naur Form

Usually CFG are written in BNF notation.

A production rule in BNF notation is written as:

$N ::= \alpha$ where N is a non terminal
and α a sequence of terminals and non-terminals

$N ::= \alpha \mid \beta \mid \ldots$ is an abbreviation for several rules with N
as left-hand side.

Sometimes non terminals are represented in angel brackets: <N> and ::= is replaced with →

Syntax Specification

Example:

```
Start ::= Letter
        | Start Letter
        | Start Digit
Letter ::= a | b | c | d | ... | z
Digit  ::= 0 | 1 | 2 | ... | 9
```

Q: What is the "language" defined by this grammar?

Note: a sequence of letters and numbers starting with a letter defined in RE as
[a-z]([a-z] | [0-9])*

What is the "language" defined by this grammar?

identifier::= available-identifier
| @ *identifier-or-keyword*
available-identifier::= identifier-or-keyword (that is not a *keyword)*
identifier-or-keyword::= identifier-start-character identifier-part-characters$_{opt}$
identifier-start-character::= letter-character
| _ (the underscore character U+005F)
identifier-part-characters::= identifier-part-character
| *identifier-part-characters identifier-part-character*
identifier-part-character::= letter-character
| *decimal-digit-character*
| *connecting-character*
| *combining-character*
| *formatting-character*
letter-character::= A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
| A *unicode-escape-sequence* representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl
combining-character::= A Unicode character of classes Mn or Mc
| A *unicode-escape-sequence* representing a character of classes Mn or Mc
decimal-digit-character::= A Unicode character of the class Nd
| A *unicode-escape-sequence* representing a character of the class Nd
connecting-character::= A Unicode character of the class Pc
| A *unicode-escape-sequence* representing a character of the class Pc
formatting-character::= A Unicode character of the class Cf
| A *unicode-escape-sequence* representing a character of the class Cf

http://msdn.microsoft.com/en-us/library/aa664812(VS.71).aspx

What is the “language” defined by this grammar?

3.8. Identifiers

An *identifier* is an unlimited-length sequence of *Java letters* and *Java digits*, the first of which must be a *Java letter*.

```
Identifier:
  IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral

IdentifierChars:
  JavaLetter {JavaLetterOrDigit}

JavaLetter:
  any Unicode character that is a "Java letter"

JavaLetterOrDigit:
  any Unicode character that is a "Java letter-or-digit"
```

A "Java letter" is a character for which the method `Character.isJavaIdentifierStart(int)` returns true.

A "Java letter-or-digit" is a character for which the method `Character.isJavaIdentifierPart(int)` returns true.

The "Java letters" include uppercase and lowercase ASCII Latin letters `A-Z` (`\u0041-\u005a`), and `a-z` (`\u0061-\u007a`), and, for historical reasons, the ASCII underscore (`_`, or `\u005f`) and dollar sign (`$`, or `\u0024`). The `$` sign should be used only in mechanically generated source code or, rarely, to access pre-existing names on legacy systems.

The "Java digits" include the ASCII digits `0-9` (`\u0030-\u0039`).

Letters and digits may be drawn from the entire Unicode character set, which supports most writing scripts in use in the world today, including the large sets for Chinese, Japanese, and Korean. This allows programmers to use identifiers in their programs that are written in their native languages.

An identifier cannot have the same spelling (Unicode character sequence) as a keyword (§3.9), boolean literal (§3.10.3), or the null literal (§3.10.7), or a compile-time error occurs.

Two identifiers are the same only if they are identical, that is, have the same Unicode character for each letter or digit. Identifiers that have the same external appearance may yet be different.

Spot the syntax error

```
{
  x = 1;
  y = 2;
  z = 1+2
}
```

Syntax Specification

Subtle example 1:

```
Block ::= { Statements }
Statements ::= Statement ; Statements
             |  Statement

Statement  ::= V-name = Expression
             | Identifier ( Expression )
             | …
```

Syntax Specification

Subtle example 2:

```
Block ::= { Statements }
Statements ::= Statement Statements
             |  Statement

Statement  ::= V-name = Expression ;
             | Identifier ( Expression ) ;
             | …
```

Syntax Specification

Subtle example 3:

```
Block ::= { Statements }
Statements ::= Statement ; Statements
             |  Statement ;

Statement  ::= V-name = Expression
             | Identifier ( Expression )
             | …
```

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Syntax Specification

Subtle example 4:

```
Block ::= begin Statements end
Statements ::= Statement ; Statements
             |  Statement ;

Statement  ::= V-name = Expression
             | Identifier ( Expression )
             | ...
```

Syntax Specification

Bad example 4:

```
Block ::= \nl Statements \nl
Statements ::= Statement \nl Statements
             |  Statement \nl

Statement  ::= V-name = Expression
             | Identifier ( Expression )
             | ...
```

BNF Fundamentals

- In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols,* or just *nonterminals*)
- *Terminals* are lexemes or tokens
- A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals
- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

- Grammar: a finite non-empty set of rules
- A *start symbol* is a special element of the nonterminals of a grammar

Note: terminals/lexemes like **if** and **then** are often used in CFG instead of tokens **if_token** and **then_token**

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
<stmt> → begin <stmt_list> end
```

- Alternative rules are written with |

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

Describing Lists

- Syntactic lists are described using recursion

```
<ident_list> → ident
             | ident, <ident_list>
```

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

Pause

An Example Grammar

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

An Example Derivation

```
<program> => <stmts> => <stmt>
                     => <var> = <expr>
                     => a = <expr>
                     => a = <term> + <term>
                     => a = <var> + <term>
                     => a = b + <term>
                     => a = b + const
```

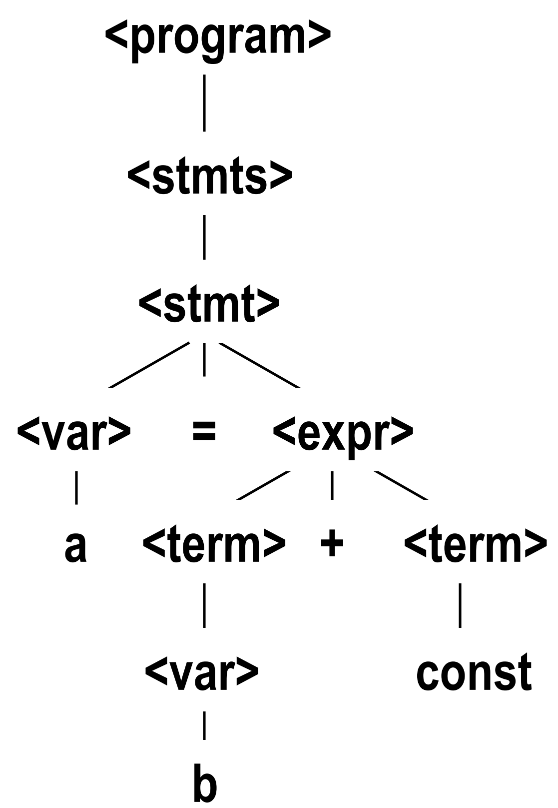
```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

Derivations

- Every string of symbols in a derivation is a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols
- A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A rightmost derivation is one in which the rightmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Parse Tree

- A hierarchical representation of a derivation



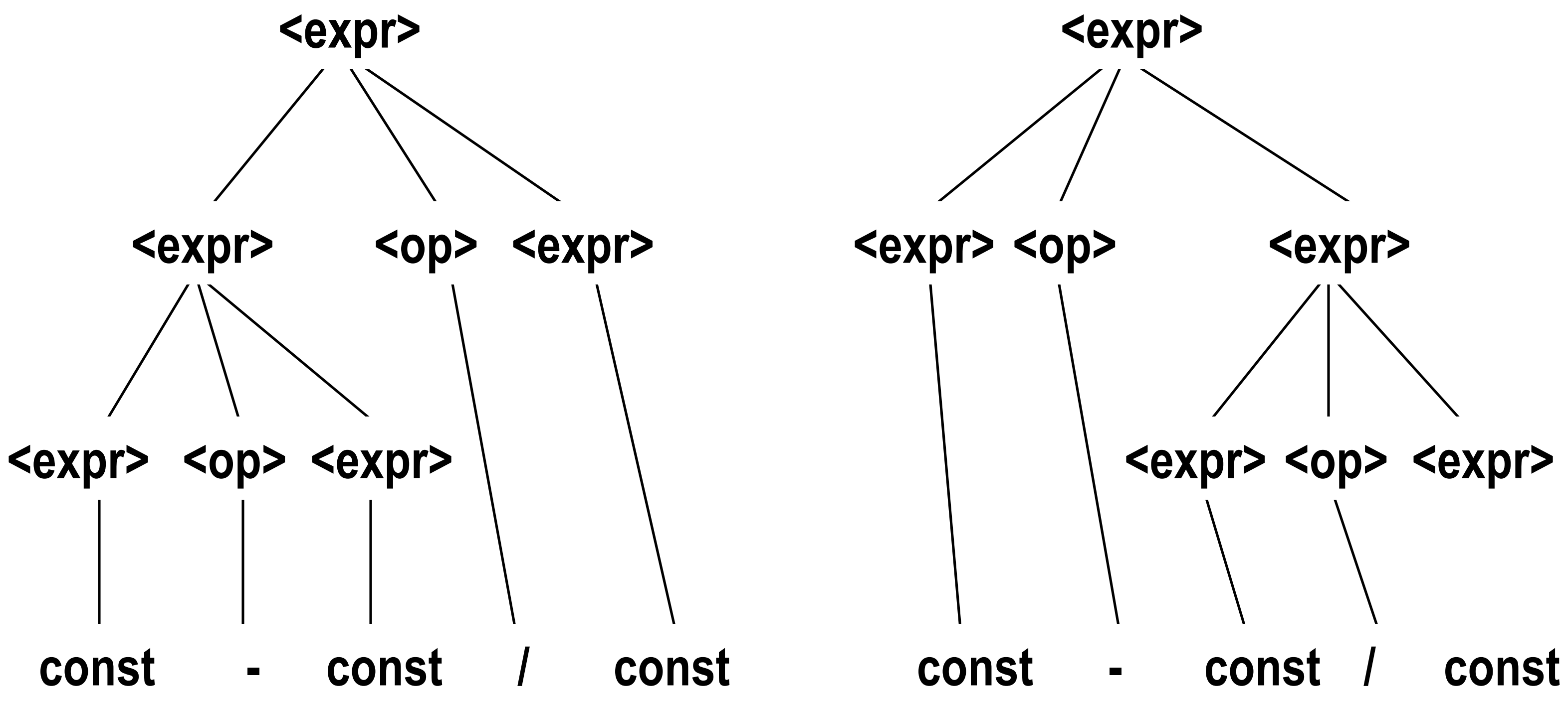
Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

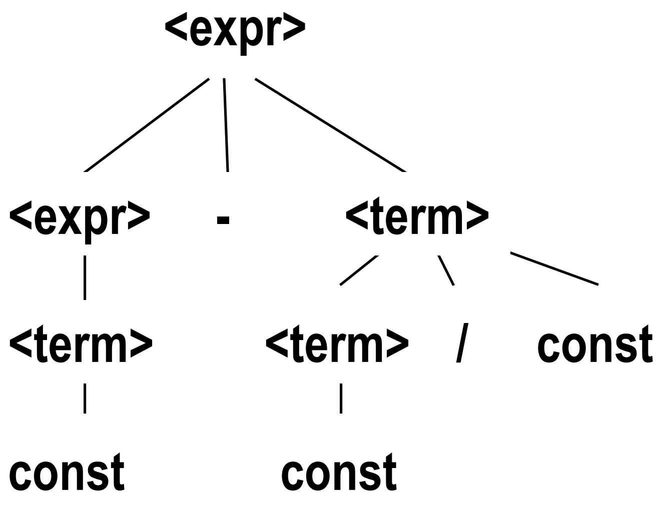
<op> → / | -



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

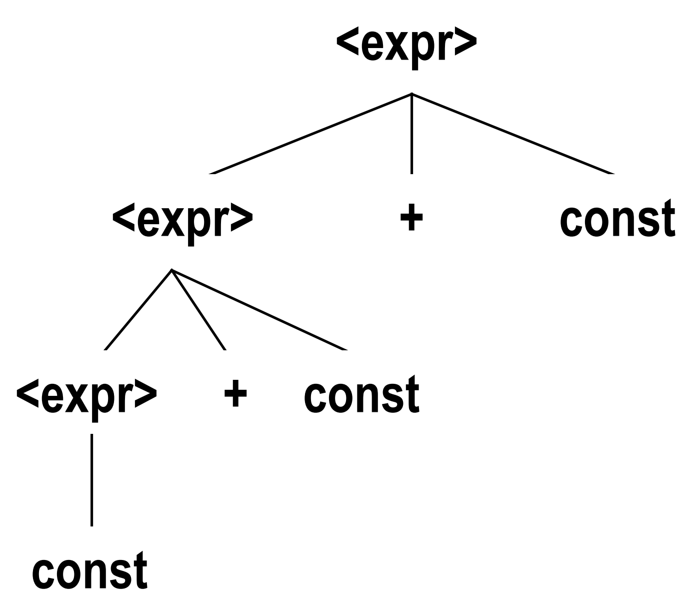
```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Extended BNF

- Optional parts are placed in brackets [ ]

```
<proc_call> -> ident [(<expr_list>)]
```

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

```
<term> → <term> (+|-) const
```

- Repetitions (0 or more) are placed inside braces { }

```
<ident> → letter {letter|digit}
```

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
             | <expr> - <term>
             | <term>
<term> → <term> * <factor>
             | <term> / <factor>
             | <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

Recent Variations in EBNF

- Alternative RHSs are put on separate lines
- Use of a colon or = or := instead of →
- Use of $_{\texttt{opt}}$ for optional parts
- Use of `oneof` for choices
- Sometimes terminal (lexems or tokens) are written in “ “ or ` ` or in **bold** or color ..
- Sometimes given in a seperate grammar and the non-terminals from this grammer is used as terminal in the CFG
- Sometimes ( )* is used for { } and ? for [ ]

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF

```
<expr> → <term> ((+ | -) <term>)*
<term> → <factor> ((* | /) <factor>)*
```

EBNF in EBNF

```
Production    = production_name "=" [ Expression ] "." .
Expression    = Alternative { "|" Alternative } .
Alternative   = Term { Term } .
Term          = production_name | token [ "…" token ]
              | Group | Option | Repetition .
Group         = "(" Expression ")" .
Option        = "[" Expression "]" .
Repetition    = "{" Expression "}" .
```

An Example Language Specification

Mini Triangle is a very simple Pascal-like language introduced in Brown & Watt's book: Language Processors in Java

An example program:

```
!This is a comment.
let const m ~ 7;
    var n
in
   begin
     n := 2 * m * m ;
     putint(n)
   end
```

Declarations: `const m ~ 7; var n`

Expression: `2 * m * m`

Command: `n := 2 * m * m`

Syntax of Mini Triangle

```
Program ::= single-Command
single-Command
      ::= V-name := Expression
        | Identifier ( Expression )
        | if Expression then single-Command
                        else single-Command
        | while Expression do single-Command
        | let Declaration in single-Command
        | begin Command end
Command ::= single-Command
          | Command ; single-Command
```

Syntax of Mini Triangle (continued)

```
Expression
   ::= primary-Expression
     | Expression Operator primary-Expression
primary-Expression
   ::= Integer-Literal
     | V-name
     | Operator primary-Expression
     | ( Expression )
V-name ::= Identifier
Identifier ::= Letter
             | Identifier Letter
             | Identifier Digit
Integer-Literal ::= Digit
                  | Integer-Literal Digit
Operator ::= + | - | * | / | < | > | =
```

Syntax of Mini Triangle (continued)

```
Declaration
  ::= single-Declaration
    | Declaration ; single-Declaration
single-Declaration
  ::= const Identifier ~ Expression
    | var Identifier := Type-denoter
Type-denoter ::= Identifier
```

```
Comment ::= ! CommentLine eol
CommentLine ::= Graphic CommentLine
Graphic ::= any printable character or space
```

Concrete Syntax of Commands

```
single-Command
      ::= V-name := Expression
        | Identifier ( Expression )
        | if Expression then single-Command
                             else single-Command
        | while Expression do single-Command
        | let Declaration in single-Command
        | begin Command end
Command ::= single-Command
          | Command ; single-Command
```

Abstract Syntax of Commands

```
Command
 ::= V-name := Expression                        AssignCmd
   | Identifier ( Expression )                   CallCmd
   | if Expression then Command
                   else Command                  IfCmd
   | while Expression do Command                 WhileCmd
   | let Declaration in Command                  LetCmd
   | Command ; Command                           SequentialCmd
```

An abstract syntax , like the above, is often used in the definition of the formal semantics

Even more Abstract Syntax of Commands

```
Command
  ::= V-name Expression                      AssignCmd
    | Identifier Expression                  CallCmd
    | Expression Command Command             IfCmd
    | Expression Command                     WhileCmd
    | Declaration Command                    LetCmd
    | Command Command                        SequentialCmd
```

An abstract syntax, like the above, may form the basis for the design of the AST

Contextual Constraints

Syntax rules alone are not enough to specify the format of well-formed programs.

Example 1:

```
let const m~2
in   m + x
```

Undefined! ⟹ Scope Rules

Example 2:

```
let const m~2 ;
    var   n:Boolean
in begin
   n  := m<4;
   n  := n+1
end
```

Type error! ⟹ Type Rules

Scope Rules

Scope rules regulate visibility of identifiers. They relate every **applied occurrence** of an identifier to a **binding occurrence**

Example 1

Binding occurence

```
let const m~2;
    var   r:Integer
in
    r := 10*m
```

Applied occurence

Example 2:

```
let const m~2
in m + x
```

?

Terminology:

Static binding vs. *dynamic binding*

Type Rules

Type rules regulate the expected types of arguments and types of returned values for the operations of a language.

Examples

Type rule of `<` :

`E1 < E2` is type correct and of type `Boolean`
if `E1` and `E2` are type correct and of type `Integer`

Type rule of `while`:

`while E do C` is type correct
if `E` of type `Boolean` and `C` type correct

Terminology:

Static typing vs. *dynamic typing*

Semantics

Specification of semantics is concerned with specifying the "meaning" of well-formed programs.

Terminology:

Expressions are **evaluated** and **yield values** (and may or may not perform side effects)

Commands are **executed** and **perform side effects**.

Declarations are **elaborated** to **produce bindings**

Side effects:

- change the values of variables
- perform input/output

Semantics

Example: The semantics of expressions.

*An **expression** is **evaluated** to yield a **value.***

An (integer literal expression) `IL` *yields the integer value of* `IL`

The (variable or constant name) expression `V` *yields the value of the variable or constant named* `V`

The (binary operation) expression `E1 O E2` *yields the value obtained by applying the binary operation* `O` *to the values yielded by (the evaluation of) expressions* `E1` *and* `E2`

etc.

Semantics

Example: The semantics of declarations.

A ***declaration*** *is* ***elaborated*** *to produce* **bindings.** *It may also have the side effect of allocating (memory for) variables.*

The constant declaration `const I~E` *is elaborated by binding the identifier value* `I` *to the value yielded by* `E`

The constant declaration `var I:T` *is elaborated by binding* `I` *to a newly allocated variable, whose initial value is undefined. The variable will be deallocated on exit from the let containing the declaration.*

The sequential declaration `D1;D2` *is elaborated by elaborating* `D1` *followed by* `D2` *combining the bindings produced by both.* `D2` *is elaborated in the environment of the sequential declaration overlaid by the bindings produced by* `D1`

Semantics

Example: The (informally specified) semantics of commands in Mini Triangle.

Commands are executed to update variables and/or perform input output.

The assignment command `V := E` *is executed as follows:*

> *first the expression* `E` *is evaluated to yield a value* ***v***
>
> *then* ***v*** *is assigned to the variable named* `V`

The sequential command `C1;C2` *is executed as follows:*

> *first the command* `C1` *is executed*
>
> *then the command* `C2` *is executed*

etc.

Structured operational semantics

$[\text{ass}_{\text{ns}}]$	$\langle x := a,\ s\rangle \rightarrow s[x\mapsto\mathcal{A}[\![a]\!]s]$
$[\text{skip}_{\text{ns}}]$	$\langle \texttt{skip},\ s\rangle \rightarrow s$
$[\text{comp}_{\text{ns}}]$	$\dfrac{\langle S_1,\ s\rangle \rightarrow s',\ \langle S_2,\ s'\rangle \rightarrow s''}{\langle S_1;S_2,\ s\rangle \rightarrow s''}$
$[\text{if}_{\text{ns}}^{\text{tt}}]$	$\dfrac{\langle S_1,\ s\rangle \rightarrow s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \rightarrow s'}$ if $\mathcal{B}[b]s = \mathbf{tt}$
$[\text{if}_{\text{ns}}^{\text{ff}}]$	$\dfrac{\langle S_2,\ s\rangle \rightarrow s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \rightarrow s'}$ if $\mathcal{B}[b]s = \mathbf{ff}$
$[\text{while}_{\text{ns}}^{\text{tt}}]$	$\dfrac{\langle S,\ s\rangle \rightarrow s',\ \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \rightarrow s''}{\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \rightarrow s''}$ if $\mathcal{B}[b]s = \mathbf{tt}$
$[\text{while}_{\text{ns}}^{\text{ff}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \rightarrow s$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Several needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies

Semantic styles

- Structural Operational Semantics
  - Sebesta's book has a very narrow view
  - Much better view in
    - Transitions and Trees: An introduction to structural operational semantics, Cambridge University Press
- Denotational Semantics
  - Based on recursive function theory
  - Originally developed by Scott and Strachey (1970)
- Axiomatic Semantics
  - Sometimes called Hoare Logic
  - Original purpose: formal program verification

Important!

- Syntax is the visible part of a programming language
  - Programming Language designers can waste a lot of time discussing unimportant details of syntax
  - But syntax is important – syntax should convey the meaning intutively
- The language paradigm is the next most visible part
  - The choice of paradigm, and therefore language, depends on how humans best think about the problem
    - Imperative, Object Oriented, Functional, ..
  - There are no right models of computations – just different models of computations, some more suited for certain classes of problems than others
- The most invisible part is the language semantics
  - Clear semantics usually leads to simple and efficient implementations

Before Language definition

- Write programs !!
- Serves as inspiration for language specification
  - Syntax
    - Tokens
    - CFG
  - Static semantics
    - Scope rules
    - Type rules
  - Semantics
    - Informal
    - Formal
- Serves as test case for compiler !!
- Read language specifications: C, C#, Java, ..

Languages and Compilers
(SProg og Oversættere)

Lecture 5
Context Free Grammars

Bent Thomsen

Department of Computer Science

Aalborg University

Programming Language Specification

- A Language specification has (at least) three parts
  - Syntax of the language:
    - **usually formal CFG in BNF or EBNF**
    - Tokens defined using regular expressions (RE)
  - Contextual constraints:
    - scope rules (often written in English, but can be formal)
    - type rules (formal or informal)
  - Semantics:
    - defined by the implementation
    - informal descriptions in English
    - formal using operational or denotational semantics

Syntax Specification

Syntax is specified using “Context Free Grammars”:

- A finite set of **terminal symbols**
- A finite set of **non-terminal symbols**
- A **start symbol**
- A finite set of **production rules**

A CFG defines a set of strings

- This is called the language of the CFG.

How to design a grammar?

- Let's write a CFG for C-style function prototypes!
- Write examples:
  - void myf1(int x, double y);
  - int myf2();
  - int myf3(double z);
  - double myf4(int, int w, int);
  - void myf5(void);

- Terminals: void, int, double, (,), , , ; , ident
  - ident = [a-z]([a-z]|[0-9])*

Designing a grammar for Function Prototypes

- Here is one possible grammar

  S → Ret ident (Args);
  Ret → Type | void
  Type → int | double
  Args → ε | void | ArgList
  ArgList → OneArg | ArgList, OneArg
  OneArg → Type | Type ident

- Examples
  - void ident(int ident, double ident);
  - int ident();
  - int ident(double ident);
  - double ident(int, int ident, int);
  - void ident(void);

Designing a grammar for Function Prototypes

- Here is another possible grammar

S → Ret ident Args ;

Ret → int | double | void

Type → int | double

Args → () | (void)| (ArgList)

ArgList → OneArg |OneArg,ArgListArg

OneArg → Type | Type ident

- Examples
  - void ident(int ident, double ident);
  - int ident();
  - int ident(double ident);
  - double ident(int, int ident, int);
  - void ident(void);

Context-Free Grammars

- Components: $G=(N,\Sigma,P,S)$
  - A finite **terminal alphabet** Σ: the set of tokens produced by the scanner
  - A finite **nonterminal alphabet** N: variables of the grammar
  - A **start symbol** S: $S \in N$ that initiates all derivations
    - *Goal symbol*
  - A finite set of **productions** P: $A \rightarrow X_1 \ldots X_m$, where $A \in N$, $X_i \in N \cup \Sigma$, $1 \leq i \leq m$ and $m \geq 0$.
    - *Rewriting rules*
- Vocabulary $V=N \cup \Sigma$
  - $N \cap \Sigma = \phi$

- CFG: recipe for creating strings
- *Derivation*: a rewriting step using the production A→α replaces the nonterminal A with the vocabulary symbols in α
  - Left-hand side (LHS): A
  - Right-hand side (RHS): α
- *Context-free language* of grammar G *L(G)*: the set of terminal strings derivable from S

- notation:
  - $A \rightarrow \alpha$
    $\mid \beta$
    …
    $\mid \zeta$
- or
  - $A \rightarrow \alpha$
    $A \rightarrow \beta$
    …
    $A \rightarrow \zeta$

- $\alpha A \beta \Rightarrow \alpha \gamma \beta$: one step of *derivation* using the production $A \rightarrow \gamma$
  - $\Rightarrow^{+}$: derives in one or more steps
  - $\Rightarrow^{*}$: derives in zero or more steps
- $S \Rightarrow^{*} \beta$: β is a sentential form of the CFG
- SF(G): the set of sentential forms of G
- $L(G)=\{w \in \Sigma^* \mid S \Rightarrow^{+} w\}$
  - $L(G)=SF(G) \cap \Sigma^*$

Two conventions that nonterminals are rewritten in some systematic order

Leftmost derivation: from left to right

Rightmost derivation: from right to left

Leftmost Derivation

- A derivation that always chooses the leftmost possible nonterminal at each step
  - $=>_{lm}$, $=>^{+}_{lm}$, $=>^{*}_{lm}$
  - A left sentential form
    - A sentential form produced via a leftmost derivation
    - E.g. production sequence in top-down parsers
    - (Fig. 4.1)

1	E	$\rightarrow$	Prefix (E)
2		|	v Tail
3	Prefix	$\rightarrow$	f
4		|	λ
5	Tail	$\rightarrow$	+ E
6		|	λ

Figure 4.1: A simple expression grammar.

- E.g: a leftmost derivation of f (v + v)
  - $E \Rightarrow_{lm}$ Prefix (E)
    $\Rightarrow_{lm}$ f (E)
    $\Rightarrow_{lm}$ f (v Tail)
    $\Rightarrow_{lm}$ f (v + E)
    $\Rightarrow_{lm}$ f (v + v Tail)
    $\Rightarrow_{lm}$ f (v + v)

1	E	$\rightarrow$	Prefix (E)
2		|	v Tail
3	Prefix	$\rightarrow$	f
4		|	λ
5	Tail	$\rightarrow$	+ E
6		|	λ

Rightmost Derivations

- The rightmost possible nonterminal is always expanded
  - $=>_{rm}$, $=>^{+}_{rm}$, $=>^{*}_{rm}$
  - A right sentential form
    - A sentential form produced via a rightmost derivation
    - E.g. produced by bottom-up parsers (Ch. 6)
    - (Fig. 4.1)

- E.g: a rightmost derivation of f (v + v)
  - E $=>_{rm}$ Prefix (E)
    $=>_{rm}$ Prefix (v Tail)
    $=>_{rm}$ Prefix (v + E)
    $=>_{rm}$ Prefix (v + v Tail)
    $=>_{rm}$ Prefix (v + v)
    $=>_{rm}$ f (v + v)

1	E	$\rightarrow$	Prefix (E)
2		|	v Tail
3	Prefix	$\rightarrow$	f
4		|	λ
5	Tail	$\rightarrow$	+ E
6		|	λ

Parse Trees

- Parse tree: graphical representation of a derivation
  - Root: start symbol S
  - Each node: either grammar symbol or λ (or ε)
  - Interior nodes: nonterminals
    - An interior node and its children: production
  - E.g. Fig. 4.2

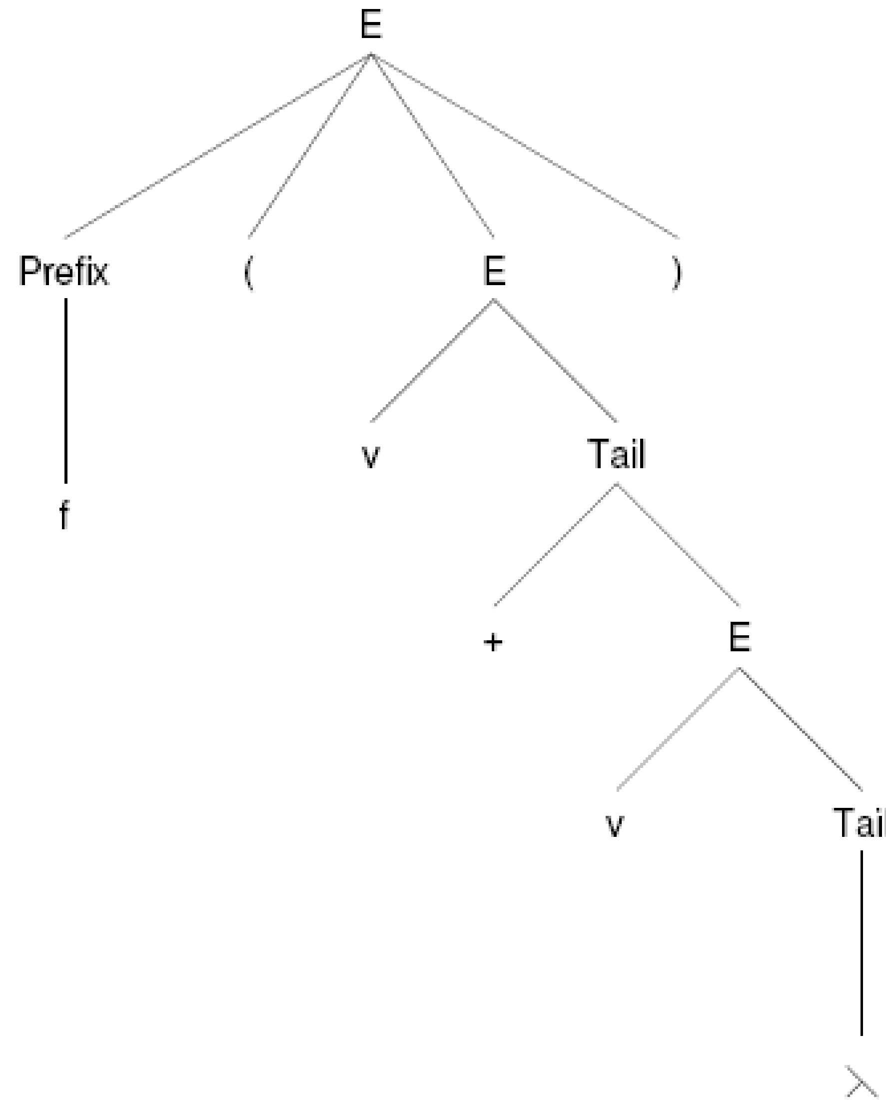


Figure 4.2: The parse tree for f (v + v) .

BNF form of grammars

- Backus-Naur Form (BNF) is a formal grammar for expressing context-free grammars.
- The single grammar rule format:
  - Non-terminal → zero or more grammar symbols

- It is usual to combine all rules with the same left-hand side into one rule, such as:

  N → α
  N → β
  N → γ

  Greek letters α,β, or γ means a string of symbols.

  are combined into one rule:

  N → α | β | γ

  α, β and γ are called the ***alternatives*** of N.

Extended BNF form of grammars

- BNF is very suitable for expressing nesting and recursion, but less convenient for repetition and optionality.

- Three additional postfix operators +,?, and *, are thus introduced:
  - R+ indicates the occurrence of one or more Rs, to express repetition (sometime R_opt isused).
  - R? indicates the occurrence of zero or one Rs, to express optionality (sometimes [R] is used).
  - R* indicates the occurrence of zero or more Rs, to express repetition (sometimes {R} is used).
- The grammar that allows the above is called Extended BNF (EBNF).

Extended forms of grammars

An example is the grammar rule in EBNF:

parameter_list →

('IN' | 'OUT')? identifier (',' identifier)*

or

parameter_list →

['IN' | 'OUT'] identifier {',' identifier}

which produces program fragments like:

a, b

IN year, month, day

OUT left, right

Extended forms of grammars

- Rewrite EBNF grammar to CFG
  – Given the EBNF grammar:

    expression → term (+ term)*

  Rewrite it to:

    expression → term term_tmp

    term_tmp → + term term_tmp

    | λ

foreach $p \in Prods$ of the form "$\mathsf{A} \rightarrow \alpha$ [$\mathcal{X}_1 \ldots \mathcal{X}_n$] β" **do**

$N \leftarrow$ NewNonTerm( )

$p \leftarrow$ "$\mathsf{A} \rightarrow \alpha\ N\ \beta$"

$Prods \leftarrow Prods \cup \{$"$N \rightarrow \mathcal{X}_1 \ldots \mathcal{X}_n$"$\}$

$Prods \leftarrow Prods \cup \{$"$N \rightarrow \lambda$"$\}$

foreach $p \in Prods$ of the form "$\mathsf{B} \rightarrow \gamma$ { $\mathcal{X}_1 \ldots \mathcal{X}_m$ } δ" **do**

$M \leftarrow$ NewNonTerm( )

$p \leftarrow$ "$\mathsf{B} \rightarrow \gamma\ M\ \delta$"

$Prods \leftarrow Prods \cup \{$"$M \rightarrow \mathcal{X}_1 \ldots \mathcal{X}_n\ M$"$\}$

$Prods \leftarrow Prods \cup \{$"$M \rightarrow \lambda$"$\}$

Figure 4.4: Algorithm to transform a BNF grammar into standard form.

Properties of grammars

- A non-terminal N is **left-recursive** if, starting with a sentential form N, we can produce another sentential form starting with N.
  - ex: expression → expression ‘+’ factor | factor

- right-recursion also exists, but is less important.
  - ex: expression → term ‘+’ expression

Properties of grammars (Cont.)

- A non-terminal N is **nullable**, if starting with a sentential form N, we can produce an empty sentential form.

  example:

  expression → λ

- A non-terminal N is **useless**, if it can never produce a string of terminal symbols.

  example:

  expression → + expression
  | - expression

Grammar Transformations

Left factorization

$$X\ Y\ |\ X\ Z \Rightarrow X(Y|Z)$$

X Y=λ Z

Example:

```
single-Command
   ::= V-name := Expression
     | if Expression then single-Command
     | if Expression then single-Command
                          else single-Command
```

```
single-Command
   ::= V-name := Expression
     | if Expression then single-Command
                 ( λ | else single-Command)
```

Grammar Transformations (ctd)

Elimination of Left Recursion

N ::= X | N Y ⇒ N ::= X Y*

N ::= X | N Y ⇒ N ::= X M
M ::= Y M | λ

Example:

```
Identifier ::= Letter
             | Identifier Letter
             | Identifier Digit
```

```
Identifier ::= Letter
             | Identifier (Letter|Digit)
```

```
Identifier ::= Letter (Letter|Digit)*
```

Grammar Transformations (ctd)

Substitution of non-terminal symbols

$$N ::= X \qquad\qquad \Rightarrow \qquad\qquad N ::= X$$
$$M ::= \alpha\ N\beta \qquad\qquad\qquad M ::= \alpha\ X\beta$$

Example:

```
single-Command
    ::= for contrVar := Expression
        to-or-dt Expression do single-Command
to-or-dt ::= to | downto
```

```
single-Command ::=
     for contrVar := Expression
     (to|downto) Expression do single-Command
```

From tokens to parse tree

The process of finding the structure in the flat stream of tokens is called **parsing**, and the module that performs this task is called **parser**.

Parsing methods

There are two well-known ways to parse:

1) top-down

Left-scan, **L**eftmost derivation (**LL**).

2) bottom-up

Left-scan, **R**ightmost derivation in reverse (**LR**).

- LL constructs the parse tree in pre-order;
- LR in post-order.

Different kinds of Parsing Algorithms

- Two big groups of algorithms can be distinguished:
  - bottom up strategies
  - top down strategies
- Example parsing of "Micro-English"

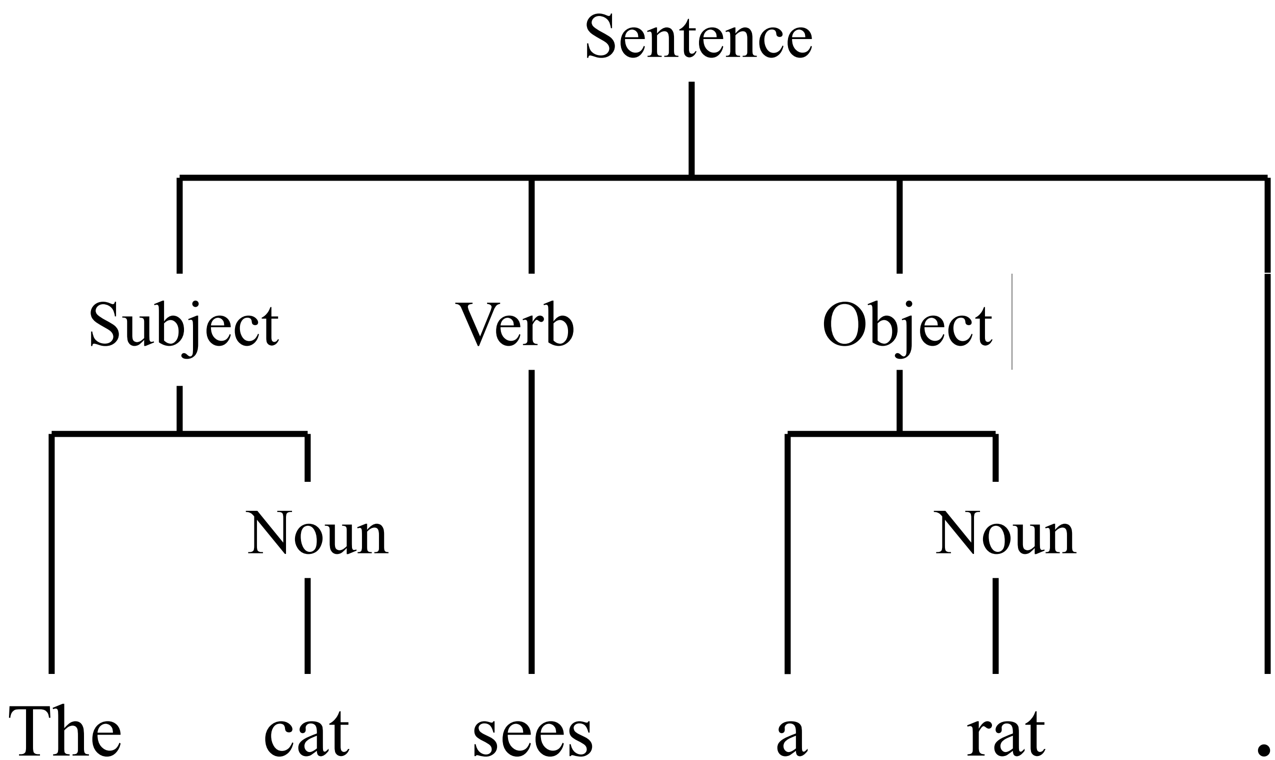
```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

The cat sees the rat.
The rat sees me.
I like a cat

The rat like me.
I see the rat.
I sees a rat.

Top-down parsing

The parse tree is constructed starting at the top (root).



Left derivations

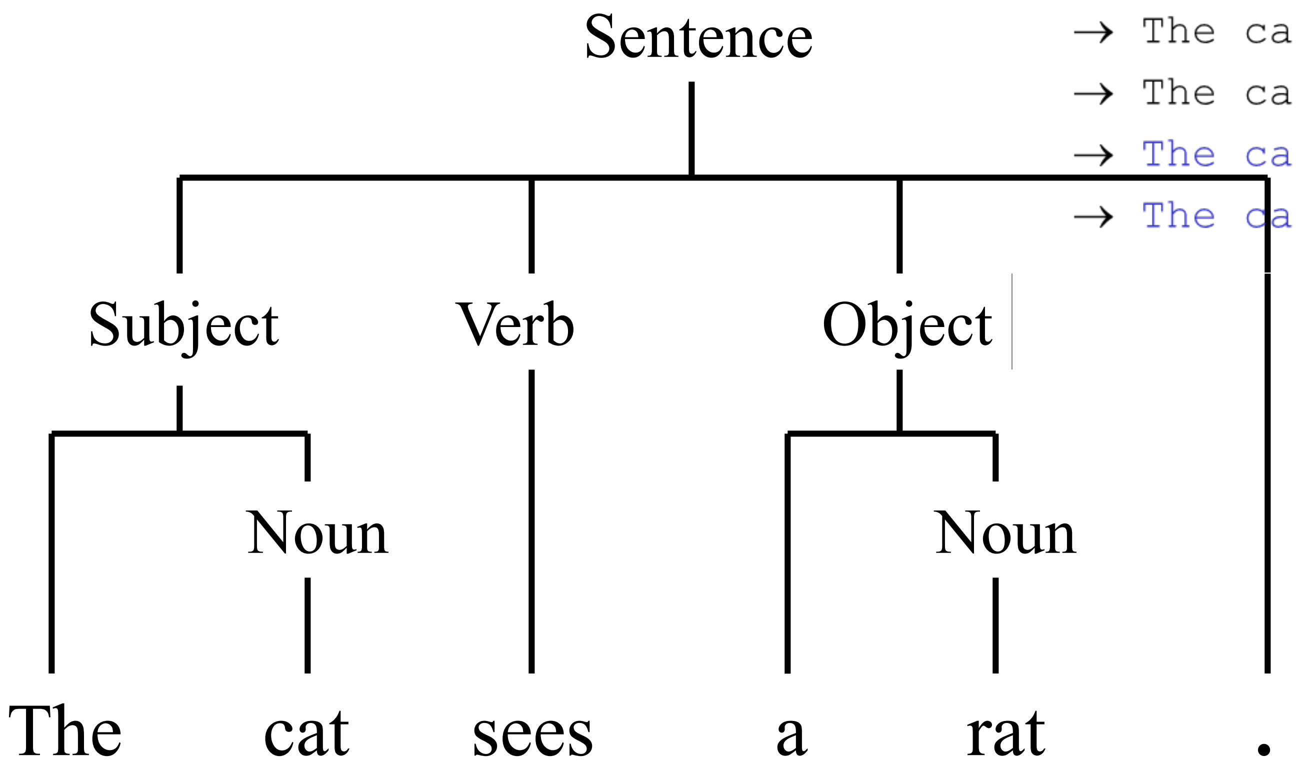
```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

```
Sentence
→ Subject Verb Object .
→ The Noun Verb Object.
→ The cat Verb Object.
→ The cat sees Object.
→ The cat sees a Noun .
→ The cat sees a rat .
```

Top-down parsing

The parse tree is constructed starting at the top (root).

```
Sentence
→ Subject Verb Object .
→ The Noun Verb Object.
→ The cat Verb Object.
→ The cat sees Object.
→ The cat sees a Noun .
→ The cat sees a rat .
```



Right derivations

```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

Sentence

→ Subject Verb Object .

→ Subject Verb a Noun .

→ Subject Verb a rat .

→ Subject sees a rat .

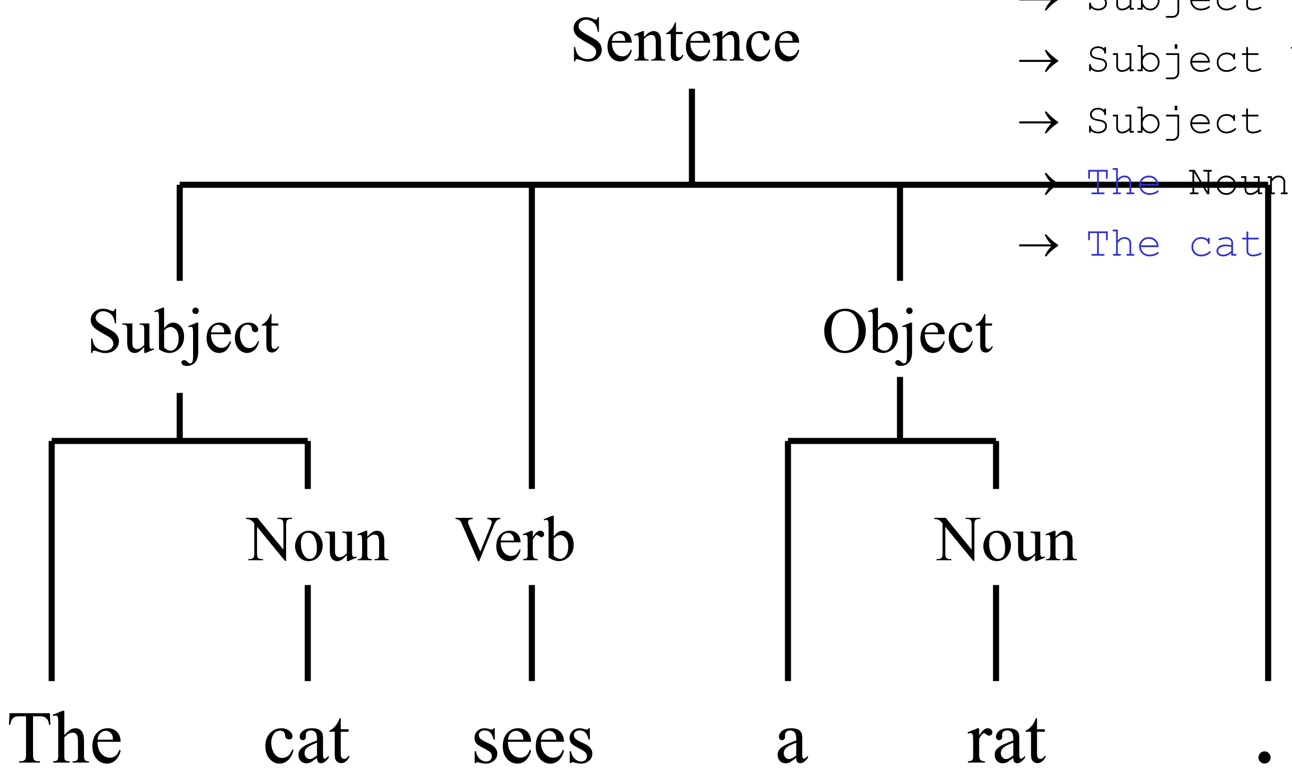
→ The Noun sees a rat .

→ The cat sees a rat .

Bottom up parsing

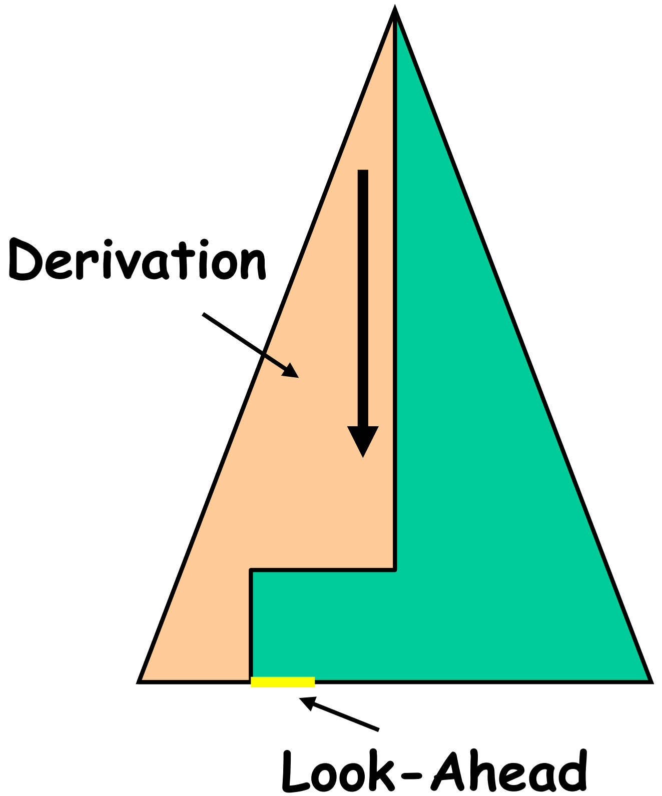
The parse tree “grows” from the bottom (leafs) up to the top (root).
Just read the right derivations backwards

```
Sentence
→ Subject Verb Object .
→ Subject Verb a Noun .
→ Subject Verb a rat .
→ Subject sees a rat .
→ The Noun sees a rat .
→ The cat sees a rat .
```

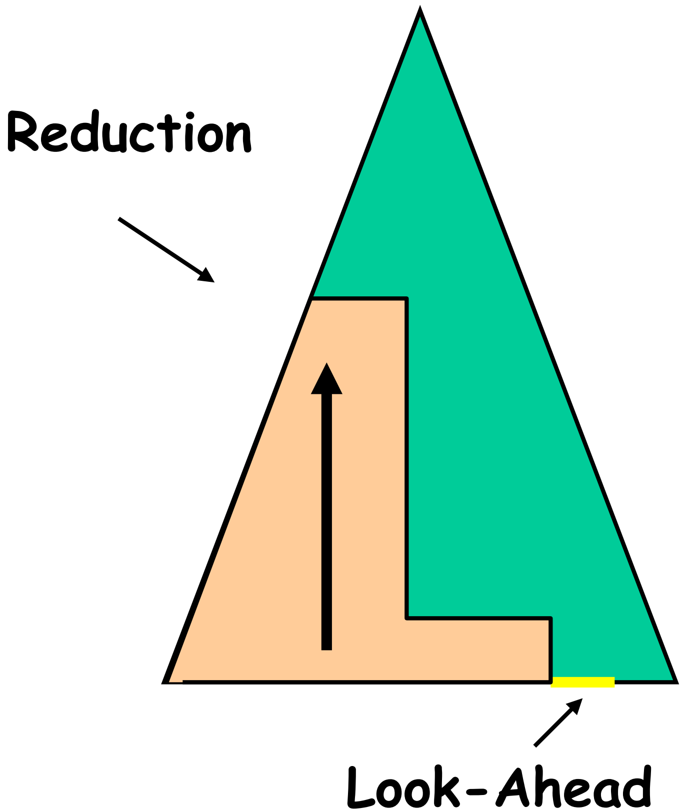


Top-Down vs. Bottom-Up parsing

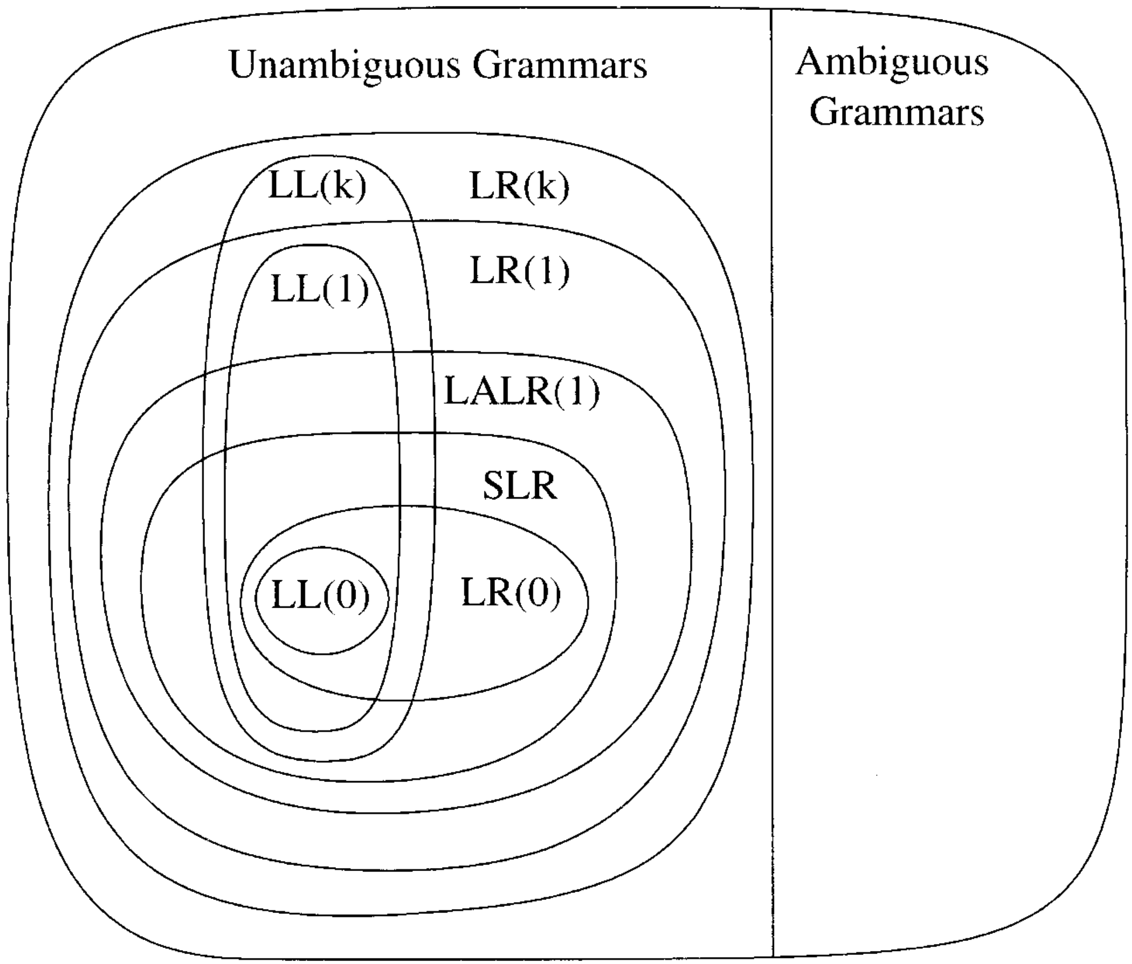
LL-Analyse (Top-Down)
Left-to-Right Left Derivative



LR-Analyse (Bottom-Up)
Left-to-Right Right Derivative



Hierarchy


Unambiguous Grammars
Ambiguous Grammars
LL(k)
LR(k)
LL(1)
LR(1)
LALR(1)
SLR
LL(0)
LR(0)

Pause

Formal definition of LL(1)

A grammar G is LL(1) iff
for each set of productions $X ::= X_1 \mid X_2 \mid \ldots \mid X_n$:

1. $first[X_1], first[X_2], \ldots, first[X_n]$ are all pairwise disjoint
2. If $X_i =>^* \lambda$ then $first[X_j] \cap follow[X] = \emptyset$, for $1 \leq j \leq n. i \neq j$

If G is λ-free then 1 is sufficient

Define FIRST(α),where α is any string of grammar symbols, to be:
the set of terminals
that begin strings derived from α

First Sets

- The set of all terminal symbols that can begin a sentential form derivable from the string α
  - First(α)={ a∈Σ | α=>*aβ }
  - We never include λ in First(α) even if α=>λ
  - E.g. (in Fig.4.1)
    - First(Tail) = {+}
    - First(Prefix) = {f}
    - First(E) = {v, f, (}

1	E	→	Prefix (E)
2		|	v Tail
3	Prefix	→	f
4		|	λ
5	Tail	→	+ E
6		|	λ

```
function FIRST(α) returns Set
    foreach A ∈ NONTERMINALS( ) do VisitedFirst(A) ← false          (9)
    ans ← INTERNALFIRST(α)
    return (ans)
end
function INTERNALFIRST(Xβ) returns Set
    if Xβ = ⊥                                                        (10)
    then return (∅)
    if X ∈ Σ                                                         (11)
    then return ({ X })
    /*    X is a nonterminal.                                      */ (12)
    ans ← ∅
    if not VisitedFirst(X)
    then
        VisitedFirst(X) ← true                                       (13)
        foreach rhs ∈ ProductionsFor(X) do
            ans ← ans ∪ INTERNALFIRST(rhs)                           (14)
    if SymbolDerivesEmpty(X)                                         (15)
    then ans ← ans ∪ INTERNALFIRST(β)
    return (ans)                                                     (16)
end
```

Figure 4.8: Algorithm for computing First(α).

Follow Sets

- The set of terminals that can follow a nonterminal A in some sentential form
  - For $A \in N$,
    - $\text{Follow}(A) = \{b \in \Sigma \mid S \Rightarrow^{+} \alpha A b \beta\}$
  - The right context associated with A
  - Fig. 4.11

Follow Sets

- Follow(A) is the set of prefixes of strings of terminals that can follow any derivation of A in G
  - $\$ \in$ follow(S) (sometimes <eof> $\in$ follow(S))
  - if $(B \rightarrow \alpha A \beta) \in P$, then
  - first(β)$\oplus$follow(B)$\subseteq$ follow(A)

- The definition of follow usually results in recursive set definitions. In order to solve them, you need to do several iterations on the equations.
  - E.g. (in Fig.4.1)
    - Follow(Tail) = {)}
    - Follow(Prefix) = {(}
    - Follow(E) = {\$,)}

```
1  E      → Prefix ( E )
2         | v Tail
3  Prefix → f
4         | λ
5  Tail   → + E
6         | λ
```

```
function Follow( A ) returns Set
    foreach A ∈ NonTerminals( ) do
        VisitedFollow(A) ← false                                  (17)
    ans ← InternalFollow( A )
    return (ans)
end
function InternalFollow( A ) returns Set
    ans ← ∅
    if not VisitedFolow(A)                                        (18)
    then
        VisitedFollow(A) ← true                                   (19)
        foreach a ∈ Occurrences( A ) do                           (20)
            ans ← ans ∪ First( Tail( a ) )                        (21)
            if AllDeriveEmpty( Tail( a ) )                        (22)
            then
                targ ← LHS( Production( a ) )
                ans ← ans ∪ InternalFollow( targ )                (23)
    return (ans)                                                  (24)
end
function AllDeriveEmpty( γ ) returns Boolean
    foreach 𝒳 ∈ γ do
        if not SymbolDerivesEmpty(𝒳) or 𝒳 ∈ Σ
        then return (false)
    return (true)
end
```

Figure 4.11: Algorithm for computing Follow(A).

A few provable facts about LL(1) grammars

- No left-recursive grammar is LL(1)
- No ambiguous grammar is LL(1)
- Some languages have no LL(1) grammar
- A λ-free grammar, where each alternative X_j for N ::= X_j begins with a distinct terminal, is a simple LL(1) grammar

LR Grammars

- A Grammar is an LR Grammar if it can be parsed by an LR parsing algorithm
- Harder to implement LR parsers than LL parsers
  - but tools exist (e.g. JavaCUP, Yacc, C#CUP and SableCC)
- Can recognize LR(0), LR(1), SLR, LALR grammars (bigger class of grammars than LL)
  - Can handle left recursion!
  - Usually more convenient because less need to rewrite the grammar.
- LR parsing methods are the most commonly used for automatic tools today (LALR in particular)

Other Types of Grammars

- Regular grammars: less powerful
- Context-sensitive and unrestricted grammars: more powerful
- Parsing Expression Grammars

Designing CFGs is a craft.

- When thinking about CFGs:
  - Think recursively: Build up bigger structures from smaller ones.
- Have a construction plan:
  - Know in what order you will build up the string.
- Store information in nonterminals:
  - Have each nonterminal correspond to some useful piece of information.

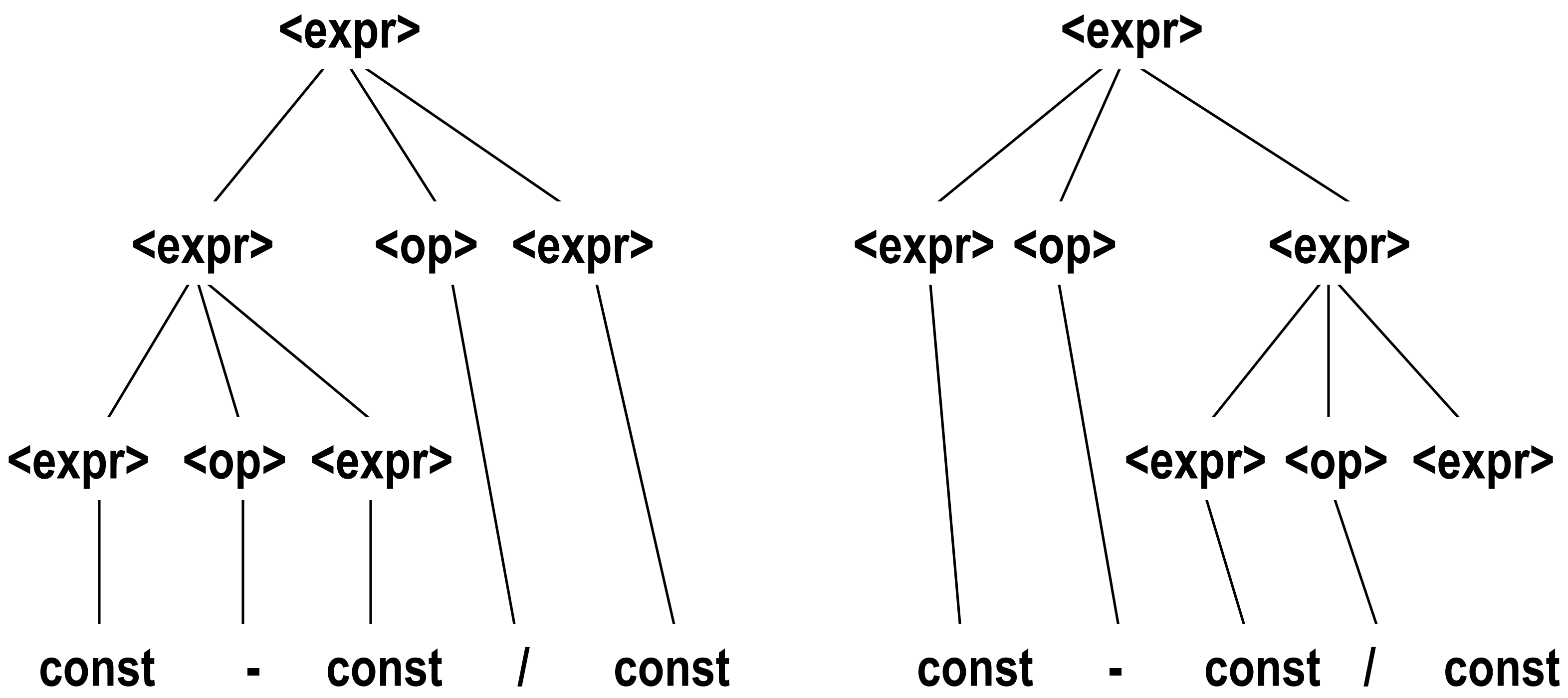
Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

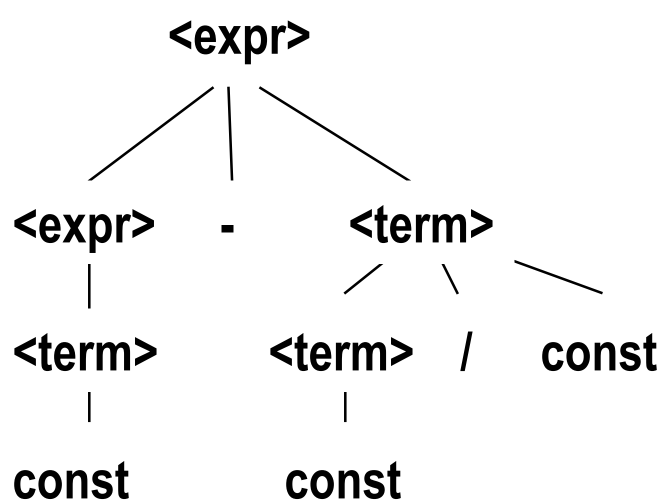
<op> → / | -



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```

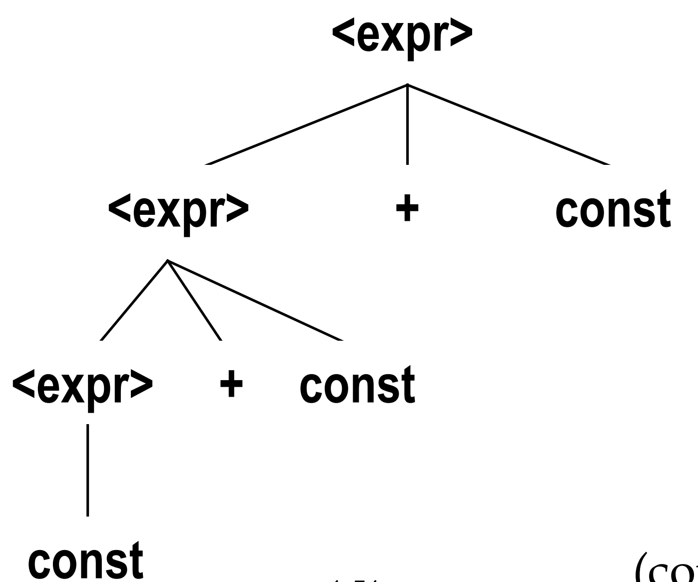


const – (const / const)

Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



(const + const) + const

Associativity and Left Resursion

```
<expr> -> <expr> + const  |  const
(unambiguous, but left recursive)

<expr> -> const + <expr>  |  const
(unambiguous, right recursive, but => right assoc.)

i.e. const + (const + const)
Not a problem for +, but what about - ?

(5 - 3) - 2 = 0
5 - (3 - 2) = 4
```

Eliminating Left recursion

```
<expr> -> <expr> (+ <expr>)*

or

<expr> -> const <exprlist>
<exprlist> -> + const <exprlist> | λ
```

Still gives the wrong parse tree, but this can be sorted when generating AST

Hidden left-factors and hidden left recursion

- Sometimes, left-factors or left recursion are hidden
- Examples:
  - The following grammar:
    - A -> da | ac B
    - B -> ab B | da A | A f
  - has two overlapping productions: B -> da A and B =>*daf .
  - The following grammar:
    - S -> T u | wx
    - T -> S q | vv S
  - has left recursion on T (T =>* Tuq)

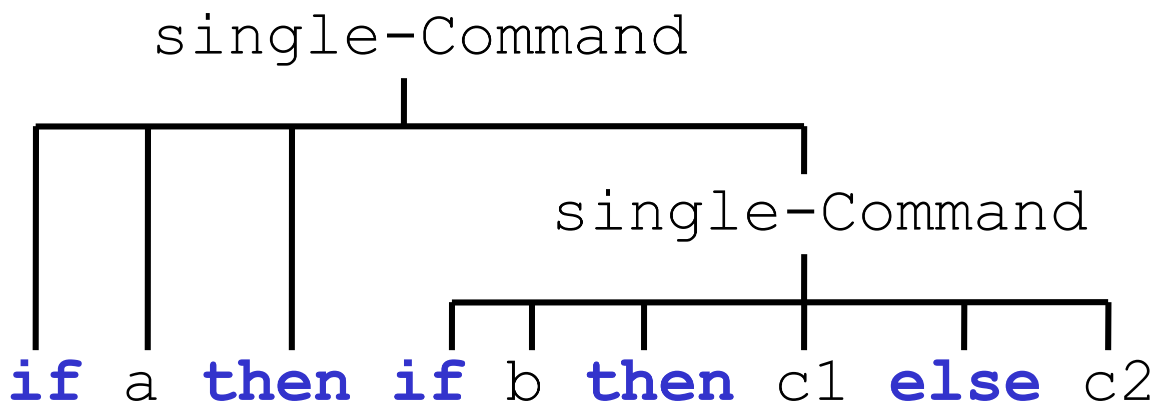
- Solution: expand the production rules by substitution to make
- left-recursion or left factors visible and then eliminate them

Dangling Else Problem

Example: (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                      else single-Command
```

This parse tree?

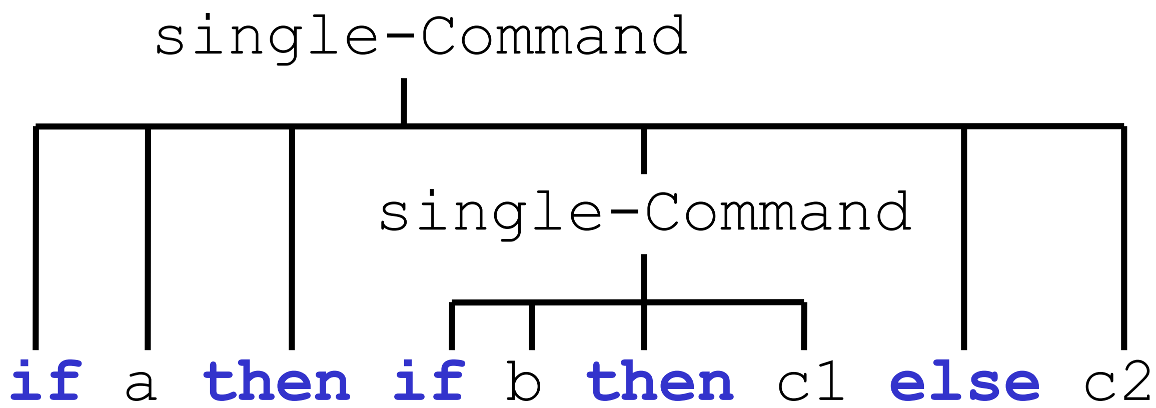


Dangling Else Problem

Example: (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                         else single-Command
```

or this one ?



Dangling Else Problem

Example: "dangling-else" problem (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                          else single-Command
```

Rewrite Grammar:

```
sC ::= if E then sC endif
     |  if E then sC else sC endif
```

Dangling Else Problem

Example: "dangling-else" problem (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                          else single-Command
```

Rewrite Grammar:

```
sC   ::= CsC
       |   OsC
CsC  ::= if E then CsC else CsC
CsC  ::= ...
OsC  ::= if E then sC
       |   if E then CsC else OsC
```

Ambiguity

- Sometimes obvious
  - Exp ::= Exp + Exp
- Sometimes difficult to spot
- Undecidable Property (known since 1962)

- Engineering approach
  - Try a parser generator
  - Use a Grammar engineering toolbox
    - KfG in AtoCC
    - Context Free Grammer tools
      - http://smlweb.cpsc.ucalgary.ca/start.html
      - http://mdaines.github.io/grammophone/

- Try ACLA
  - (Ambiguity Checking with Language Approximations)
  - http://services2.brics.dk/java/grammar/demo.html

What can you do in your project?

- Start writing a CFG
  - Define keywords, identifiers, numbers, ..
  - Define productions
- Test it with
  - kfG Edit
  - Context Free Grammer tool
  - ACLA

You may need more than one Grammar

- Abstract Syntax
  - To communicate the essentials of the language
  - To serve as design pattern for AST
  - To serve in the formal specification of the semantics
  - May be ambiguous
- Concrete Syntax
  - The grammar we use as specification for building a parser
  - Must be unambiguous

- Lexical elements (Syntax given as Regular Expressions)
  - Identifiers e.g. Id := [a-z]([a-z]|[0-9])*
  - Keywords (or reserved words)
    - if, then, while,
    - begin .. end v.s. { .. }

Grammar tools

- Demo
  - Prefix
  - Exp with ambiguity and without
  - Dangling else
  - LL(1) – first and follow

Languages and Compilers (SProg og Oversættere)

Lecture 6 Lexical Analysis

Bent Thomsen
Department of Computer Science
Aalborg University

Learning goals

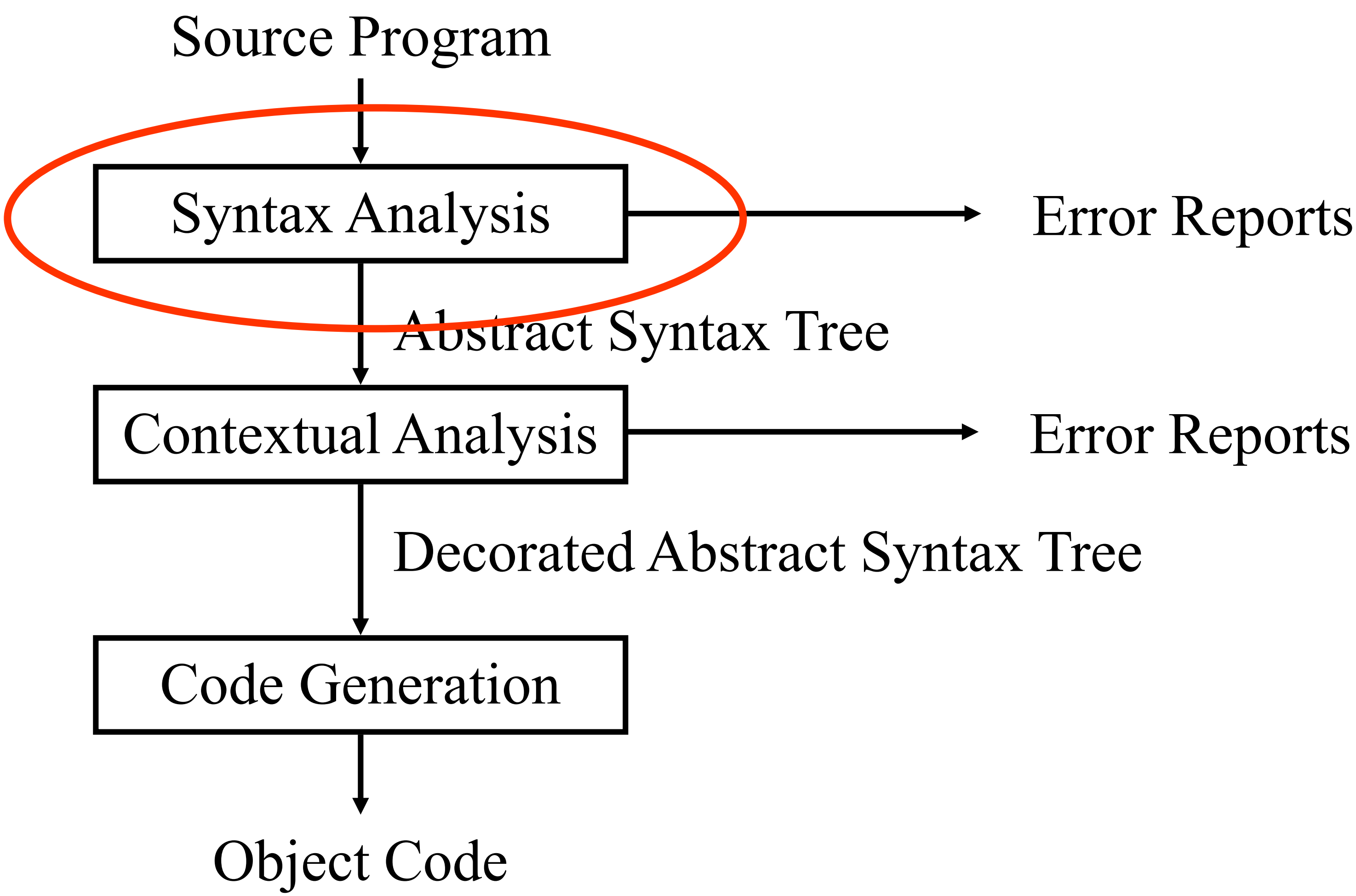
- Understand the lexical analysis phase of the compiler
- Understand the role of regular expressions
- Understand the structure of the lexical analysis
- Understand the role of finite automata
- Get an overview of the Jlex tool

Remember exercise 4 from before lecture 1 ?

- Write a Java program that can read the string “a + n * 1” and produce a collection of objects containing the individual symbols when blank spaces are ignored (or used as separator).

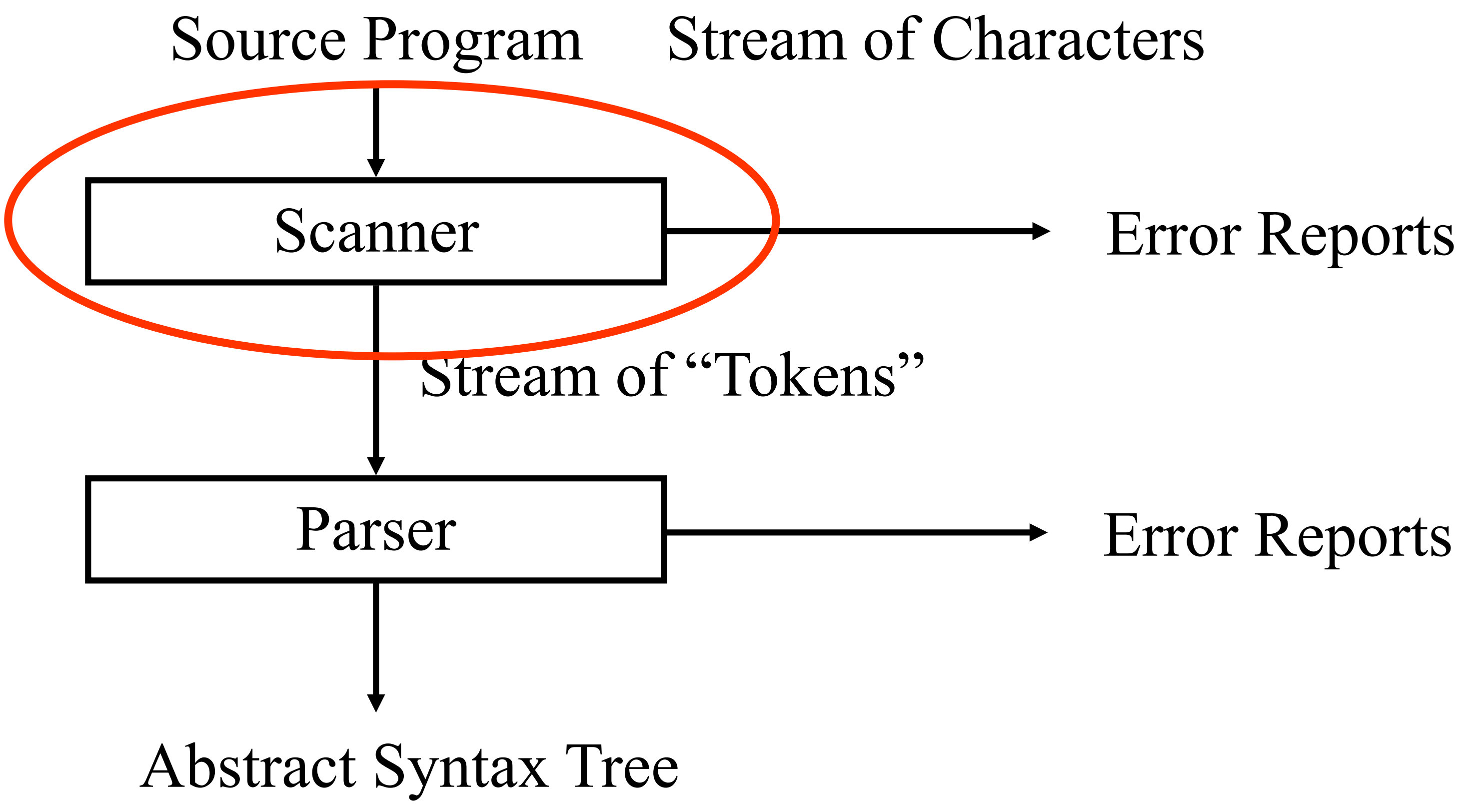
- Today we shall see several ways of solving this exercise

The “Phases” of a Compiler



Syntax Analysis: Scanner

Dataflow chart



1) Scan: Divide Input into Tokens

An example ac source program:

```
f b
i a
a = 5
b = a + 3.2
p b
```

Lexems are “words” in the input, for example keywords, operators, identifiers, literals, etc.
Tokens is a datastructure for lexems and additional information

scanner

floatdl	*id*	*intdcl*	*id*	*id*	*assign*	*inum*	
f	b	i	a	a	=	5	...

	assign	*id*	*plus*	*fnum*	*print*	*id*	*eot*
...	=	a	+	3.2	p	b	

Developing a Scanner

In Java the scanner will normally return instances of Token:

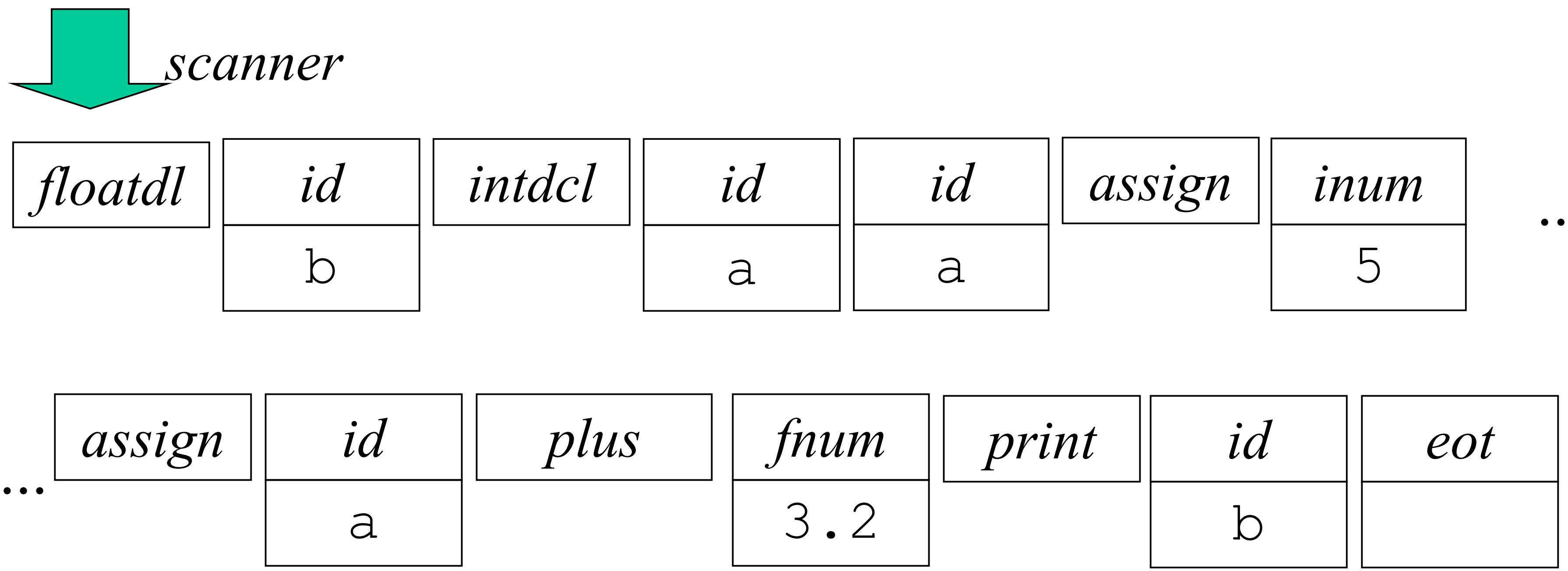
```
public class Token {
  byte kind; String spelling;
  final static byte
    IDENTIFIER = 0; INTLITERAL = 1; OPERATOR   = 2;
    BEGIN      = 3; CONST      = 4; ...
    ...

  public Token(byte kind, String spelling) {
    this.kind = kind; this.spelling = spelling;
}


  ...
}
```

1) Scan: Divide Input into Tokens

An example ac source program:

```
f b
i a
a = 5
b = a + 3.2
p b
```

Lexems are “words” in the input, for example keywords, operators, identifiers, literals, etc.
Tokens is a datastructure for lexems and additional information



Developing a Scanner

In Java the scanner will normally return instances of Token, but we could also use a subclass hierachy:

```
abstract class Token ..

public class IdentToken extends Token {
  String spelling;
...

  public IdentToken(String spelling) {
    this.spelling = spelling;
}

public class AssignToken extends Token {


  ...
}
```

Programming Language Specification

- A Language specification has (at least) three parts
  - Syntax of the language:
    - **Lexems/tokens as regular expressions**
      - » **Reserved words**
    - Grammar (CFG) - usually formal in BNF or EBNF
  - Contextual constraints:
    - scope rules (often written in English, but can be formal)
    - type rules (formal or informal)
  - Semantics:
    - defined by the implementation
    - informal descriptions in English
    - formal using operational or denotational semantics

Lexical Elements

- Character set
  - Ascii vs Unicode
- Identifiers
  - Java vs C#
- Operators
  - +, -, /, * , …
- Keywords
  - If, then, while
- Noise words
- Elementary data
  - numbers
    - integers
    - floating point
  - strings
  - symbols
- Delimiters
  - Begin .. End vs {…}
- Comments
  - /* vs. # vs. !
- Blank space
- Layout
  - Free- and fixed-field formats

Java Keywords

abstract continue for new switch assert default if package synchronized boolean do goto private this break double implements protected throw byte else import public throws case enum instanceof return transient catch extends int short try char final interface static void class finally long strictfp volatile const float native super while

- The keywords const and goto are reserved, even though they are not currently used.
- While true and false might appear to be keywords, they are technically Boolean literals
- Similarly, while null might appear to be a keyword, it is technically the null literal

Lexems

- The Lexem structure can be more detailed and subtle than one might expect
  - String constants: “”
    - Escape sequence: \”, \n, …
    - Null string
  - Rational constants
    - 0.1, 10.01,
    - .1, 10. vs. 1..10
- Design guideline:
  - if the lexem structure is complex then examine the language for design flaws !!
- Note recent research shows huge difference between novices and experienced programmers views on keywords:
  - repeat while … do .. end vs. while (..) {…}

(Try to) Avoide Weird Stuff

- PL/I
  - IF IF = THEN THEN = ELSE; ELSE ELSE = END; END
- C#
  - if (@if == then) then = @else; else @else = end;
- C
  - a (* b) … call of a with pointer to b or declaration on pointer b to a type where a is defined using typedef
- Whitespace language
  - Commands composed of sequences of spaces, tab stops and linefeeds

Simple grammar for Identifiers

Example:

```
Start ::= Letter
        | Start Letter
        | Start Digit
Letter ::= a | b | c | d | ... | z
Digit  ::= 0 | 1 | 2 | ... | 9
```

This grammar can be transformed to a regular expression:

[a-z]([a-z]|[0-9])*

Regular Expressions

ε	The empty string
t	Generates only the string *t*
X Y	Generates any string *xy* such that *x* is generated by *x* and *y* is generated by *Y*
X | *Y*	Generates any string which generated either by *X* or by *Y*
*X**	The concatenation of zero or more strings generated by *X*
(*X*)	For grouping

Identifier Grammar Easily Transform to RE

Elimination of Left Recursion

N ::= X | N Y ⇒ N ::= X $Y*$

Left factorization

X Y | X Z ⇒ $X(Y|Z)$

Example:

```
Identifier ::= Letter
             | Identifier Letter
             | Identifier Digit
```

↳

```
Identifier ::= Letter
             | Identifier (Letter|Digit)
```

↳

```
Identifier ::= Letter (Letter|Digit)*
```

Regular Grammers

- A grammar is regular if by substituting every nonterminal (except the root one) with its righthand side, you can reduce it down to a single production for the root, with only terminals and operators on the right-hand side.
- I.e. this grammer is regular:

```
Identifier ::= Letter
             | Identifier Letter
             | Identifier Digit
```

- Because it can be reduced to:

```
Identifier ::= Letter (Letter|Digit)*
```

Regular Grammers

- Or rather

```
( a | b | c | d | ... | z )((a | b | c | d | ... | z)|(0 | 1 | 2 | ... | 9 ))*
```

- Which is called a regular expression, often written as:

```
[a-z]([a-z]|[0-9])*
```

- Sometimes regular grammers are described as:
  - Right regular i.e. having the form A := a A | b
  - Left regular i.e. having the form A := A a | b

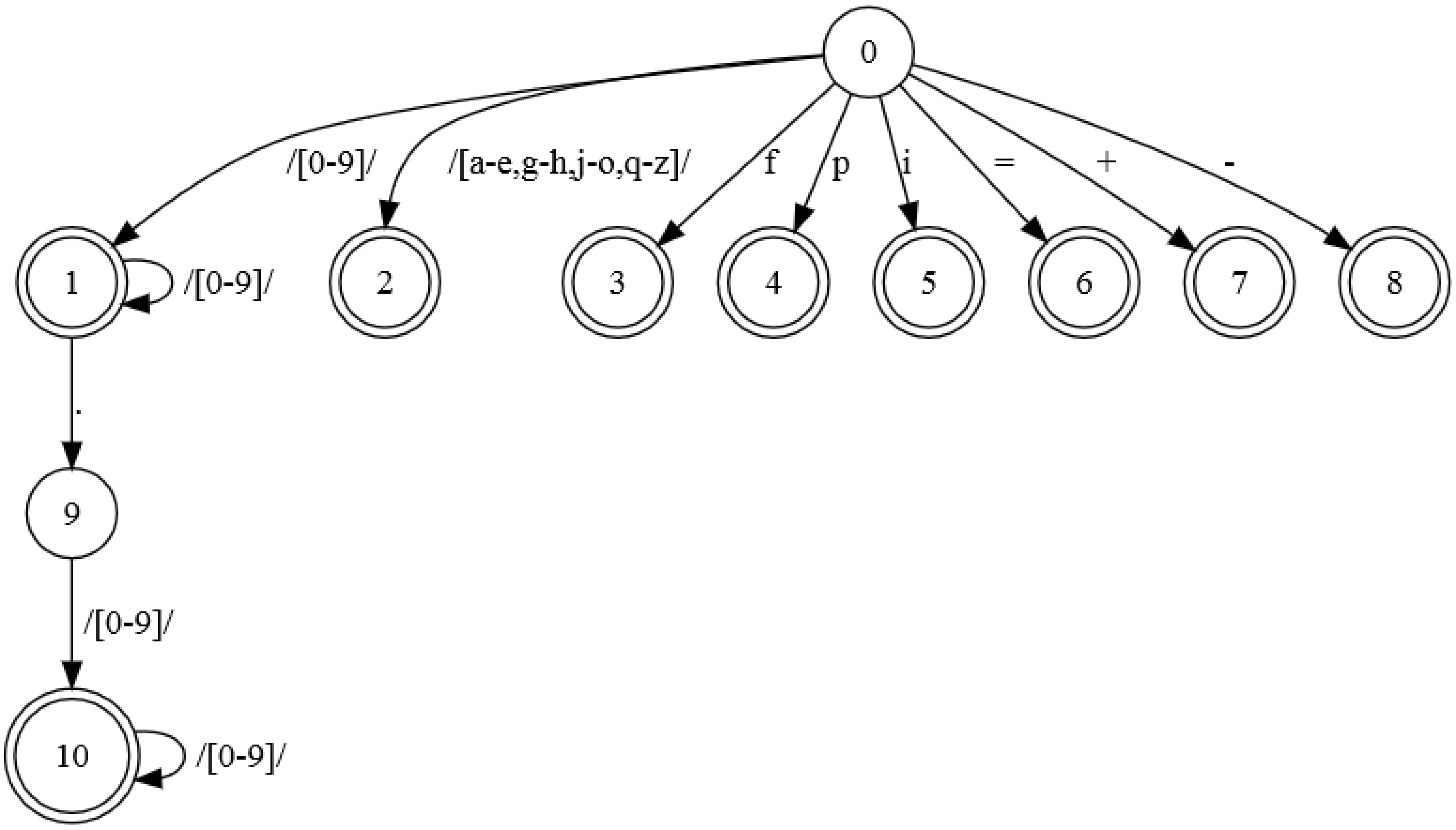
- Why are we so interested in Regular Expressions?
  - Because there are simple implementation techniques for Res
  - REs can be implemented via Finite State Machines (FSM)

ac Token Specification

Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	$[a - e] \mid [g - h] \mid [j - o] \mid [q - z]$
assign	"="
plus	"+"
minus	"-"
inum	$[0 - 9]^+$
fnum	$[0 - 9]^+ . [0 - 9]^+$
blank	$(" \ ")^+$

Figure 2.3: Formal definition of ac tokens.

[0-9]+ | [0-9]+.[0-9]+ | [a-e,g-h,j-o,q-z] | f | p | i | = | \+ | -



```
function SCANNER( ) returns Token
    while s.PEEK( ) = blank do call s.ADVANCE( )
    if s.EOF( )
    then ans.type ← $
    else
        if s.PEEK( ) ∈ { 0, 1, . . . , 9 }
        then ans ← SCANDIGITS( )
        else
            ch ← s.ADVANCE( )
            switch (ch)
                case { a, b, . . . , z } − { i, f, p }
                    ans.type ← id
                    ans.val ← ch
                case f
                    ans.type ← floatdcl
                case i
                    ans.type ← intdcl
                case p
                    ans.type ← print
                case =
                    ans.type ← assign
                case +
                    ans.type ← plus
                case -
                    ans.type ← minus
                case default
                    call LEXICALERROR( )
    return (ans)
end
```

Figure 2.5: Scanner for the ac language. The variable s is an input stream of characters.

```java
/**
 * Figure 2.5 code, processes the input stream looking
 *   for the next Token.
 * @return the next input Token
 */
public static Token Scanner() {
    Token ans;
    while (s.peek() == BLANK)
        s.advance();
    if (s.EOF())
        ans = new Token(EOF);
    else {
        if (isDigit(s.peek()))
            ans = ScanDigits();
        else {
            char ch = s.advance();

            switch(representativeChar(ch)) {
            case 'a':  // matches {a, b, ..., z} - {f, i, p}
                ans = new Token(ID, ""+ch); break;
            case 'f':
                ans = new Token(FLTDCL);  break;
            case 'i':
                ans = new Token(INTDCL);    break;
            case 'p':
                ans = new Token(PRINT);     break;
            case '=':
                ans = new Token(ASSIGN);    break;
            case '+':
                ans = new Token(PLUS);      break;
            case '-':
                ans = new Token(MINUS);     break;
            default:
                throw new Error("Lexical error on character with decimal value: " + (int)ch);

            }
        }

    }
    return ans;
}

/**
```

```
function ScanDigits( ) returns token
    tok.val ← " "
    while s.peek( ) ∈ {0, 1, . . . , 9} do
        tok.val ← tok.val + s.advance( )
    if s.peek( ) ≠ "."
    then  tok.type ← inum
    else
        tok.type ← fnum
        tok.val ← tok.val + s.advance( )
        while s.peek( ) ∈ {0, 1, . . . , 9} do
            tok.val ← tok.val + s.advance( )
    return (tok)
end
```

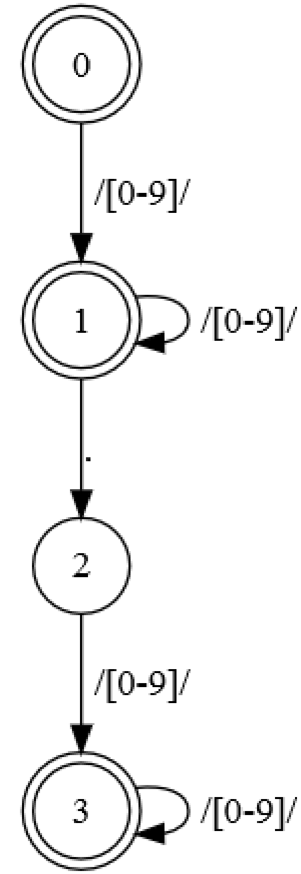


Figure 2.6: Finding inum or fnum tokens for the ac language.

```java
/**
 * Figure 2.6 code, processes the input stream to form
 *    a float or int constant.
 * @return the Token representing the discovered constant
 */

private static Token ScanDigits() {
    String val = "";
    int   type;
    while (isDigit(s.peek())) {
        val = val + s.advance();
    }
    if (s.peek() != '.')
        type = INUM;
    else {
        type = FNUM;
        val = val + s.advance();
        while (isDigit(s.peek())) {
            val = val + s.advance();
        }
    }
    return new Token(type, val);
}
```

How to change code to accept: 0 | [1-9][0-9]*(.[0-9]*)

```
function ScanDigits( ) returns token
    tok.val ← " "
    while s.peek( ) ∈ { 0, 1, . . . , 9 } do
        tok.val ← tok.val + s.advance( )
    if s.peek( ) ≠ "."
    then  tok.type ← inum
    else
        tok.type ← fnum
        tok.val ← tok.val + s.advance( )
        while s.peek( ) ∈ { 0, 1, . . . , 9 } do
            tok.val ← tok.val + s.advance( )
    return (tok)
end
```

Figure 2.6: Finding inum or fnum tokens for the ac language.

Pause

Implement Scanner based on RE by hand

1) Express the "lexical" grammar as RE

   (sometimes it is easier to start with a BNF or an EBNF and do necessary transformations)

- For each variant make a switch on the first character by peeking the input stream
- For each repetition (..)* make a while loop with the condition to keep going as long as peeking the input still yields an expected character
- Sometimes the "lexical" grammar is not reduced to one single RE but a small set of REs – in this case a switch or if-then-else case analysis is used to determine which rule is being recognized, before following the first two steps

Developing a Scanner

- Express the “lexical” grammar in EBNF

```
Token ::= Identifier | Integer-Literal | Operator |
        ; | : | := | ~ | ( | ) | eot
Identifier ::= Letter (Letter | Digit)*
Integer-Literal ::= Digit Digit*
Operator ::= + | - | * | / | < | > | =
Separator ::= Comment | space | eol
Comment ::= ! Graphic* eol
```

Now perform substitution and left factorization...

```
Token ::= Letter (Letter | Digit)*
        | Digit Digit*
        | + | - | * | / | < | > | =
        | ; | : (=|ε) | ~ | ( | ) | eot
Separator ::= ! Graphic* eol | space | eol
```

```
Token ::= Letter (Letter | Digit)*
        | Digit Digit*
        | + | - | * | / | < | > | =
        | ; | : (=|ε) | ~ | ( | ) | eot
```

```
private byte scanToken() {
  switch (currentChar) {
     case 'a': case 'b': ... case 'z':
     case 'A': case 'B': ... case 'Z':
        scan Letter (Letter | Digit)*
        return Token.IDENTIFIER;
     case '0': ... case '9':
        scan Digit Digit*
        return Token.INTLITERAL ;
     case '+': case '-': ... : case '=':
        takeIt();
        return Token.OPERATOR;
     ...etc...
}
```

Developing a Scanner

Let's look at the identifier case in more detail

```
  ...
  return ...
case 'a': case 'b': ... case 'z':
case 'A': case 'B': ... case 'Z':
  acceptIt();
  while (isLetter(currentChar)
        || isDigit(currentChar) )
    acceptIt();
  return Token.IDENTIFIER;
case '0': ... case '9':
  ...
```

Thus developing a scanner is a mechanical task.

Developing a Scanner

In Java the scanner will normally return instances of Token:

```
public class Token {
  byte kind; String spelling;
  final static byte
    IDENTIFIER = 0; INTLITERAL = 1; OPERATOR   = 2;
    BEGIN      = 3; CONST      = 4; ...
    ...

  public Token(byte kind, String spelling) {
    this.kind = kind; this.spelling = spelling;
    if spelling matches a keyword change my kind
    automatically
  }

  ...
}
```

Developing a Scanner

The scanner will return instances of Token:

```
public class Token {
...
   public Token(byte kind, String spelling) {
        if (kind == Token.IDENTIFIER) {
             int currentKind = firstReservedWord;
             boolean searching = true;
             while (searching) {
                       int comparison = tokenTable[currentKind].compareTo(spelling);
                       if (comparison == 0) {
                        this.kind = currentKind;
                       searching = false;
                       } else if (comparison > 0 || currentKind == lastReservedWord) {
                                 this.kind = Token.IDENTIFIER;
                                  searching = false;
                       } else {          currentKind ++;        }
              }
             } else
                       this.kind = kind;
 ...
```

Developing a Scanner

The scanner will return instances of Token:

```
public class Token {
...

        private static String[] tokenTable = new String[] {
        "<int>",    "<char>",    "<identifier>",    "<operator>",
        "array",    "begin",    "const",    "do",    "else",    "end",
        "func",    "if",    "in",    "let",    "of",    "proc",    "record",
        "then",    "type",    "var",    "while",
        ".",    ":",    ";",    ",",    ":=",    "~",    "(",    ")",    "[",    "]",    "{",    "}",    ""
        "<error>"  };

        private final static int firstReservedWord = Token.ARRAY,
                                 lastReservedWord  = Token.WHILE;

...
}
```

Developing a Scanner

Alternative implementation recognizing reserved words

```
  ...
  return ...
case 'i': acceptIt(); if (currentChar == 'f')  {acceptIt(); return Token.IF }
                      else if (currentChar == 'n')  {acceptIt(); return Token.IN }
...
case 'a': case 'b': ... case 'z':
case 'A': case 'B': ... case 'Z':
  acceptIt();
  while (isLetter(currentChar)
        || isDigit(currentChar) )
    acceptIt();
  return Token.IDENTIFIER;
case '0': ... case '9':
  ...
```

Thus developing a scanner is a mechanical task.

Developing a Scanner

- Developing a scanner by hand is relatively easy for simple token grammars
- But for complex token grammars it can be hard and error prone
- The task can be automated
- Programming scanner generator is an example of declarative programming
  - What to scan, not how to scan
- Most compilers are developed using a generated scanner
- But before we look at doing that, we need some theory!

FA and the implementation of Scanners

- Regular expressions, (N)DFA-ε and NDFA and DFA's are all equivalent formalism in terms of what languages can be defined with them.
- Regular expressions are a convenient notation for describing the "tokens" of programming languages.
- Regular expressions can be converted into FA's (the algorithm for conversion into NDFA-ε is straightforward)
- DFA's can be easily implemented as computer programs.

will explain this in subsequent slides

Generating Scanners

- Generation of scanners is based on
  - Regular Expressions: to describe the tokens to be recognized
  - Finite State Machines: an execution model to which RE's are "compiled"

Recap: Regular Expressions

ε	The empty string
t	Generates only the string t
$X\ Y$	Generates any string xy such that x is generated by x and y is generated by Y
$X \mid Y$	Generates any string which generated either by X or by Y
X^*	The concatenation of zero or more strings generated by X
(X)	For grouping

Generating Scanners

- Regular Expressions can be recognized by a finite state machine. (often used synonyms: finite automaton (acronym FA))

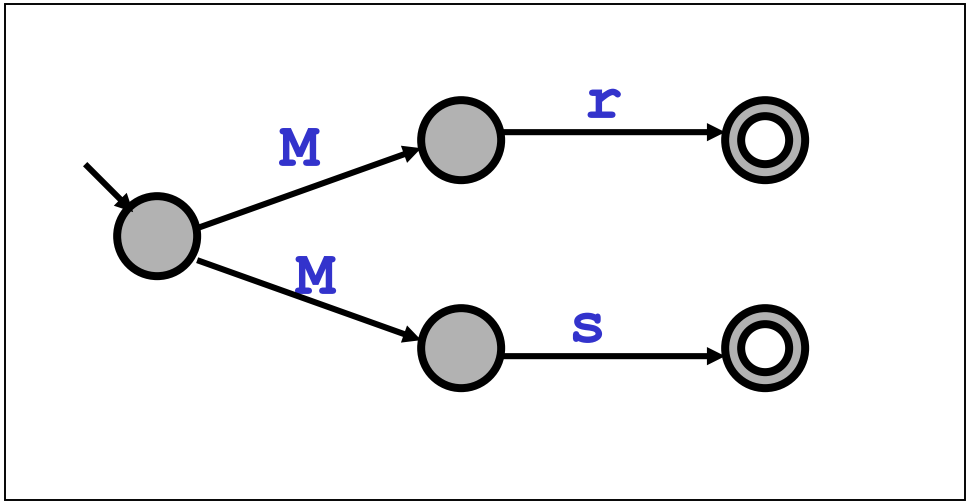
Definition: A finite state machine is an N-tuple (*States*, Σ, *start*, δ, *End*)

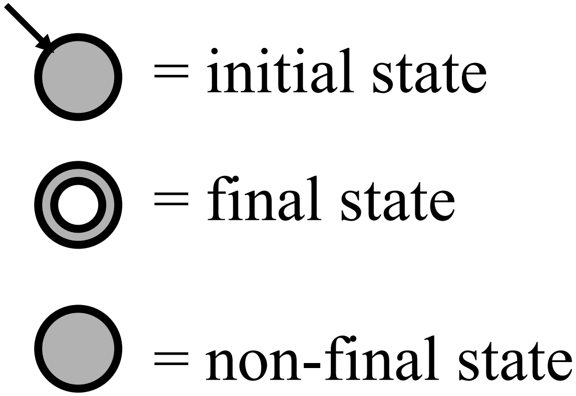
States	A finite set of "states"
Σ	An "alphabet": a finite set of symbols from which the strings we want to recognize are formed (for example: the ASCII char set)
start	A "start state" $Start \in States$
δ	Transition relation $\delta \subseteq States \times States \times \Sigma$. These are "arrows" between states labeled by a letter from the alphabet.
End	A set of final states. $End \subseteq States$

Generating Scanners

- Finite state machine: the easiest way to describe a Finite State Machine (FSM) is by means of a picture:

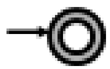
Example: an FA that recognizes `M r | M s`



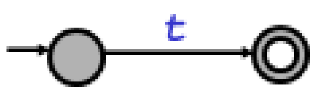


Converting a RE into an NDFA-ε

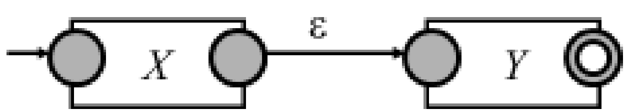
RE: ε
FA:



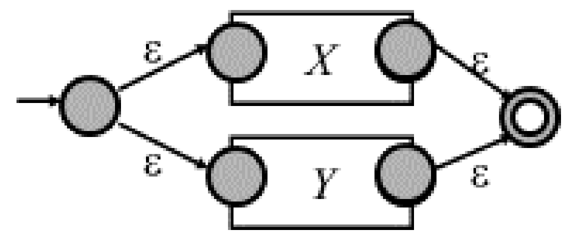
RE: t
FA:



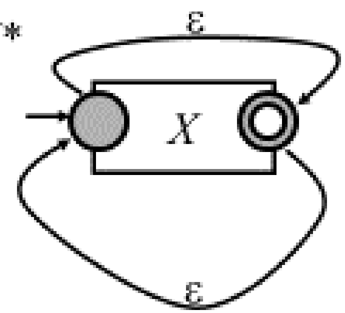
RE: *XY*
FA:



RE: *X*|*Y*
FA:



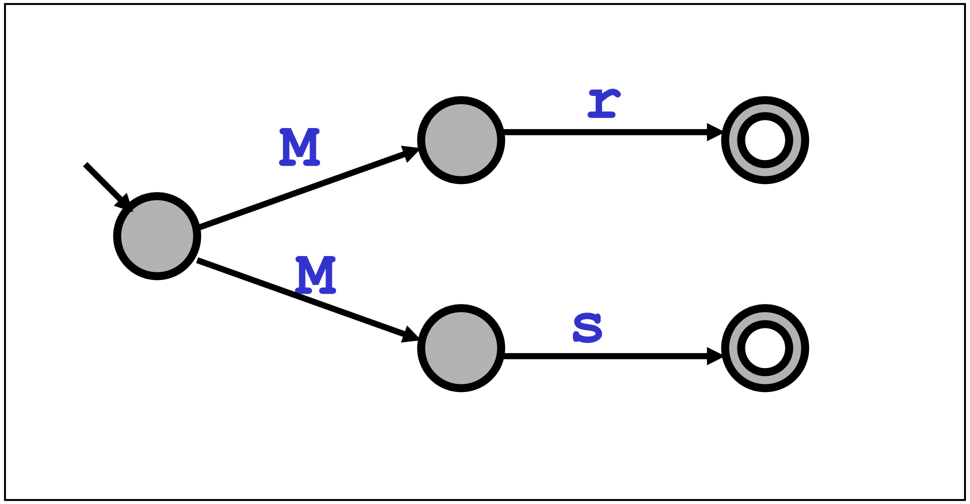
RE: *X**
FA:



Deterministic, and non-deterministic FA

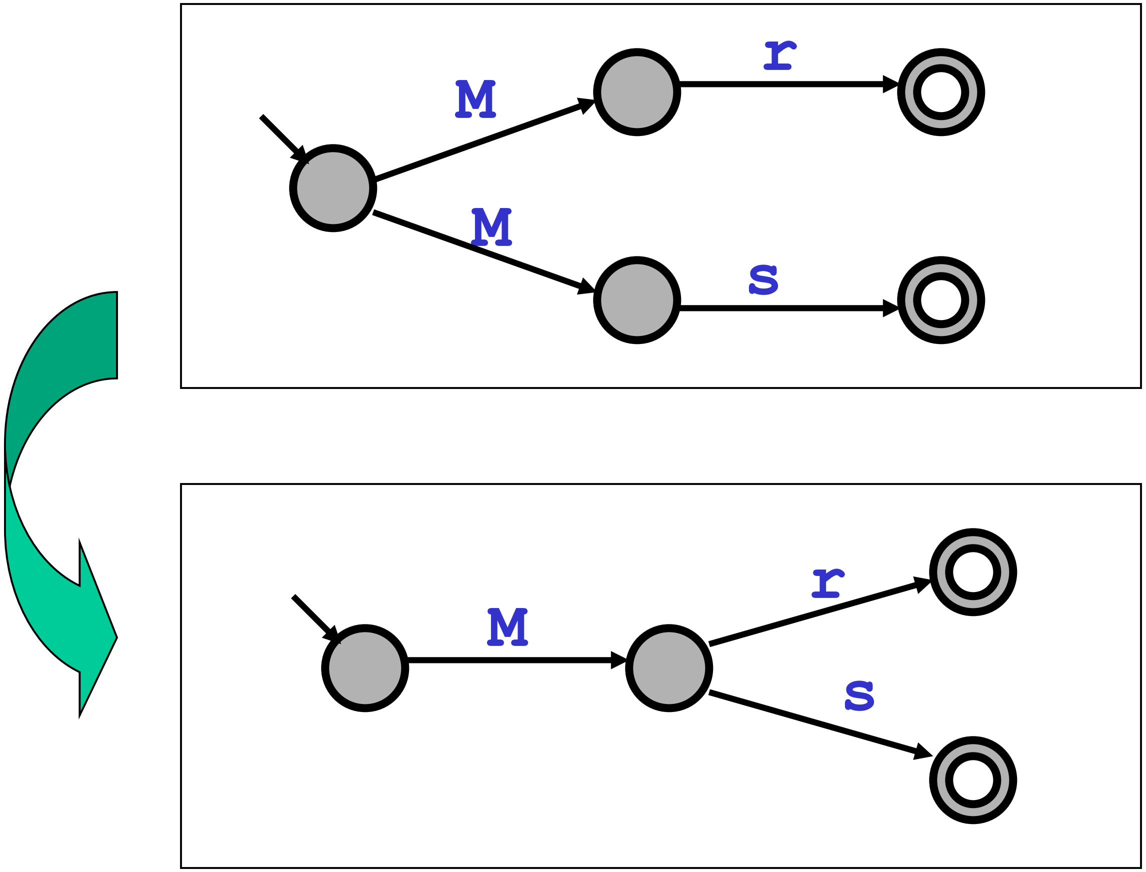
- An FA is called deterministic (acronym: DFA) if for every state and every possible input symbol, there is only one possible transition to choose from. Otherwise it is called non-deterministic (NDFA).

Q: Is this FSM deterministic or non-deterministic:



Deterministic, and non-deterministic FA

- Theorem: every NDFA can be converted into an equivalent DFA.



```
function MakeDeterministic(N) returns DFA
    D.StartState ← RecordState({ N.StartState })
    foreach S ∈ WorkList do
        WorkList ← WorkList − { S }
        foreach c ∈ Σ do  D.T(S,c) ← RecordState( ⋃_{s∈S} N.T(s,c))
    D.AcceptStates ← { S ∈ D.States | S ∩ N.AcceptStates ≠ ∅ }
end

function Close(S,T) returns Set
    ans ← S
    repeat
        changed ← false
        foreach s ∈ ans do
            foreach t ∈ T(s,λ) do
                if t ∉ ans
                then
                    ans ← ans ∪ { t }
                    changed ← true
    until not changed
    return (ans)
end

function RecordState(s) returns Set
    s ← Close(s, N.T)
    if s ∉ D.States
    then
        D.States ← D.States ∪ { s }
        WorkList ← WorkList ∪ { s }
    return (s)
end
```

Figure 3.23: Construction of a DFA D from an NFA N.

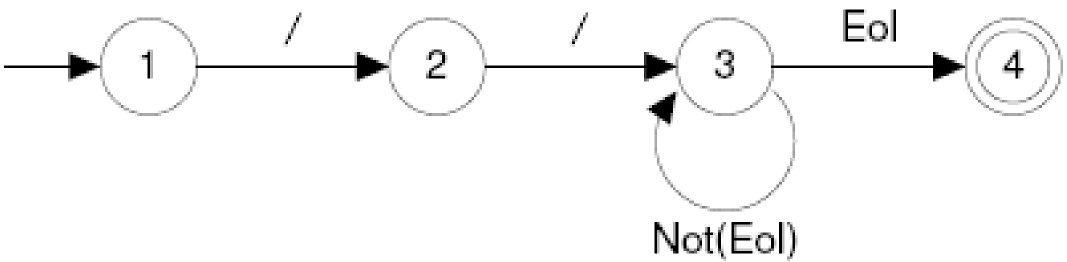
Implementing a DFA

Definition: A finite state machine is an N-tuple ($States, \Sigma, start, \delta, End$)

States	N different states => integers {0,..,N-1} => **`int`** data type
Σ	**`byte`** or **`char`** data type.
start	An integer number
δ	Transition relation $\delta \subseteq States \times \Sigma \times States$. For a DFA this is a function $States \times \Sigma \rightarrow States$ Represented by a two dimensional array (one dimension for the current state, another for the current character. The contents of the array is the next state.
End	A set of final states. Represented (for example) by an array of booleans (mark final state by true and other states by false)

Comment -> //(Not(Eol))*Eol

(a)



(b)

State	Character				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

Figure 3.2: DFA for recognizing a single-line comment. (a) transition diagram; (b) corresponding transition table.

```
/*    Assume CurrentChar contains the first character to be scanned    */
State ← StartState
while true do
    NextState ← T[State, CurrentChar]
    if NextState = error
    then break
    State ← NextState
    CurrentChar ← READ( )
if State ∈ AcceptingStates
then   /* Return or process the valid token */
else   /* Signal a lexical error */
```

Figure 3.3: Scanner driver interpreting a transition table.

Implementing a Scanner as a DFA

Slightly different from previously shown implementation (but similar in spirit):

- Not the goal to match entire input
  => when to stop matching?
  - Token(if), Token(Ident i) vs. Token(Ident ifi)

  Match longest possible token

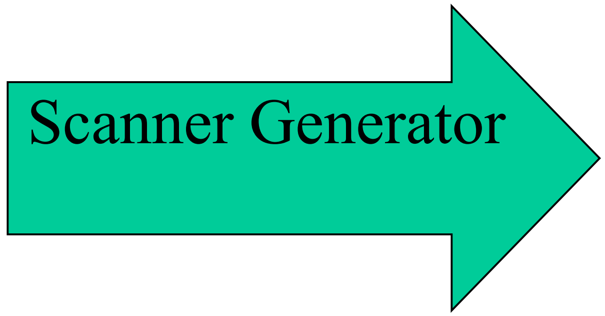
  Report error (and continue) when reaching error state.
- How to identify matched token class (not just true|false)

  Final state determines matched token class

FA and the implementation of Scanners

What a typical scanner generator does:

Token definitions
Regular expressions

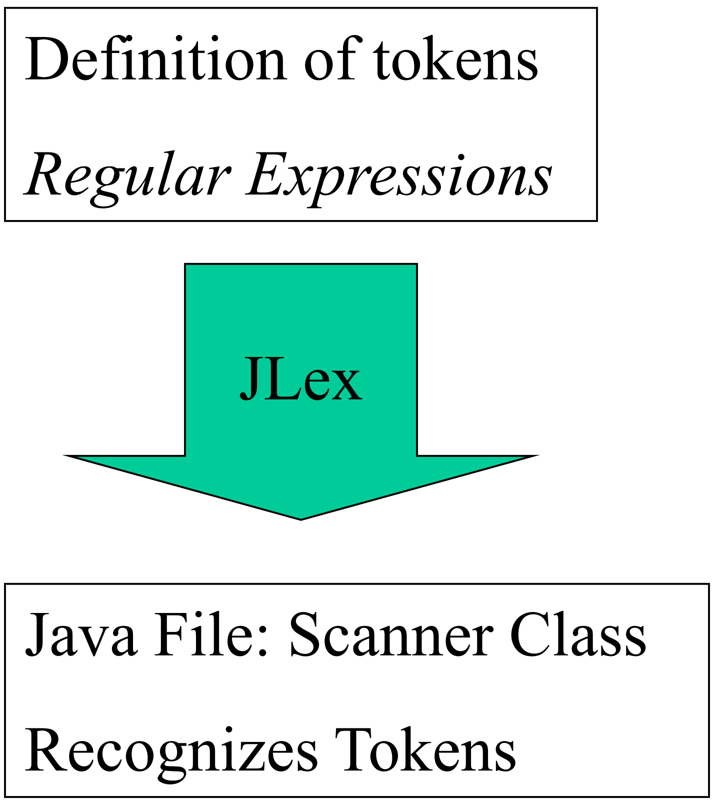


Scanner DFA
Java or C or ...

A possible algorithm:
- Convert RE into NDFA-ε
- Convert NDFA-ε into NDFA
- Convert NDFA into DFA
- generate Java/C/... code

note: In practice this exact algorithm is not used. For reasons of performance, sophisticated optimizations are used.
- direct conversion from RE to DFA
- minimizing the DFA

JLex Lexical Analyzer Generator for Java



Writing scanners is a rather "robotic" activity which can be automated.

We will look at an example JLex specification (adopted from the manual).

Consult the manual for details on how to write your own JLex specifications.

The JLex tool

Layout of JFLex file:

user code (added to start of generated file)

User code is copied directly into the output class

%%

options

JLex directives allow you to include code in the lexical analysis class, change names of various components, switch on character counting, line counting, manage EOF, etc.

%{

user code (added inside the scanner class declaration)

%}

macro definitions

Macro definitions gives names for useful regexps

%%

lexical declaration

Regular expression rules define the tokens to be recognised and actions to be taken

JLex Regular Expressions

- Regular expressions are expressed using ASCII characters (0 – 127) or UNICODE using the `%unicode` directive.
- The following characters are *metacharacters*.

```
? * + | ( ) ^ $ . [ ] { } " \
```

- Metacharacters have special meaning; they do not represent themselves.
- All other characters represent themselves.

JLex Regular Expressions

- Brackets `[ ]` match any single character listed within the brackets.
  - `[abc]` matches `a` or `b` or `c`.
  - `[A-Za-z]` matches any letter.
- If the first character after `[` is `^`, then the brackets match any character *except* those listed.
  - `[^A-Za-z]` matches any non-letter.
- Some escape sequences.
  - `\n` matches newline.
  - `\b` matches backspace.
  - `\r` matches carriage return.
  - `\t` matches tab.
  - `\f` matches formfeed.
- If `c` is not a special escape-sequence character, then `\c` matches `c`.

JLex Regular Expressions

- Let `r` and `s` be regular expressions.
- `r?` matches *zero or one* occurrences of `r`.
- `r*` matches *zero or more* occurrences of `r`.
- `r+` matches *one or more* occurrences of `r`.
- `r|s` matches `r` or `s`.
- `rs` matches `r` concatenated with `s`.

- Parentheses are used for grouping.

```
("+"|"-")?
```

- Regular expression beginning with `^` is matched only at the beginning of a line.
- Regular expression ending with `$` is matched only at the end of a line.
- The dot `.` matches any non-newline character.

```
%%
[a-eghj-oq-z]       { return(ID); }
%%
```

Figure 3.10: A Lex definition for ac's identifiers.

```
%%
(" ")+                            { /* delete blanks */}
f                                 { return(FLOATDCL); }
i                                 { return(INTDCL); }
p                                 { return(PRINT); }
[a-eghj-oq-z]                     { return(ID); }
([0-9]+)|([0-9]+"."[0-9]+)        { return(NUM);  }
"="                               { return(ASSIGN); }
"+"                               { return(PLUS); }
"-"                               { return(MINUS); }
%%
```

Figure 3.11: A Lex definition for ac's tokens.

```
%%
Blank                                   " "
Digits                                  [0-9]+
Non_f_i_p                               [a-eghj-oq-z]
%%
{Blank}+                                { /* delete blanks */}
f                                       { return(FLOATDCL); }
i                                       { return(INTDCL); }
p                                       { return(PRINT); }
{Non_f_i_p}                             { return(ID); }
{Digits}|({Digits}"."{Digits})          { return(NUM);  }
"="                                     { return(ASSIGN); }
"+"                                     { return(PLUS); }
"-"                                     { return(MINUS); }
%%
```

Figure 3.12: An alternative definition for ac's tokens.

Jlex for ac

```
package acFLEXCUP;

import java_cup.runtime.*;
import java.io.IOException;

import .AcLEXSym;
import static .AcLEXSym.*;

%%

%class AcLEXLex

%unicode
%line
%column

// %public
%final
// %abstract

%cupsym .AcLEXSym
%cup
// %cupdebug

%init{
    // TODO: code that goes to constructor
%init}

%{
    private Symbol sym(int type)
    {
        return sym(type, yytext());
    }

    private Symbol sym(int type, Object value)
    {
        return new Symbol(type, yyline, yycolumn, value);
    }

    private void error()
    throws IOException
    {
```

```
/* ANY          =   .
LineTerminator = \r | \n | \r\n
InputCharacter = [^\r\n]
WhiteSpace     = {LineTerminator} | [ \t\f]     /* The blank after the bracket is significant */


%%

/* {ANY}          {   return sym(ANY); }
[a-e] | [g-h] | [j-o] | [q-z] {return Symbol(Sym.ID);}
"f" {return Symbol.(Sym.FLTDCL);}
"i" {return Symbol.(Sym.INTDCL);}
"p" {return Symbol.(Sym.PRINT);}
"=" {return Symbol.(Sym.ASSIGN);}
"+" {return Symbol.(Sym.PLUS);}
"-" {return Symbol.(Sym.MINUS);}
([0-9])+ {return Symbol(Sym.INUM);}
([0-9])+"."([0-9])+ {return(Sym.FNUM);}

{WhiteSpace}                    { /* ignore */ }
```

JLex generated Lexical Analyser

- Class Yylex
  - Name can be changed with %class directive
  - Default construction with one arg – the input stream
    - You can add your own constructors
  - The method performing lexical analysis is yylex()
    - Public Yytoken yylex() which return the next token
    - You can change the name of yylex() with %function directive
  - String yytext() returns the matched token string
  - Int yylength() returns the length of the token
  - Int yychar is the index of the first matched char (if %char used)
- Class Yytoken
  - Returned by yylex() – you declare it or supply one already defined
  - You can supply one with %type directive
    - Java_cup.runtime.Symbol is useful
  - Actions typically written to return Yytoken(…)

Performance considerations

- Performance of scanners is important for production compilers, for example:
  - 30,000 lines per minute (500 lines per second)
  - 10,000 characters per second (for an average line of 20 characters)
  - For a processor that executes 10,000,000 instructions per second, 1,000 instructions per input character
  - Considering other tasks in compilers, 250 instructions per character is more realistic
- Size of scanner sometimes matters
  - Including keyword in scanner increases table size
    - E.g. Pascal has 35 keywords, including them increases states from 37 to 165
    - Uncompressed this increases table entries from 4699 to 20955
- Note modern scanners use explicit control, not table !
  - Why?

Other Scanner Generators

- Flex:
  - It produces scanners than are faster than the ones produced by Lex
  - Options that allow tuning of the scanner size vs. speed
- JFlex: in Java
- GLA: Generator for Lexical Analyzers
  - It produces a directly executable scanner in C
  - It's typically twice as fast as Flex, and it's competitive with the best hand-written scanners
- re2c
  - It produces directly executable scanners
- Alex, Lexgen, …
- Others are parts of complete suites of compiler development tools
  - JavaCC
  - Coco/R
  - SableCC
  - ANTLR

Conclusions

- Don't worry too much about DFAs
- You **do** need to understand how to specify regular expressions
- Note that different tools have different notations for regular expressions.
- You would probably only need to use Lex/Flex resp. Jlex/JFLex if you also use Yacc resp. CUP

- Sometimes it is easier to develop the scanner by hand transforming the RE into a case based direct scanner !
- In your project you can define the token grammar and implement a scanner by hand and/or by JFlex

Languages and Compilers (SProg og Oversættere)

Lecture 7
Top Down Parsing

Bent Thomsen

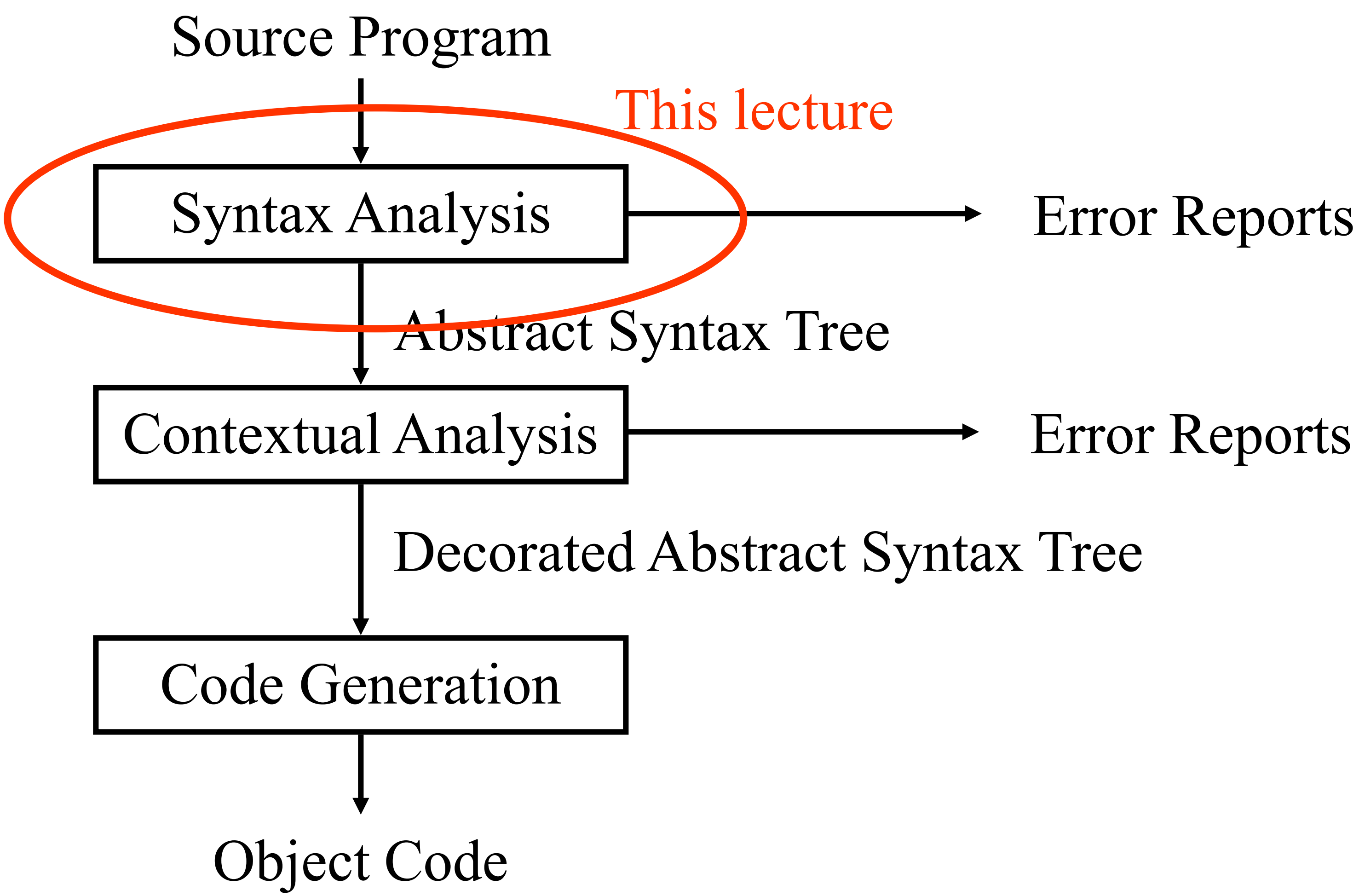
Department of Computer Science

Aalborg University

Learning goals

- To understand top down parsing
- To understand recursive decent parsers
- To understand the role of LL grammers
- To get an overview of table driven top down parsing
- To get an overview of top down parsing tools

The “Phases” of a Compiler



Syntax Analysis

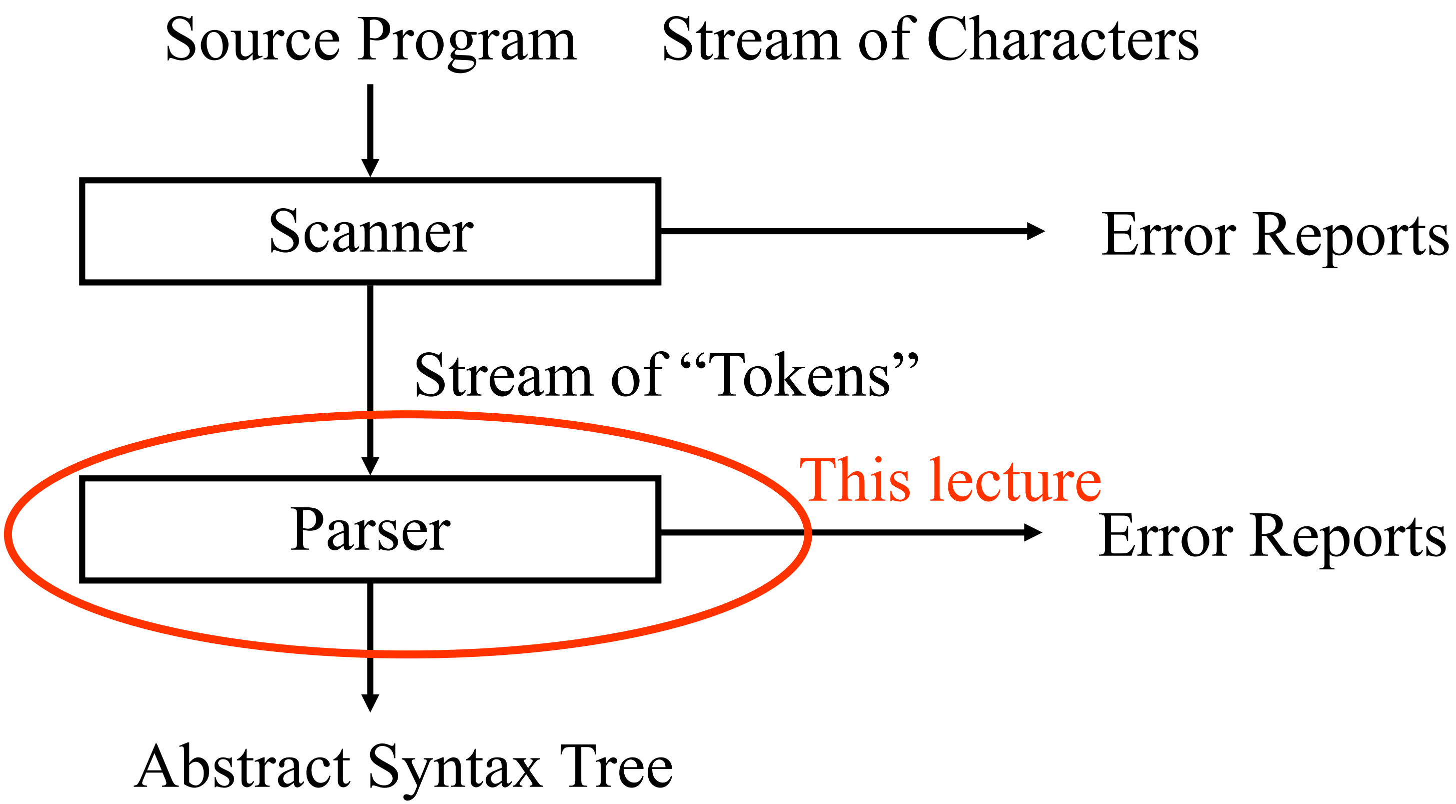
- The "job" of syntax analysis is to read the source text and determine its phrase structure.
- Subphases
  - Scanning
  - Parsing
  - Construct an internal representation of the source text that reifies the phrase structure (usually an AST)

Reify - To regard or treat (an abstraction) as if it had concrete or material existence

Note: A single-pass compiler usually does not construct an AST.

Syntax Analysis

Dataflow chart

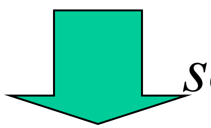


1) Scan: Divide Input into Tokens

An example ac source program:

```
f b
i a
a = 5
b = a + 3.2
p b
```

Lexems are “words” in the input, for example keywords, operators, identifiers, literals, etc.
Tokens is a datastructure for lexems and additional information



scanner

floatdl	*id*	*intdcl*	*id*	*id*	*assign*	*inum*	
f	b	i	a	a	=	5	…

	assign	*id*	*plus*	*fnum*	*print*	*id*	*eot*
…	=	a	+	3.2	p	b	

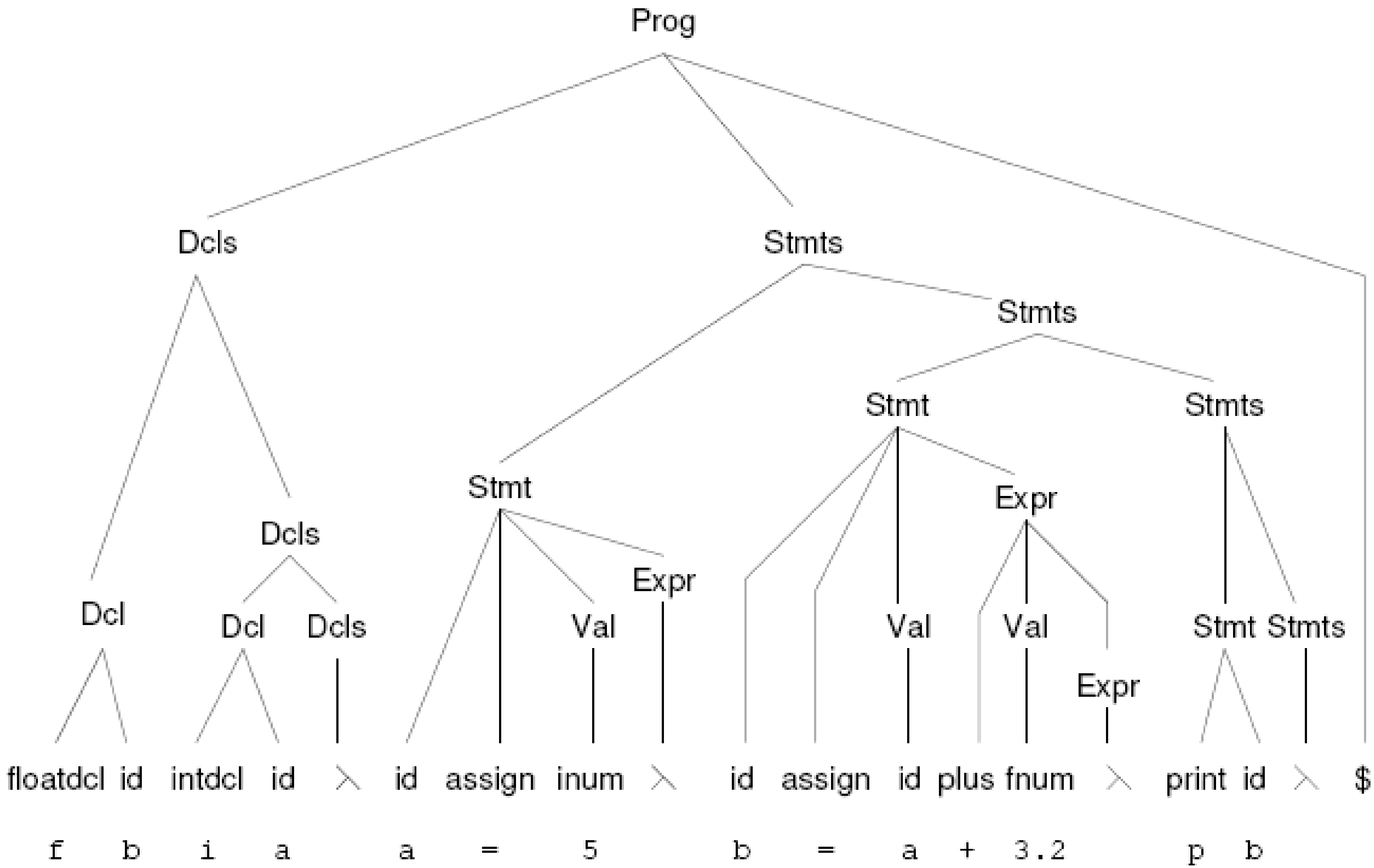


Figure 2.4: An ac program and its parse tree.

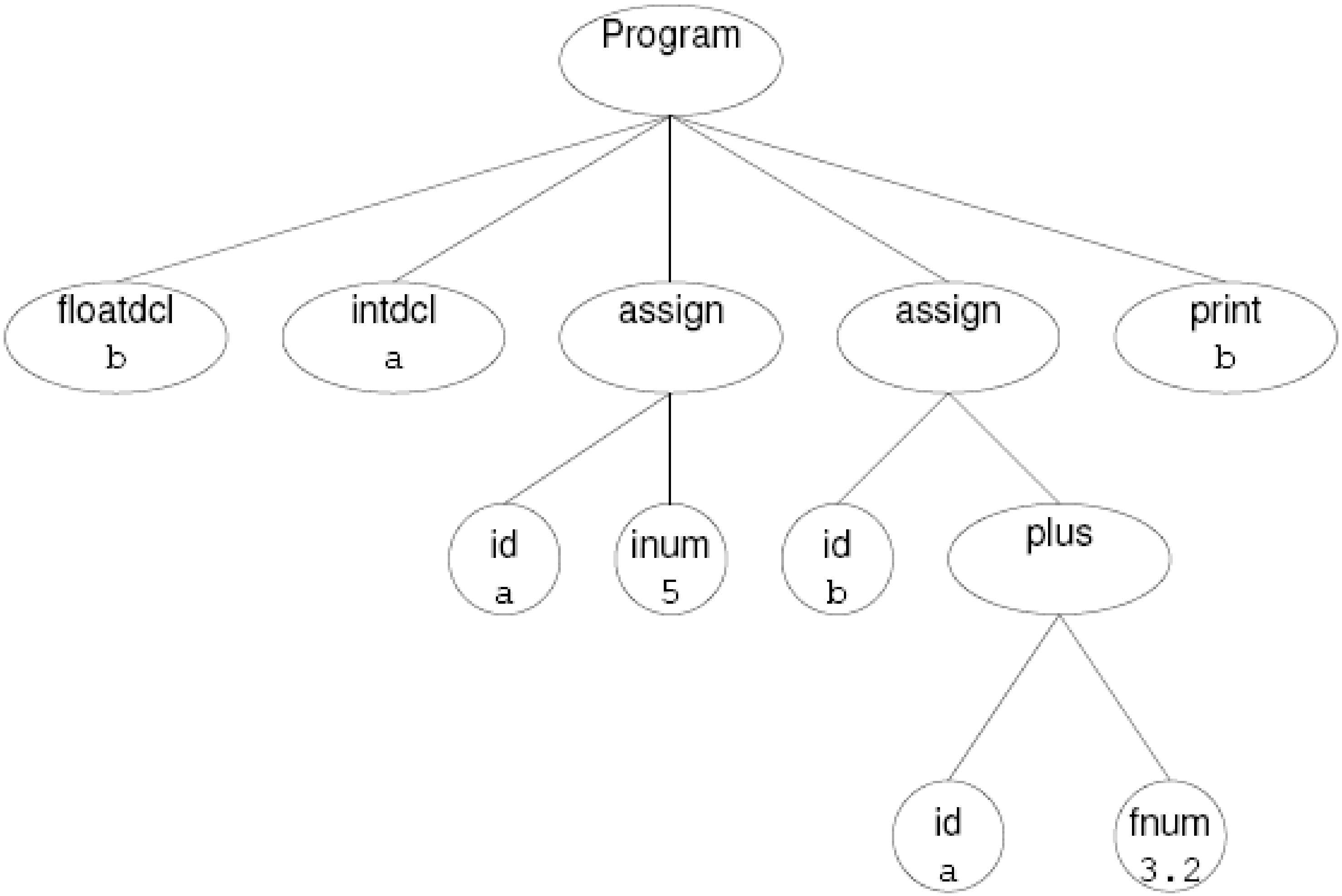
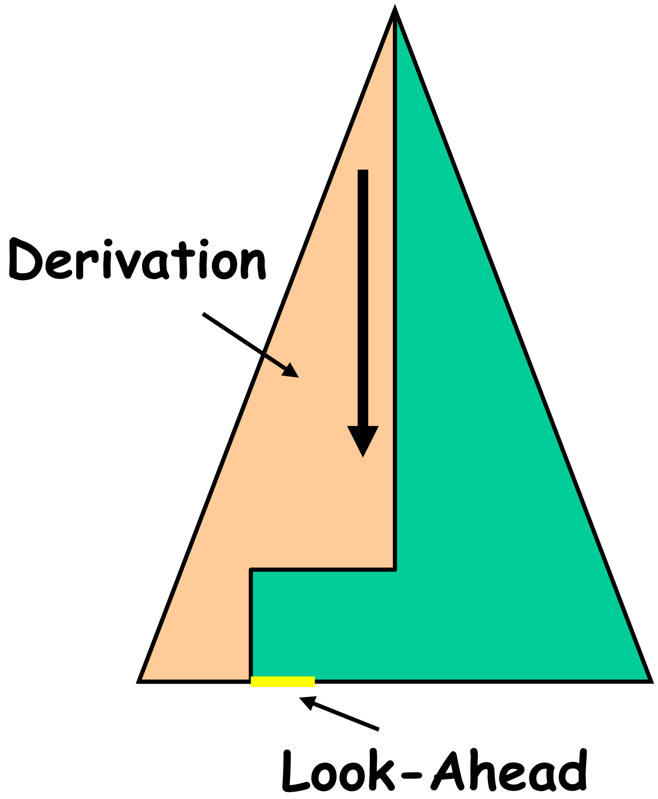


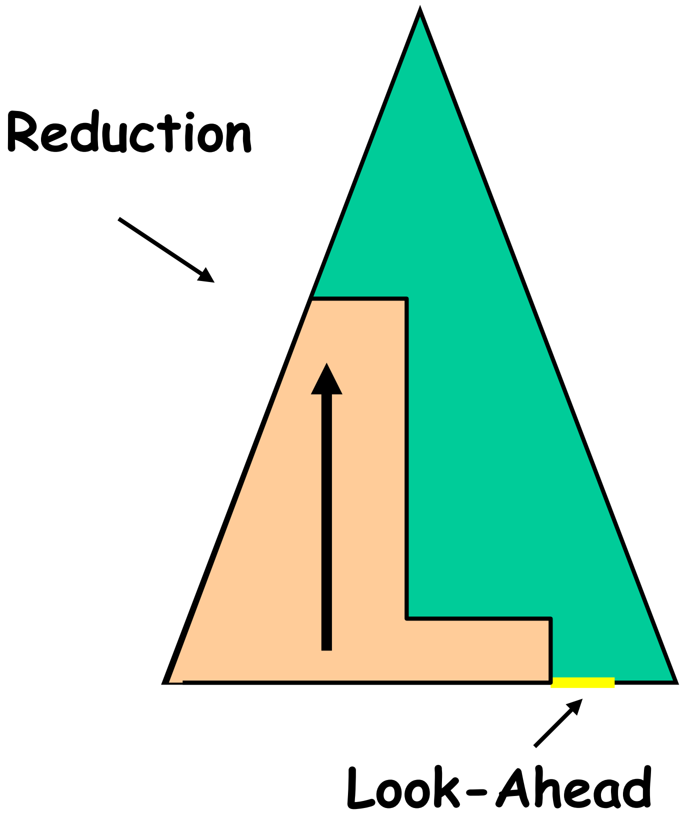
Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Top-Down vs. Bottom-Up parsing

LL-Analyse (Top-Down)
Left-to-Right Left Derivative



LR-Analyse (Bottom-Up)
Left-to-Right Right Derivative



Top Down Parsing Algorithms

- Example parsing of "Micro-English"

```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

The cat sees the rat.
The rat sees me.
I like a cat

The rat like me.
I see the rat.
I sees a rat.

Left derivations

```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

Sentence

→ Subject Verb Object .

→ The Noun Verb Object.

→ The cat Verb Object.

→ The cat sees Object.

→ The cat sees a Noun .

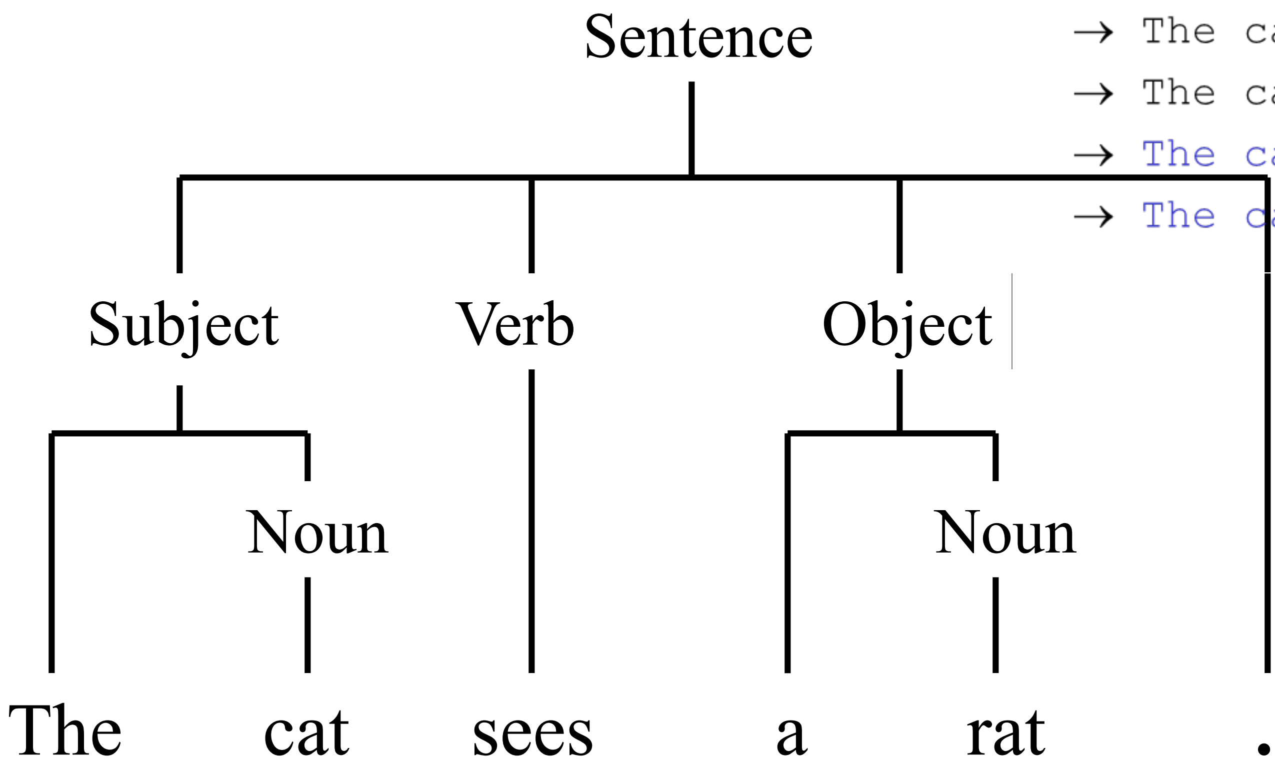
→ The cat sees a rat .

Top-down parsing

The parse tree is constructed starting at the top (root).

Corresponds to following left derivations

```
Sentence
→ Subject Verb Object .
→ The Noun Verb Object.
→ The cat Verb Object.
→ The cat sees Object.
→ The cat sees a Noun .
→ The cat sees a rat .
```



Recursive Descent Parsing

- Recursive descent parsing is a straightforward top-down parsing algorithm.
- We will now look at how to develop a recursive descent parser from an EBNF specification for a simple LL(1) grammar.
- Idea: the parse tree structure corresponds to the "call graph" structure of parsing procedures that call each other recursively.

Recursive Descent Parsing

```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

Define a procedure parseN for each non-terminal N

```
private void parseSentence() ;
private void parseSubject();
private void parseObject();
private void parseNoun();
private void parseVerb();
```

Recursive Descent Parsing: Auxiliary Methods

```
public class MicroEnglishParser {

   private TerminalSymbol currentTerminal

   private void accept(TerminalSymbol expected) {
      if (currentTerminal matches expected)
               currentTerminal = next input terminal ;
      else
               report a syntax error
   }

   ...
}
```

Recursive Descent Parsing: Parsing Methods

```
Sentence  ::= Subject Verb Object .
```

```
private void parseSentence() {
   parseSubject();
   parseVerb();
   parseObject();
   accept('.');
}
```

Recursive Descent Parsing: Parsing Methods

```
Subject    ::= I | a Noun | the Noun
```

```
private void parseSubject() {
   if (currentTerminal matches 'I')
      accept('I');
   else if (currentTerminal matches 'a') {
      accept('a');
      parseNoun();
   }
   else if (currentTerminal matches 'the') {
      accept('the');
      parseNoun();
   }
   else
     report a syntax error
}
```

Recursive Descent Parsing: Parsing Methods

```
Noun          ::= cat | mat | rat
```

```
private void parseNoun() {
    if (currentTerminal matches ‘cat’)
        accept(‘cat’);
    else if (currentTerminal matches ‘mat’)
        accept(‘mat’);
    else if (currentTerminal matches ‘rat’)
        accept(‘rat’);
    else
      report a syntax error
}
```

Algorithm to convert EBNF into a RD parser

- The conversion of an EBNF specification into a Java implementation for a recursive descent parser is so "mechanical" that it can easily be automated!

=> JavaCC and Coco/R does that in fact

- We can describe the algorithm by a set of mechanical rewrite rules

N ::= *X*

```
private void parseN() {
    parse X
}
```

Algorithm to convert EBNF into a RD parser

parse t where *t* is a terminal

```
accept(t);
```

parse N where *N* is a non-terminal

```
parseN();
```

parse ε

```
// a dummy statement
```

parse XY

parse X
parse Y

Algorithm to convert EBNF into a RD parser

*parse X**

```
while (currentToken.kind is in first[X]) {
   parse X
}
```

parse X|Y

```
switch (currentToken.kind) {
   cases in first[X]:
     parse X
     break;
   cases in first[Y]:
     parse Y
     break;
   default: report syntax error
}
```

Note: first[X] is sometimes called starters(X)

Systematic Development of RD Parser

(1) Express grammar in EBNF

(2) Grammar Transformations:

Left factorization and Left recursion elimination

(3) Create a parser class with

- private variable `currentToken`
- methods to call the scanner: `accept` and `acceptIt`

(4) Implement private parsing methods:

- add private `parse`***N*** method for each non terminal ***N***
- public `parse` method that
  - gets the first token form the scanner
  - calls `parse`***S*** (***S*** is the start symbol of the grammar)

Recursive Descent Parsing with AST

```
Sentence  ::= Subject Verb Object .
Subject   ::= I | a Noun | the Noun
Object    ::= me | a Noun | the Noun
Noun      ::= cat | mat | rat
Verb      ::= like | is | see | sees
```

Define a procedure parseN for each non-terminal N

```
private AST parseSentence() ;
private AST parseSubject();
private AST parseObject();
private AST parseNoun();
private AST parseVerb();
```

Recursive Descent Parsing: Parsing Methods

```
Sentence  ::= Subject Verb Object .
```

```
private AST parseSentence() {
   AST theAST;
   AST subject = parseSubject();
   AST verb = parseVerb();
   AST object = parseObject();
   accept('.');
   theAST = new Sentence(subject,verb,object);
   return theAST;
}
```

Converting EBNF into RD parsers

- The conversion of an EBNF specification into a Java implementation for a recursive descent parser is so “mechanical” that it can easily be automated!

=> JavaCC “Java Compiler Compiler”

JavaCC

- JavaCC is a parser generator
- JavaCC can be thought of as “Lex and Yacc” for implementing parsers in Java
- JavaCC is based on LL(k) grammars
- JavaCC transforms an EBNF grammar into an LL(k) parser
- The lookahead can be change by writing LOOKAHEAD(…)
- The JavaCC can have action code written in Java embedded in the grammar
- JavaCC has a companion called JJTree which can be used to generate an abstract syntax tree

JavaCC input format

- One file with extension .jj containing
  - Header
  - Token specifications
  - Grammar
- Example:

```
TOKEN:
{
	<INTEGER_LITERAL: ([“1”-”9”]([“0”-”9”])*|”0”)>
}

void StatementListReturn() :
{}
{
	(Statement())* “return” Expression() “;”
}
```

JavaCC token specifications use regular expressions

- Characters and strings must be quoted
  – “;”, “int”, “while”
- Character lists […] is shorthand for |
  – [“a”-”z”] matches “a” | “b” | “c” | … | “z”
  – [“a”,”e”,”i”,”o”,u”] matches any vowel
  – [“a”-”z”,”A”-”Z”] matches any letter
- Repetition shorthand with * and +
  – [“a”-”z”,”A”-”Z”]* matches zero or more letters
  – [“a”-”z”,”A”-”Z”]+ matches one or more letters
- Shorthand with ? provides for optionals:
  – (“+”|”-”)?[“0”-”9”]+ matches signed and unsigned integers
- Tokens can be named
  – TOKEN : {<IDENTIFIER:<LETTER>(<LETTER>|<DIGIT>)*>}
  – TOKEN : {<LETTER: [“a”-”z”,”A”-”Z”] >|<DIGIT:[“0”-”9”]>}
  – Now <IDENTIFIER> can be used in defining syntax

ac in BNF and EBNF

prog - > dcls stmts

dcls -> dcl dcls | epsilon

dcl -> floatdcl id

| intdcl id

stmts -> stmt stmts | epsilon

stmt - > id assign val expr

| print id

expr - > plus val expr

| minus val expr

| epsilon

val - > id | fnum | inum

prog - > dcl* stmt*

stmt - > id assign val expr?

| print id

expr - > plus val expr?

| minus val expr?

JavaCC Grammar for ac

```
SKIP :
{
  " "
| "\r"
| "\t"
| "\n"
}

TOKEN : /* OPERATORS */
{
  < PLUS : "+" >
| < MINUS : "-" >
| < FLOATDCL : "f" >
| < INTDCL : "i" >
| < PRINT : "p" >
| < ASSIGN : "=" >
}

TOKEN :
{
  < INUM : (< DIGIT >)+ >
| < FNUM : (< DIGIT >)+ (".") (< DIGIT >)+ >
| < #DIGIT : [ "0"-"9" ] >
| < ID : ["a"-"e"]|["g"-"h"]|["j"-"o"]|["q"-"z"] >
}
```

```
void prog() :
{}
{(dcl())+ (stmt())*
}

void dcl() :
{}
{
  < FLOATDCL > <ID > | < INTDCL > <ID >
}

void stmt() :
{}
{
  < ID ><ASSIGN > val() (expr())?
| < PRINT > <ID >
}

void val() :
{}
{
  < INUM > | < FNUM > | <  ID >
}

void expr() :
{}
{
    < PLUS > val() (expr())?
  | < MINUS > val() (expr())?
}
```

Adding AST actions for ac

```
AST prog() :
{Prog itsAST = new Prog(new ArrayList<AST >());
 AST dcl;
 AST stm;
}
{(
  dcl = dcl()
  {itsAST.prog.add(dcl);}
  )+
  (stm = stmt()
  {itsAST.prog.add(stm);}
  )*
  {return itsAST;}

}
```

```
AST dcl() :
{Token t;}
{
  (< FLOATDCL > t = <ID >)
  {return new FloatDcl(t.image);}
 | (< INTDCL > t = <ID >)
  {return new IntDcl(t.image);}
}

AST stmt() :
{Boolean b = true;
 AST v;
 Computing e = null;
 Token t;
}
{
  (t = < ID ><ASSIGN > v = val() ((e = expr()){b = false;})?)
  {if (b) return v; else { e.child1 = v; return e;}}
| (< PRINT > t = <ID >)
  {return new Printing(t.image);}
}
```

Generating a parser with JavaCC

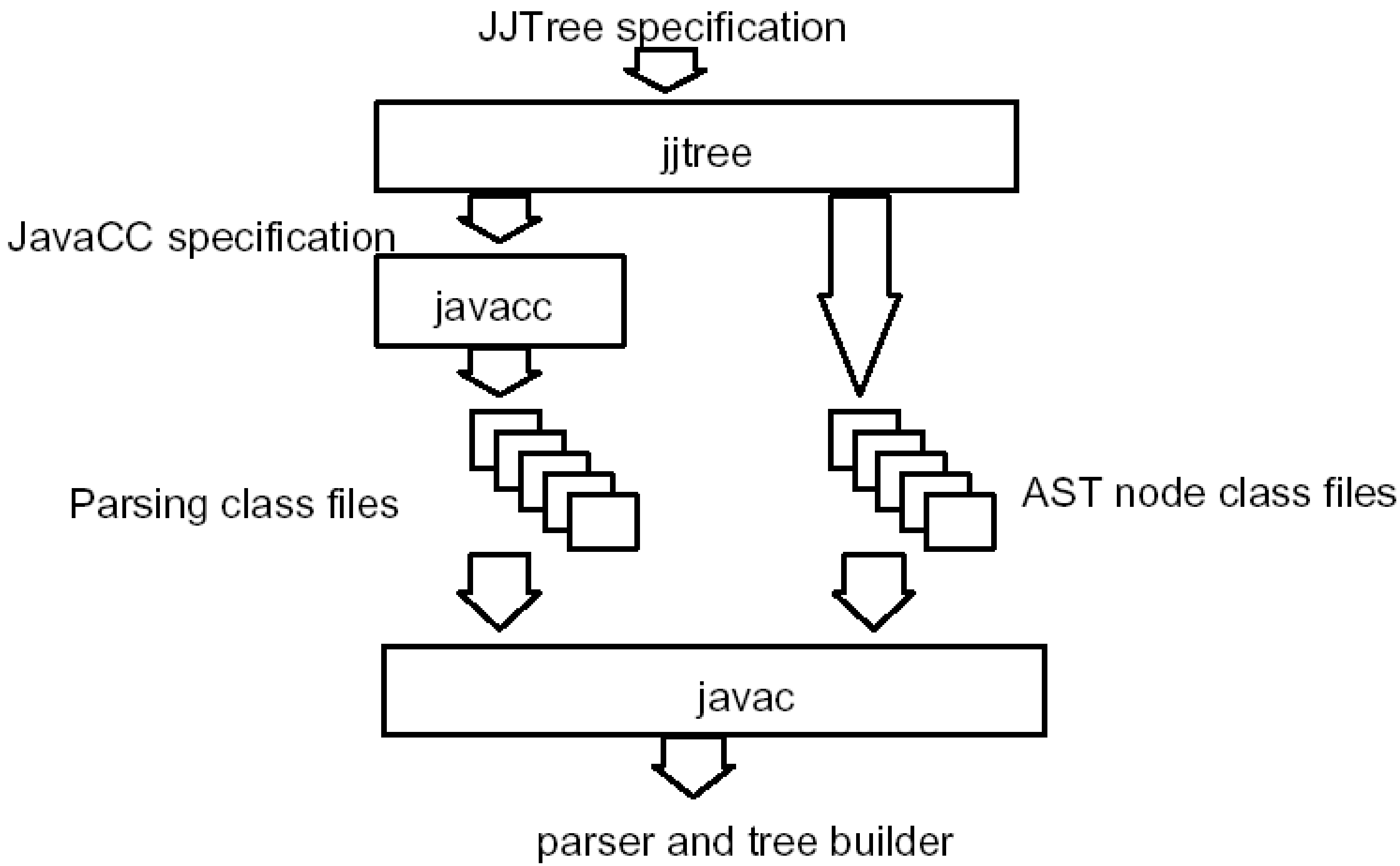
- javacc *filename.jj*
  - generates a parser with specified name
  - Lots of .java files
- javac *.java
  - Compile all the .java files
- There is a plug-in for eclipse

- Note the parser doesn't do anything on its own.
- You have to either
  - Add actions to grammar by hand
  - Use JJTree to generate actions for building AST
  - Use JBT to generate AST and visitors

JavaCC and JJTree

- JavaCC is a parser generator
  - Inputs a set of token definitions, grammar and actions
  - Outputs a Java program which performs syntatic analysis
    - Finding tokens
    - Parses the tokens according to the grammar
    - Executes actions
- JJTree is a preprocessor for JavaCC
  - Inputs a grammar file
  - Inserts tree building actions
  - Outputs JavaCC grammar file with actions
- From this you can add code to traverse the tree to do static analysis, code generation or interpretation.

JavaCC and JJTree



Using JJTree

- JJTree is a preprocessor for JavaCC
- JTree transforms a bare JavaCC grammar into a grammar with embedded Java code for building an AST
  - Classes Node and SimpleNode are generated
  - Can also generate classes for each type of node
- All AST nodes implement interface Node
  - Useful methods provided include:
    - `Public void jjtGetNumChildren()` - returns the number of children
    - `Public void jjtGetChild(int i)` - returns the i'th child
  - The "state" is in a parser field called jjtree
    - The root is at `Node rootNode()`
    - You can display the tree with
    - `((SimpleNode)parser.jjtree.rootNode()).dump(" ");`
- JJTree supports the building of abstract syntax trees which can be traversed using the visitor design pattern

JBT

- JBT – Java Tree Builder is an alternative to JJTree
- It takes a plain JavaCC grammar file as input and automatically generates the following:
  - A set of syntax tree classes based on the productions in the grammar, utilizing the Visitor design pattern.
  - Two interfaces: Visitor and ObjectVisitor. Two depth-first visitors: DepthFirstVisitor and ObjectDepthFirst, whose default methods simply visit the children of the current node.
  - A JavaCC grammar with the proper annotations to build the syntax tree during parsing.
- New visitors, which subclass DepthFirstVisitor or ObjectDepthFirst, can then override the default methods and perform various operations on and manipulate the generated syntax tree.

The Visitor Pattern

For object-oriented programming the *visitor pattern* enables the definition of a *new operator* on an *object structure* without *changing the classes* of the objects

When using visitor pattern

- The set of classes must be fixed in advance
- Each class must have an accept method
- Each accept method takes a visitor as argument
- The purpose of the accept method is to invoke the visitor which can handle the current object.
- A visitor contains a visit method for each class (overloading)
- A method for class C takes an argument of type C

- The advantage of Visitors: New methods without recompilation!

Pause

LL(1) Grammars

- The presented algorithm to convert EBNF into a parser does not work for all possible grammars.
- It only works for so called simple LL(1) grammars.
- What grammars are LL(1)?
- Basically, an **LL(1) grammar** is a grammar which can be parsed with a **top-down parser** with a **lookahead** (in the input stream of tokens) of **one token**.

How can we recognize that a grammar is (or is not) LL(1)?

⇒There is a formal definition

⇒We can deduce the necessary conditions from the parser generation algorithm.

Formal definition of LL(1)

A grammar G is LL(1) iff
for each set of productions $X ::= X_1 \mid X_2 \mid ... \mid X_n$:

1. *first*[X_1], *first*[X_2], …, *first*[X_n] are all pairwise disjoint
2. *If* X_i =>* ε then *first*[X_j]∩ *follow*[X]=Ø, for $1 \leq j \leq n . i \neq j$

If G is ε-free then 1 is sufficient

NOTE: first[X_1] is sometimes called *starters*[X_1]

first[X] = {t in Terminals | X =>* t β }
Follow[X] = {t in Terminals | S =>+ α X t β }

LL(1) Grammars

*parse X**

```
while (currentToken.kind is in first[X]) {
   parse X
}
```

Condition: *first*[*X*] must be disjoint from the set of tokens that can immediately follow *X* *

parse X|Y

```
switch (currentToken.kind) {
   cases in first[X]:
     parse X
     break;
   cases in first[Y]:
     parse Y
     break;
   default: report syntax error
}
```

Condition: *first*[*X*] and *first*[*Y*] must be disjoint sets.

First Sets

Informal Definition:

The starter set of a RE *X* is the set of terminal symbols that can occur as the start of any string generated by *X*

Example :

first[`(+|-|ε)(0|1|...|9)*`] = {+,-, 0,1,...,9}

Formal Definition:

first[ε] ={}

first[*t*] ={*t*} (where *t* is a terminal symbol)

first[*X Y*] = *first*[*X*] ∪ *first*[*Y*] (if *X* generates ε)

first[*X Y*] = *first*[*X*] (if not *X* generates ε)

first[*X* | *Y*] = *first*[*X*] ∪ *first*[*Y*]

first[*X**] = *first*[*X*]

´First Sets (ctd)

Informal Definition:

The starter set of RE can be generalized to extended BNF

Formal Definition:

$first[N] = first[X]$ (for production rules N ::= X)

Example :

$$\begin{aligned} first[\text{Expression}] &= first[\text{PrimaryExp (Operator PrimaryExp)*}] \\ &= first[\text{PrimaryExp}] \\ &= first[\text{Identifiers}] \cup first[\textbf{(}\text{Expression}\textbf{)}] \\ &= first[\textbf{a} \mid \textbf{b} \mid \textbf{c} \mid \ldots \mid \textbf{z}] \cup \{\textbf{(}\} \\ &= \{\textbf{a}, \textbf{b}, \textbf{c}, \ldots, \textbf{z}, \textbf{(}\} \end{aligned}$$

```
function First(α) returns Set
    foreach A ∈ NonTerminals( ) do VisitedFirst(A) ← false        (9)
    ans ← InternalFirst(α)
    return (ans)
end
function InternalFirst(𝒳β) returns Set
    if 𝒳β = ⊥                                                     (10)
    then return (∅)
    if 𝒳 ∈ Σ                                                      (11)
    then return ({ 𝒳 })
    /*  𝒳 is a nonterminal.                                     */ (12)
    ans ← ∅
    if not VisitedFirst(𝒳)
    then
        VisitedFirst(𝒳) ← true                                    (13)
        foreach rhs ∈ ProductionsFor(𝒳) do
            ans ← ans ∪ InternalFirst(rhs)                        (14)
    if SymbolDerivesEmpty(𝒳)                                      (15)
    then ans ← ans ∪ InternalFirst(β)
    return (ans)                                                  (16)
end
```

Figure 4.8: Algorithm for computing First(α).

```
function Follow(A) returns Set
    foreach A ∈ NonTerminals( ) do
        VisitedFollow(A) ← false                                      (17)
    ans ← InternalFollow(A)
    return (ans)
end
function InternalFollow(A) returns Set
    ans ← ∅
    if not VisitedFolow(A)                                            (18)
    then
        VisitedFollow(A) ← true                                       (19)
        foreach a ∈ Occurrences(A) do                                 (20)
            ans ← ans ∪ First(Tail(a))                                (21)
            if AllDeriveEmpty(Tail(a))                                (22)
            then
                targ ← LHS(Production(a))
                ans ← ans ∪ InternalFollow(targ)                      (23)
    return (ans)                                                      (24)
end
function AllDeriveEmpty(γ) returns Boolean
    foreach X ∈ γ do
        if not SymbolDerivesEmpty(X) or X ∈ Σ
        then return (false)
    return (true)
end
```

Figure 4.11: Algorithm for computing Follow(A).

A variant on First and Follow sets

Rules for First Sets

1. If X is a terminal **then** First(X) is just X!
2. If there is a Production X → ε **then** add ε to first(X)
3. If there is a Production X → Y1Y2..Yk **then** add first(Y1Y2..Yk) to first(X)
4. First(Y1Y2..Yk) is **either**
   1. First(Y1) (if First(Y1) doesn't contain ε)
   2. **OR** (if First(Y1) does contain ε) then First (Y1Y2..Yk) is everything in First(Y1) <except for ε > as well as everything in First(Y2..Yk)
   3. If First(Y1) First(Y2)..First(Yk) all contain ε **then** add ε to First(Y1Y2..Yk) as well.

Rules for Follow Sets

1. First put $ (the end of input marker) in Follow(S) (S is the start symbol)
2. If there is a production A → aBb, (where a can be a whole string) **then** everything in FIRST(b) except for ε is placed in FOLLOW(B).
3. If there is a production A → aB, **then** everything in FOLLOW(A) is in FOLLOW(B)
4. If there is a production A → aBb, where FIRST(b) contains ε, **then** everything in FOLLOW(A) is in FOLLOW(B)

Source: https://www.jambe.co.nz/UNI/FirstAndFollowSets.html

First and Follow in KfG Edit

```
S -> A C
C -> c
   | EPSILON
A -> a B C d
   | B Q
B -> b B
   | EPSILON
Q -> q
   | EPSILON
```

LL(1) first condition fulfilled!

```
FIRST (S) = {a, b, EPSILON, q, c}
FOLLOW(S) = {$}
FIRST (S) ∩ FOLLOW(S) = ∅
```

```
FIRST (C) = {c, EPSILON}
FOLLOW(C) = {$, d}
FIRST (C) ∩ FOLLOW(C) = ∅
```

```
FIRST (A) = {a, b, EPSILON, q}
FOLLOW(A) = {$, c}
FIRST (A) ∩ FOLLOW(A) = ∅
```

```
FIRST (B) = {b, EPSILON}
FOLLOW(B) = {c, d, $, q}
FIRST (B) ∩ FOLLOW(B) = ∅
```

```
FIRST (Q) = {q, EPSILON}
FOLLOW(Q) = {$, c}
FIRST (Q) ∩ FOLLOW(Q) = ∅
```

LL(1) second condition fulfilled!

```
function IsLL1(G) returns Boolean
    foreach A ∈ N do
        PredictSet ← ∅
        foreach p ∈ ProductionsFor(A) do
            if Predict(p) ∩ PredictSet ≠ ∅                              (4)
            then return (false)
            PredictSet ← PredictSet ∪ Predict(p)
    return (true)
end
```

Figure 5.4: Algorithm to determine if a grammar G is LL(1).

```
function Predict(p : A → X1 ... Xm) : Set
    ans ← First(X1 ... Xm)                                          (1)
    if RuleDerivesEmpty(p)                                          (2)
    then
        ans ← ans ∪ Follow(A)                                       (3)
    return (ans)
end
```

Figure 5.1: Computation of Predict sets.

1	S	$\rightarrow$	A C $
2	C	$\rightarrow$	c
3		|	λ
4	A	$\rightarrow$	a B C d
5		|	B Q
6	B	$\rightarrow$	b B
7		|	λ
8	Q	$\rightarrow$	q
9		|	λ

Figure 5.2: A CFGs.

Rule Number	A	$X_1 \ldots X_m$	First($X_1 \ldots X_m$)	Derives Empty?	Follow(A)	Answer
1	S	A C $	a,b,q,c,$	No		a,b,q,c,$
2	C	c	c	No		c
3		λ		Yes	d,$	d,$
4	A	a B C d	a	No		a
5		B Q	b,q	Yes	c,$	b,q,c,$
6	B	b B	b	No		b
7		λ		Yes	q,c,d,$	q,c,d,$
8	Q	q	q	No		q
9		λ		Yes	c,$	c,$

Figure 5.3: Predict calculation for the grammar of Figure 5.2.

1 S → A C $
2 C → c
3 | λ
4 A → a B C d
5 | B Q
6 B → b B
7 | λ
8 Q → q
9 | λ

```
procedure A(ts)
    switch (. . . )
        case ts.PEEK( ) ∈ Predict(p_1)
            /*  Code for p_1                                  */
        case ts.PEEK( ) ∈ Predict(p_i)
            /*  Code for p_2                                  */
        /*  .                                                 */
        /*  .                                                 */
        /*  .                                                 */
        case ts.PEEK( ) ∈ Predict(p_n)
            /*  Code for p_n                                  */
        case default
            /*  Syntax error                                  */
end
```

Figure 5.6: A typical recursive-descent procedure. Successful LL(1) analysis ensures that only one of the **case** predicates is **true**.

```
procedure S( )
    switch (...)
        case ts.PEEK( ) ∈ { a, b, q, c, $ }
            call A( )
            call C( )
            call MATCH( $ )
end
procedure C( )
    switch (...)
        case ts.PEEK( ) ∈ { c }
            call MATCH( c )
        case ts.PEEK( ) ∈ { d, $ }
            return ()
end
procedure A( )
    switch (...)
        case ts.PEEK( ) ∈ { a }
            call MATCH( a )
            call B( )
            call C( )
            call MATCH( d )
        case ts.PEEK( ) ∈ { b, q, c, $ }
            call B( )
            call Q( )
end
procedure B( )
    switch (...)
        case ts.PEEK( ) ∈ { b }
            call MATCH( b )
            call B( )
        case ts.PEEK( ) ∈ { q, c, d, $ }
            return ()
end
procedure Q( )
    switch (...)
        case ts.PEEK( ) ∈ { q }
            call MATCH( q )
        case ts.PEEK( ) ∈ { c, $ }
            return ()
end
```

Figure 5.7: Recursive-descent code for the grammar shown in Figure 5.2. The variable *ts* denotes the token stream produced by the scanner.

1	S	→	A C $
2	C	→	c
3		|	λ
4	A	→	a B C d
5		|	B Q
6	B	→	b B
7		|	λ
8	Q	→	q
9		|	λ

```
procedure MATCH( ts, token )
    if ts.PEEK( ) = token
    then call ts.ADVANCE( )
    else call ERROR( Expected token )
end
```

Figure 5.5: Utility for matching tokens in an input stream.

Recursive Decent Parser for ac

```java
 * Recursive-descent parser based on the grammar given
 *   in Figure 2.1
 * @author cytron
 *
 */
public class Parser {

    private TokenStream ts;

    public Parser(CharStream s) {
        ts = new TokenStream(s);
    }


    public void Prog() {
        if (ts.peek() == FLTDCL || ts.peek() == INTDCL || ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
            Dcls();
            Stmts();
            expect(EOF);
        }
        else error("expected floatdcl, intdcl, id, print, or eof");
    }

    public void Dcls() {
        if (ts.peek() == FLTDCL || ts.peek() == INTDCL) {
            Dcl();
            Dcls();
        }
        else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
            // Do nothing for lambda-production
        }
        else error("expected floatdcl, intdcl, id, print, or eof");
    }

    public void Dcl() {
        if (ts.peek() == FLTDCL) {
            expect(FLTDCL);
            expect(ID);
        }
        else if (ts.peek() == INTDCL) {
            expect(INTDCL);
            expect(ID);
        }
        else error("expected float or int declaration");
    }
```

prog - > dcls stmts

dcls -> dcl dcls | epsilon

dcl -> floatdcl id

| intdcl id

stmts -> stmt stmts | epsilon

stmt - > id assign val expr

| print id

expr - > plus val expr

| minus val expr

| epsilon

val - > id | fnum | inum

Recursive Decent Parser for ac

```java
/**
 * Figure 2.7 code
 */
public void Stmts() {
    if (ts.peek() == ID || ts.peek() == PRINT) {
        Stmt();
        Stmts();
    }
    else if (ts.peek() == EOF) {
        // Do nothing for lambda-production
    }
    else error("expected id, print, or eof");

}

public void Stmt() {
    if (ts.peek() == ID) {
        expect(ID);
        expect(ASSIGN);
        Val();
        Expr();
    }
    else if (ts.peek() == PRINT) {
        expect(PRINT);
        expect(ID);
    }
    else error("expected id or print");

}

public void Expr() {
    if (ts.peek() == PLUS) {
        expect(PLUS);
        Val();
        Expr();
    }
    else if (ts.peek() == MINUS) {
        expect(MINUS);
        Val();
        Expr();

    }
    else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
        // Do nothing for lambda-production
    }
    else error("expected plus, minus, id, print, or eof");

}
```


```java
public void Expr() {
    if (ts.peek() == PLUS) {
        expect(PLUS);
        Val();
        Expr();
    }
    else if (ts.peek() == MINUS) {
        expect(MINUS);
        Val();
        Expr();

    }
    else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
        // Do nothing for lambda-production
    }
    else error("expected plus, minus, id, print, or eof");

}
```


```java
public void Val() {
    if (ts.peek() == ID) {
        expect(ID);
    }
    else if (ts.peek() == INUM) {
        expect(INUM);
    }
    else if (ts.peek() == FNUM) {
        expect(FNUM);
    }
    else error("expected id, inum, or fnum");

}

private void expect(int type) {
    Token t = ts.advance();
    if (t.type != type) {
        throw new Error("Expected type "
                + Token.token2str[type]
                            + " but received type "
                            + Token.token2str[t.type]);

    }
}

private void error(String message) {
    throw new Error(message);
}
}
```

stmts -> stmt stmts | epsilon
stmt - > id assign val expr
| print id
expr - > plus val expr
| minus val expr
| epsilon
val - > id | fnum | inum

Recursive Decent Parser for ac with AST

```java

public class ASTParser {
    private TokenStream ts;

    public ASTParser(CharStream s) {
        ts = new TokenStream(s);
    }


    public AST Prog() {
        Prog itsAST = new Prog(new ArrayList<AST>());
        if (ts.peek() == FLTDCL || ts.peek() == INTDCL || ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
            ArrayList<AST> dcllist = Dcls();
            ArrayList<AST> stmlist = Stmts();
            expect(EOF);
            if (dcllist != null) itsAST.prog.addAll(dcllist);
            if (stmlist != null) itsAST.prog.addAll(stmlist);
        }
        else error("expected floatdcl, intdcl, id, print, or eof");
        return itsAST;
    }

    public ArrayList<AST> Dcls() {
        ArrayList<AST> astlist = new ArrayList<AST>();
        if (ts.peek() == FLTDCL || ts.peek() == INTDCL) {
            AST dcl = Dcl();
            ArrayList<AST> dcls = Dcls();
            astlist.add(dcl);
            astlist.addAll(dcls);
        }
        else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
            // Do nothing for lambda-production
        }
        else error("expected floatdcl, intdcl, id, print, or eof");
        return astlist;
    }

    public AST Dcl() {
        AST itsAst = null;
        if (ts.peek() == FLTDCL) {
            expect(FLTDCL);
            Token t = expect(ID);
            itsAst = new FloatDcl(t.val);
        }
        else if (ts.peek() == INTDCL) {
            expect(INTDCL);
            Token t = expect(ID);
            itsAst = new IntDcl(t.val);
        }
        else error("expected float or int declaration");
        return itsAst;
```

Recursive Decent Parser for ac with AST

```java
    /**
     * Figure 2.7 code
     */
    public ArrayList<AST> Stmts() {
        ArrayList<AST> astlist = new ArrayList<AST>();
        if (ts.peek() == ID || ts.peek() == PRINT) {
            AST stmt = Stmt();
            ArrayList<AST> stms = Stmts();
            astlist.add(stmt);
            astlist.addAll(stms);
        }
        else if (ts.peek() == EOF) {
            // Do nothing for lambda-production
        }
        else error("expected id, print, or eof");
        return astlist;

    }

    public AST Stmt() {
        AST itsAst = null;
        if (ts.peek() == ID) {
            Token tid = expect(ID);
            expect(ASSIGN);
            AST val = Val();
            Computing expr = Expr();
            if (expr == null) itsAst = new Assigning(tid.val,val);
            else {expr.child1 = val; itsAst = new Assigning(tid.val, expr);};
        }
        else if (ts.peek() == PRINT) {
            expect(PRINT);
            Token tid = expect(ID);
            itsAst = new Printing(tid.val);
        }
        else error("expected id or print");
        return itsAst;

    }

```

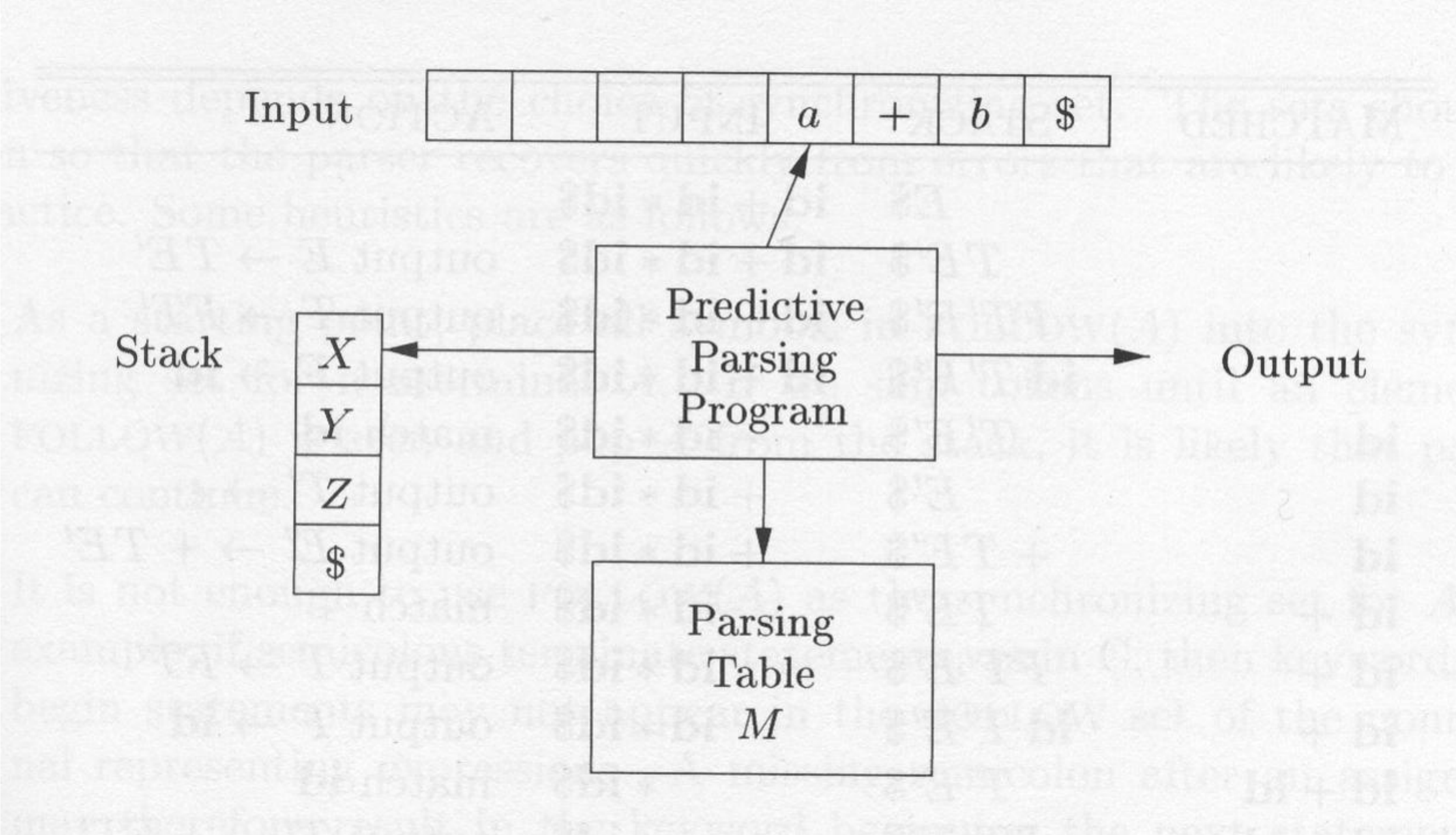
Recursive Decent Parser for ac with AST

```
    }

    public Computing Expr() {
        Computing itsAst = null;
        if (ts.peek() == PLUS) {
            expect(PLUS);
            AST val = Val();
            Computing expr = Expr();
            //The construction of the AST is a little messy as the grammar for the ac language is Expr -> (+|-) Val Expr
            //which will be used in the Stm -> Id assign Val Expr production. However, we really want the AST
            //to have an Assigning node corresponding to Id assign Expr where Expr -> Val (+|-) Expr i.e. a Computing node
            //thus we create a Computing node in this parse method with an empty left child and
            //in the parse method for STM we adjust the AST with the correct left child
            if (expr != null) {expr.child1 = val; itsAst = new Computing("+",null, expr);}
            else itsAst = new Computing("+",null,val);
        }
        else if (ts.peek() == MINUS) {
            expect(MINUS);
            AST val = Val();
            Computing expr = Expr();
            if (expr != null) {expr.child1 = val; itsAst = new Computing("-",null, expr);}
            else itsAst = new Computing("-",null,val);

        }
        else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
            // Do nothing for lambda-production
        }
        else error("expected plus, minus, id, print, or eof");
        return itsAst;

    }

    public AST Val() {
        AST itsAst = null;
        if (ts.peek() == ID) {
            Token tid = expect(ID);
            itsAst = new SymReferencing(tid.val);
        }
        else if (ts.peek() == INUM) {
            Token tid = expect(INUM);
            itsAst = new IntConsting(tid.val);
        }
        else if (ts.peek() == FNUM) {
            Token tid = expect(FNUM);
            itsAst = new FloatConsting(tid.val);
        }
        else error("expected id, inum, or fnum");
        return itsAst;

    }

```

Table-Driven LL(1) Parsers

- Creating recursive-descent parsers can be automated, but
  - Size of parser code
  - Inefficiency: overhead of method calls and returns
- To create table-driven parsers, we use stack to simulate the actions by MATCH() and calls to nonterminals' procedures
  - Terminal symbol: MATCH
  - Nonterminal symbol: table lookup
  - (Fig. 5.8)

Model of a table-driven predictive parser



```
procedure LLparser( ts )
    call push( S )
    accepted ← false
    while not accepted do                                        (5)
        if TOS( ) ∈ Σ                                            (6)
        then
            call match( ts, TOS( ) )                             (7)
            if TOS( ) = $                                        (8)
            then  accepted ← true
            call pop( )                                          (9)
        else
            p ← LLtable[TOS( ), ts.peek( )]                      (10)
            if p = 0
            then
                call error( Syntax error—no production applicable )
            else  call apply( p )
end
procedure apply( p : A → X_1 ... X_m )
    call pop( )                                                  (11)
    for i = m downto 1 do                                        (12)
        call push( X_i )
end
```

Figure 5.8: Generic LL(1) parser.

How to Build LL(1) Parse Table

procedure FILLTABLE(*LLtable*)
  foreach A $\in N$ **do**
    foreach a $\in \Sigma$ **do** *LLtable*[A][a] $\leftarrow 0$
  foreach A $\in N$ **do**
    foreach $p \in$ *ProductionsFor*(A) **do**
      foreach a $\in$ Predict(p) **do** *LLtable*[A][a] $\leftarrow p$
end

Figure 5.9: Construction of an LL(1) parse table.

1 S → A C $
2 C → c
3 | λ
4 A → a B C d
5 | B Q
6 B → b B
7 | λ
8 Q → q
9 | λ

	Lookahead					
Nonterminal	a	b	c	d	q	$
S	1	1	1		1	1
C			2	3		3
A	4	5	5		5	5
B		6	7	7	7	7
Q			9		8	9

Figure 5.10: LL(1) table. The blank entries should trigger error actions in the parser.

ANTLR

- ANTLR is a popular lexer and parser generator in Java.
- Regexp FSM (lexer machine) for tokens
- It allows LL(*) grammars.
  - Does top-down parsing
  - Uses lookahead tokens to decide which path to take
  - Is table driven
  - Each match could
    - invoke a custom action
    - write some text via StringTemplate,
    - generate a Parse tree (or an Abstract Syntax Tree ANTLR v.3)

  - Note LL(*) means that ANTLR uses a parse algorithm that uses k lookahead (usually k=1) as often as possible, but can use regular expressions or even backtracking when making decision. Theory elaborated in 2011 PLDI paper

Java

```
grammar SimpleCalc;

tokens {
    PLUS    = '+' ;
    MINUS   = '-' ;
    MULT    = '*' ;
    DIV = '/' ;
}

@members {
    public static void main(String[] args) throws Exception {
        SimpleCalcLexer lex = new SimpleCalcLexer(new ANTLRFileStream(args[0]));
        CommonTokenStream tokens = new CommonTokenStream(lex);

        SimpleCalcParser parser = new SimpleCalcParser(tokens);

        try {
            parser.expr();
        } catch (RecognitionException e)  {
            e.printStackTrace();
        }
    }
}

/*------------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------------*/

expr    : term ( ( PLUS | MINUS )  term )* ;

term    : factor ( ( MULT | DIV ) factor )* ;

factor  : NUMBER ;


/*------------------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------------------*/

NUMBER  : (DIGIT)+ ;

WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+    { $channel = HIDDEN; } ;

fragment DIGIT  : '0'..'9' ;
```

What can you do in your projects now?

- You should now be able to define the lexical grammar for your langauge
- Implement the Lexer (scanner) by hand or using JLex

- Define the CFG for your language
- Check it is LL(1) or LL(n) for some n
- If it is LL(n) you should be able to implement a parser
  - Recursive decent by hand
  - Recursive decent by using a tool like JavaCC or CoCo/R
  - Table driven by using a tool like ANTLR

Remarks

- Tools
  - Many different tools
  - Downloading and installing them is part of the exercises
  - Judging if a tool is worthwhile using include judging how difficult it is to install and how difficult it is to use
  - Sometimes it is easier to do things by hand than using a tool
  - But if you haven't tried you don't know when

  - Try out the different tools and techniques on a small language or a subset of your own language.
  - Write down proc and cons for each.
  - Lo and behold – you have a section for your report!

Error Reporting

- A common technique is to print the offending line with a pointer to the position of the error.
- The parser might add a diagnostic message like “semicolon missing at this position” if it knows what the likely error is.
- The way the parser is written may influence error reporting is:

```
private void parseAorB () {
        switch (currentToken.kind) {
        case Token.A: {
                acceptIT();
                …
        }
        break;
        case Token.B: {
                acceptIT();
                …
        }
        break;
        default:
                report a syntax error
        }
}
```

Error Reporting

```
private void parseAorB () {
        if (currentToken.kind == Token.A) {
                acceptIT();
                …
        } else {
                acceptIT();
                …
        }
}
```

How to handle Syntax errors

- Error Recovery : The parser should try to recover from an error quickly so subsequent errors can be reported. If the parser doesn't recover correctly it may report spurious errors.

- Possible strategies:
  - Panic-mode Recovery
  - Phase-level Recovery
  - Error Productions

Panic-mode Recovery

- Discard input tokens until a synchronizing token (like; or end) is found.
- Simple but may skip a considerable amount of input before checking for errors again.
- Will not generate an infinite loop.

Phrase-level Recovery

- Perform local corrections
- Replace the prefix of the remaining input with some string to allow the parser to continue.
  - Examples: replace a comma with a semicolon, delete an extraneous semicolon or insert a missing semicolon. Must be careful not to get into an infinite loop.

Recovery with Error Productions

- Augment the grammar with productions to handle common errors

- Example:

```
param_list
 ::= identifier_list : type
 |   param_list, identifier_list : type
 |   param_list; error identifier_list : type
    (“comma should be a semicolon”)
```

```
1  S → [ E ]
2    | ( E )
3  E → a
```

```
procedure S( ts, termset )
    switch ()
        case ts.PEEK( ) ∈ { [ }
            call MATCH( [ )
            call E( ts, termset ∪ { ] } )                                (18)
            call MATCH( ] )
        case ts.PEEK( ) ∈ { ( }
            call MATCH( ( )
            call E( ts, termset ∪ { ) } )                                (19)
            call MATCH( ) )
end
procedure E( ts, termset )
    if ts.PEEK( ) = a
    then  call MATCH( ts, a )
    else
        call ERROR( Expected an a )
        while ts.PEEK( ) ∉ termset  do  call ts.ADVANCE( )
end
```

Figure 5.26: A grammar and its Wirth-style, error-recovering parser.

Languages and Compilers
(SProg og Oversættere)

Lecture 8
Bottom Up Parsing

Bent Thomsen

Department of Computer Science

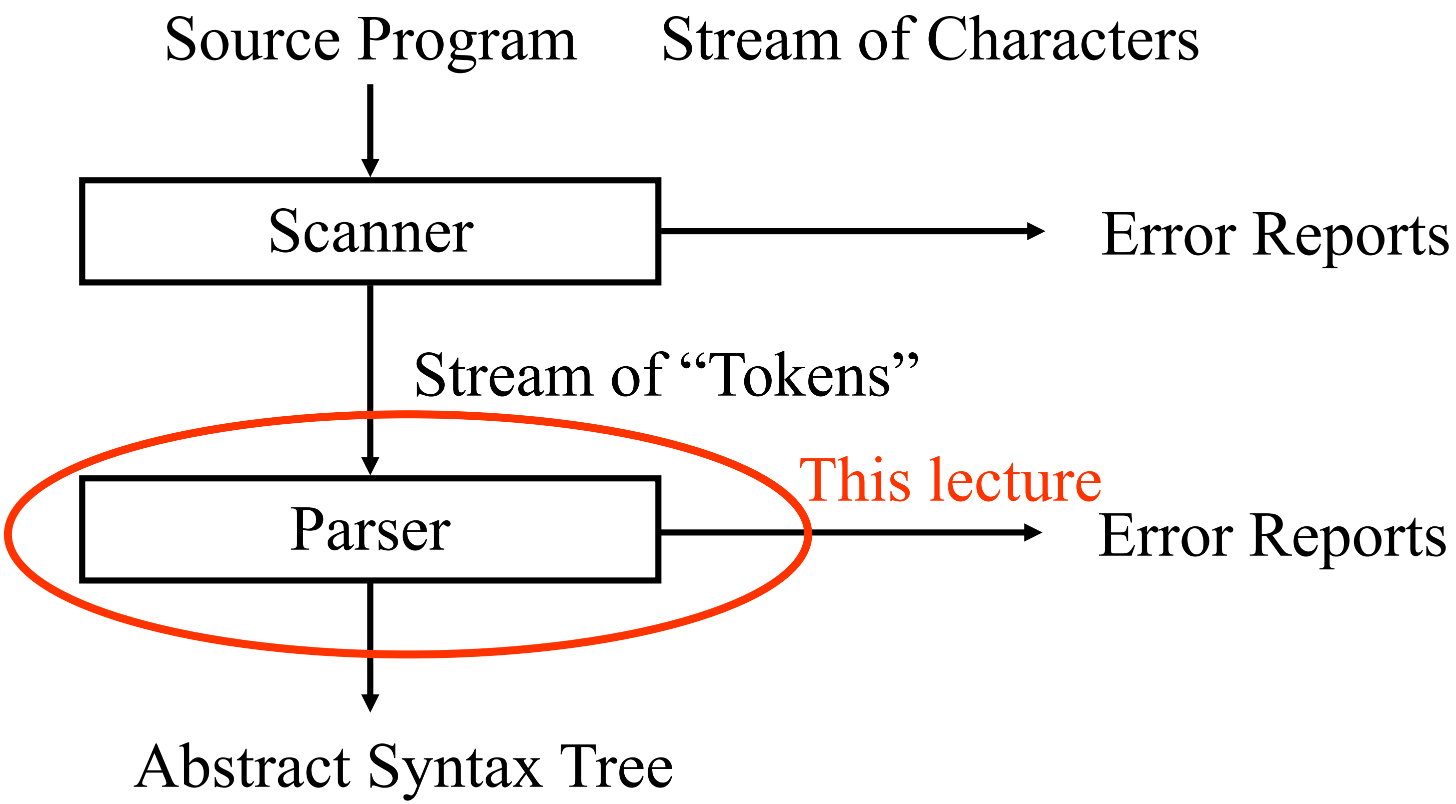
Aalborg University

Learning goals

- Get an overview of bottom up parsing
- Understand what shift/reduce and reduce/reduce conflicts are
- Get an overview of JavaCUP
- Get an overview of SableCC

Syntax Analysis

Dataflow chart



Generation of parsers

- We have seen that recursive decent parsers can be constructed by hand or automatically, e.g. JavaCC
- However, recursive decent parsers only work for LL(k) grammars
  - No Left-recursion
  - No Common prefixes (*)

  - (*) Note that the LL(*) approach used by ANTLR can deal with common prefixes, but not left recursion in general, though ANTLR4 can do some left recursion elimination.

1	Stmt	→	if Expr then StmtList endif
2		|	if Expr then StmtList else StmtList endif
3	StmtList	→	StmtList ; Stmt
4		|	Stmt
5	Expr	→	var + Expr
6		|	var

Figure 5.12: A grammar with common prefixes.

---

procedure FACTOR ( )
    foreach $\mathsf{A} \in N$ **do**
        $\alpha \leftarrow LongestCommonPrefix(ProductionsFor(\mathsf{A}))$
        while $|\alpha| > 0$ **do**
            $V \leftarrow$ **new** *NonTerminal* ( )
            $Productions \leftarrow Productions \cup \{ \mathsf{A} \rightarrow \alpha V \}$
            foreach $p \in ProductionsFor(A) \mid RHS(p) = \alpha\beta_p$ **do** (13)
                $Productions \leftarrow Productions - \{ p \}$
                $Productions \leftarrow Productions \cup \{ V \rightarrow \beta_p \}$
            $\alpha \leftarrow LongestCommonPrefix(ProductionsFor(\mathsf{A}))$
end

Figure 5.13: Factoring common prefixes.

1	Stmt	$\rightarrow$	if Expr then StmtList V_1
2	V_1	$\rightarrow$	endif
3		|	else StmtList endif
4	StmtList	$\rightarrow$	StmtList ; Stmt
5		|	Stmt
6	Expr	$\rightarrow$	var V_2
7	V_2	$\rightarrow$	+ Expr
8		|	λ

Figure 5.14: Factored version of the grammar in Figure 5.12.

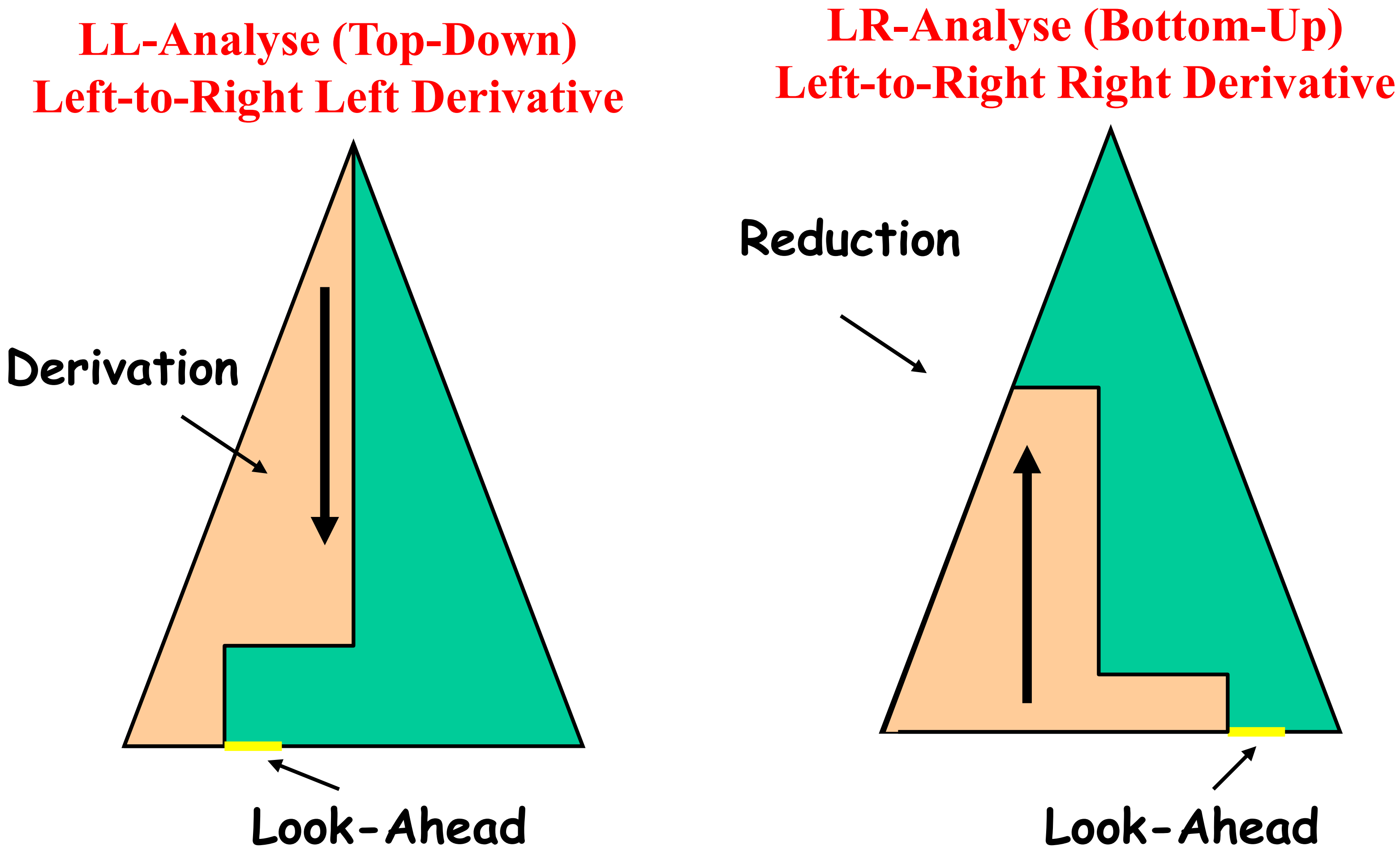
```
procedure EliminateLeftRecursion( )
    foreach A ∈ N do
        if ∃ r ∈ ProductionsFor(A) | RHS(r) = Aα
        then
            X ← new NonTerminal( )
            Y ← new NonTerminal( )
            foreach p ∈ ProductionsFor(A) do
                if p = r
                then  Productions ← Productions ∪ { A → X Y }
                else  Productions ← Productions ∪ { X → RHS(p) }
            Productions ← Productions ∪ { Y → αY, Y → λ }
end
```

Figure 5.15: Eliminating left recursion.

1	Stmt	$\rightarrow$	if Expr then StmtList V_1
2	V_1	$\rightarrow$	endif
3		|	else StmtList endif
4	StmtList	$\rightarrow$	X Y
5	X	$\rightarrow$	Stmt
6	Y	$\rightarrow$	; Stmt Y
7		|	λ
8	Expr	$\rightarrow$	var V_2
9	V_2	$\rightarrow$	+ Expr
10		|	λ

Figure 5.16: LL(1) version of the grammar in Figure 5.14.

Top-Down vs. Bottom-Up parsing

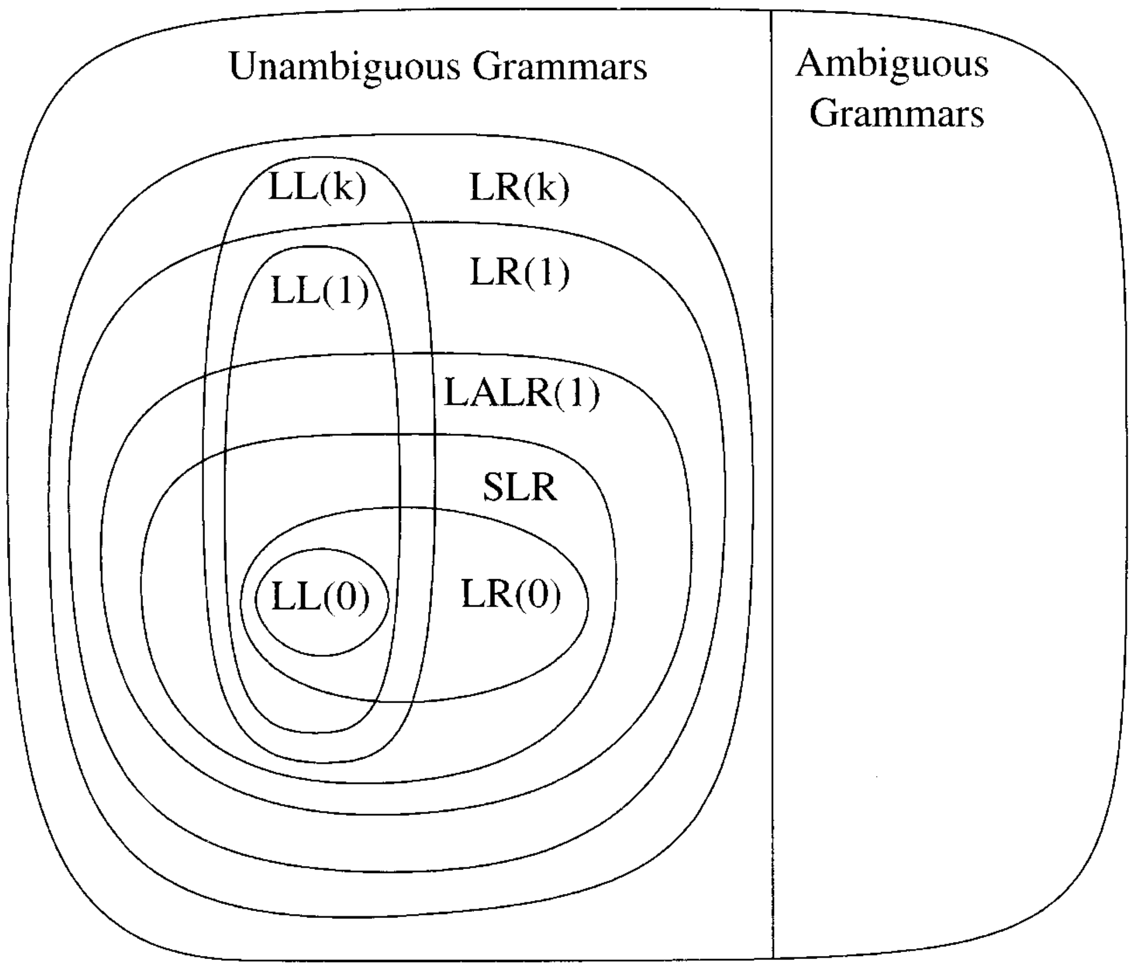


Generation of parsers

- Sometimes we need a more powerful language
- The LR languages are more powerful
  - Can recognize LR(0), SLR(1), LALR(1), LR(k) grammars
    - bigger class of grammars than LL
  - Can handle left recursion!
  - Usually more convenient because less need to rewrite the grammar.
- LR parsing methods are the most commonly used for automatic tools today (LALR in particular)
  - Parsers for LR languages use a bottom-up parsing strategy
  - Harder to implement than LL parsers
    - but tools exist (e.g. JavaCUP, Yacc, C#CUP and SableCC)

- Bottom-up parsers can handle the largest class of grammars that can be parsed deterministically

Hierarchy



Unambiguous Grammars
Ambiguous Grammars
LL(k)
LR(k)
LL(1)
LR(1)
LALR(1)
SLR
LL(0)
LR(0)

Bottom Up Parsers: Overview of Algorithms

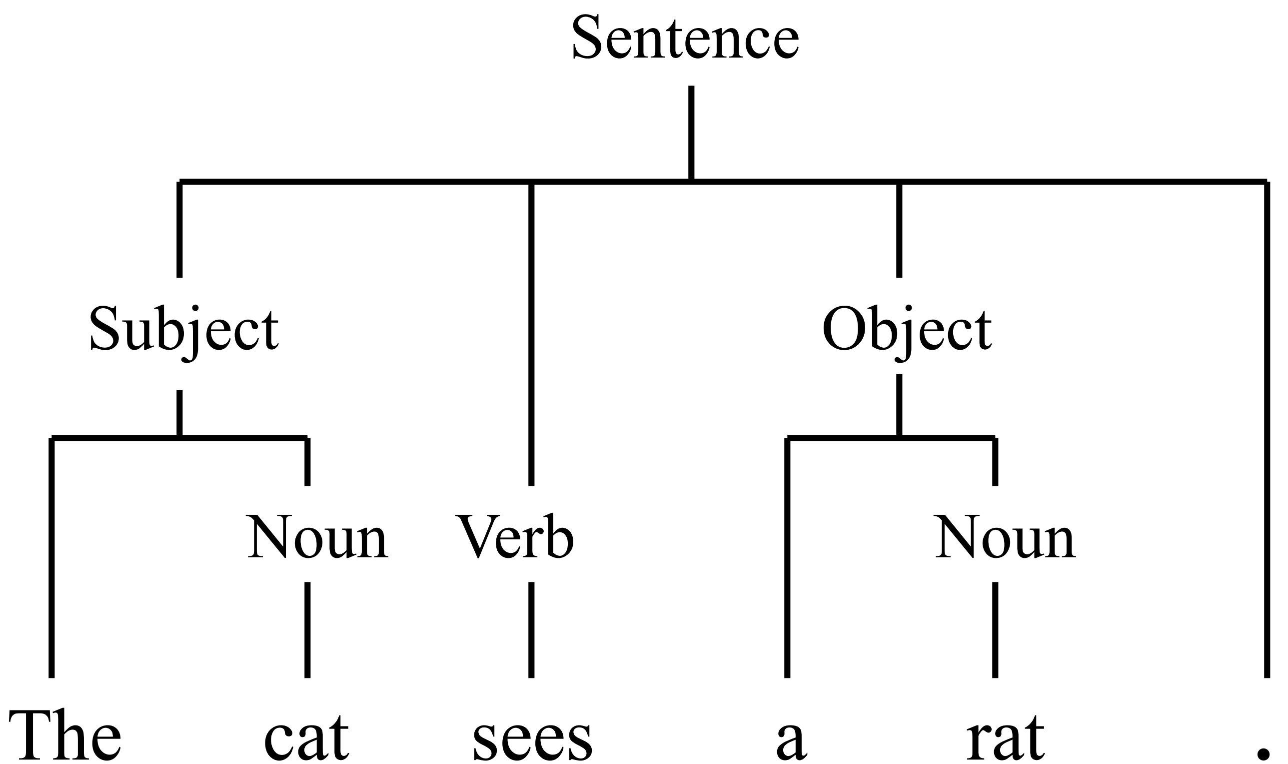
- LR(0) : The simplest algorithm
  - theoretically important but rather weak (not practical)
- SLR(1) : An improved version of LR(0)
  - more practical but still rather weak.
- LR(1) : LR(0) algorithm with extra lookahead token.
  - very powerful algorithm. Not often used because of large memory requirements (very big parsing tables)
  - Note: LR(0) and LR(1) use 1 lookahead taken when operating
    - 0 resp. 1 refer to token used in table construction.
- LR(k) for k>0, k tokens are use for operation and table
- LALR : "Watered down" version of LR(1)
  - still very powerful, but has much smaller parsing tables
  - most commonly used algorithm today

Fundamental idea

- Read through every construction and recognize the construction at the end
- LR:
  - Left – the string is read from left to right
  - Right – we get a right derivation (in reverse)
- The parse tree is build from bottom up
  - Corresponds to a right derivation in reverse

Bottom up parsing

The parse tree “grows” from the bottom (leafs) up to the top (root).



Right derivations

```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

Sentence

→ Subject Verb Object .

→ Subject Verb a Noun .

→ Subject Verb a rat .

→ Subject sees a rat .

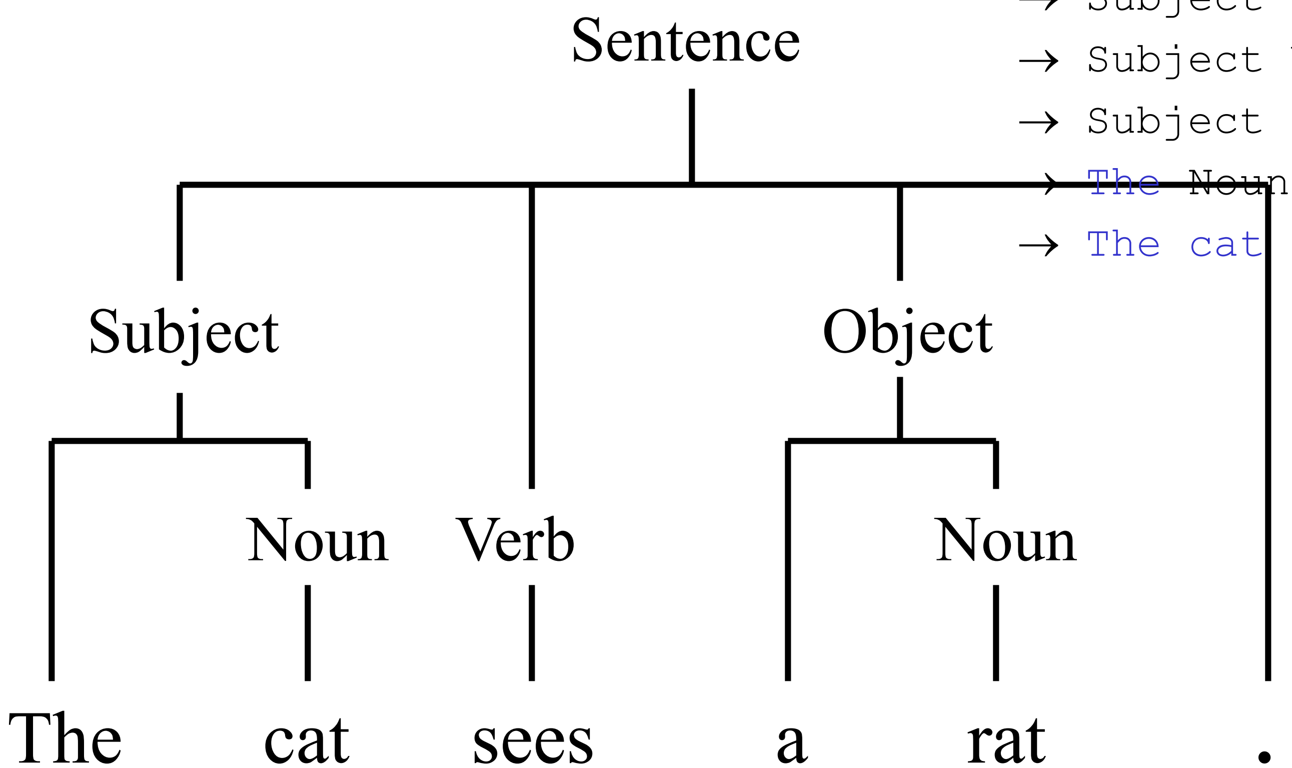
→ The Noun sees a rat .

→ The cat sees a rat .

Bottom up parsing

The parse tree "grows" from the bottom (leafs) up to the top (root).
Just read the right derivations backwards

```
Sentence
→ Subject Verb Object .
→ Subject Verb a Noun .
→ Subject Verb a rat .
→ Subject sees a rat .
→ The Noun sees a rat .
→ The cat sees a rat .
```



Some Terminology

- A Rightmost (canonical) derivation is a derivation where the rightmost nonterminal is replaced at each step. A rightmost derivation from α to β is noted $\alpha \overset{*}{\Rightarrow}_{rm} \beta$.
- A reduction transforms uwv to uAv if $A \rightarrow w$ is a production
- α is a right sentential form if $S \overset{*}{\Rightarrow}_{rm} \alpha$ with $\alpha = \beta x$ where x is a string of terminals.
- A handle of a right sentential form γ $(= \alpha\beta w)$ is a production $A \rightarrow \beta$ and a position in γ where β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ:

$$S \overset{*}{\Rightarrow}_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$$

  - Informally, a handle is a production we can reverse without getting stuck.
  - If the handle is $A \rightarrow \beta$, we will also call β the handle.

handles and reductions

```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

```
The cat sees a rat .
→ the Noun sees a rat .
→ Subject sees a rat .
→ Subject Verb a rat .
→ Subject Verb a Noun .
→ Subject Verb Object .
→ Sentence
```

```
Handles:
Noun ::= cat
Subject ::= the Noun
Verb ::= sees
Noun ::= rat
Object ::= a Noun
Sentence  ::=
 Subject Verb Object.
```

Shifting and reducing

```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

```
Shift                                 → ← the cat sees a rat .
Shift                             the → ← cat sees a rat .
Reduce                        the cat → ← sees a rat .
Shift                            the  → ← Noun sees a rat .
Reduce                       the Noun → ← sees a rat .
Reduce                                → ← Subject sees a rat .
Shift                         Subject → ← sees a rat .
Reduce                   Subject sees → ← a rat .
Shift                        Subject  → ← Verb a rat .
Shift                    Subject Verb → ← a rat .
Shift                  Subject Verb a → ← rat .
Reduce             Subject Verb a rat → ←.
Shift                    Subject Verb → ← Noun.
Reduce            Subject Verb a Noun → ←.
Shift                    Subject Verb → ← Object.
Shift             Subject Verb Object → ←.
Shift           Subject Verb Object . → ←
Reduce                                → ← Sentence
Finish                       Sentence → ←
```

Shifting and reducing

```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

```
Shift                                  → ← the cat sees a rat .
Shift                              the → ← cat sees a rat .
Reduce                         the cat → ← sees a rat .
Reduce                        the Noun → ← sees a rat .
Reduce                         Subject → ← sees a rat .
Shift                     Subject sees → ← a rat .
Shift                   Subject Verb a → ← rat .
Shift               Subject Verb a rat → ←.
Reduce             Subject Verb a Noun → ←.
Reduce             Subject Verb Object → ←.
Shift            Subject Verb Object . → ←
Reduce                        Sentence → ←
```

The knitting games

1 Start → E $
2 E → plus E E
3 | num

Derivation

$$Start \Rightarrow_{rm} E\ \$ \Rightarrow_{rm} plus\ E\ E\ \$ \Rightarrow_{rm} plus\ E\ num\ \$ \Rightarrow_{rm} plus\ num\ num\ \$$$

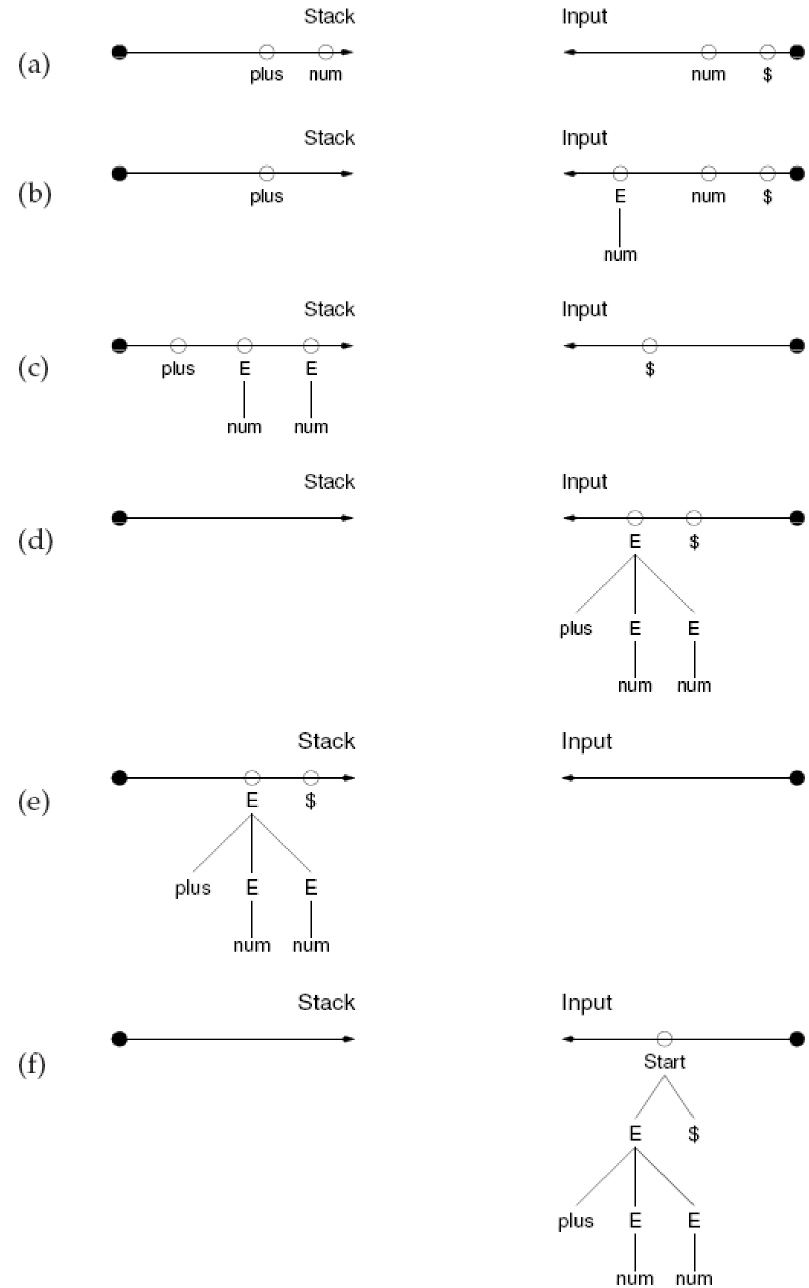


Figure 6.1: Bottom-up parsing resembles knitting.

Bottom Up Parsing

- The main task of a bottom-up parser is to find the leftmost node in the parse tree that has not yet been constructed but all of whose children have been constructed.
- The sequence of children is the **handle**.
- Creating a parent node *N* and connecting the children in the handle to *N* is called **reducing** to *N*.

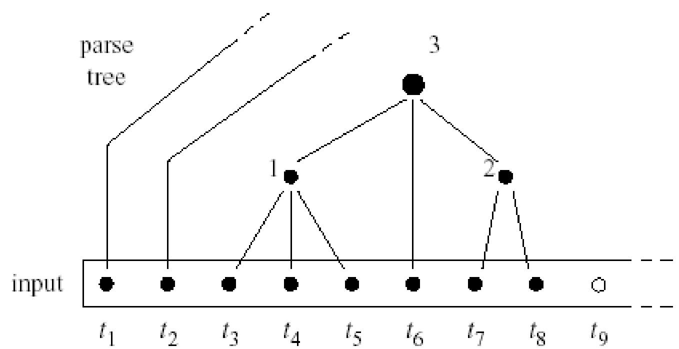


Figure 2.52 A bottom-up parser constructing its first, second, and third nodes.

(1,6,2) is a handle

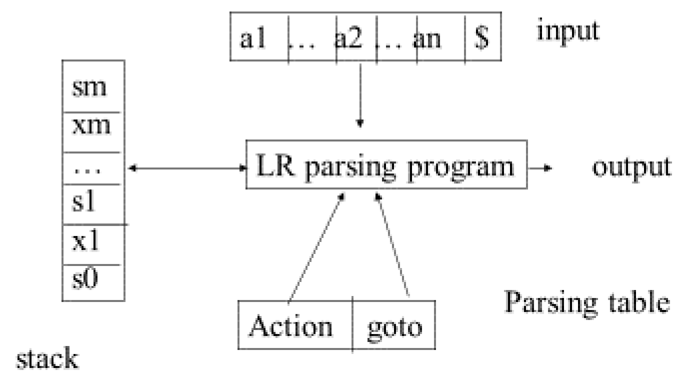
Bottom Up Parsers

- All bottom up parsers have similar algorithm:
  - A loop with these parts:
    - try to find the leftmost node of the parse tree which has not yet been constructed, but all of whose children *have* been constructed.
      - This sequence of children is called a **handle**
      - The sequence of children is built by pushing also called **shifting** elements on a stack
    - construct a new parse tree node.
      - This is called **reducing**
- The difference between different algorithms is only in the way they find a handle.

The LR-parse algorithm

- A stack
  - with objects (symbol, state)
- A finite automaton
  - With transitions and states
- A parse table

Model of an LR parser:



si is a state, xi is a grammar symbol

All LR parsers use the same algorithm, different grammars have different parsing tables.

Bottom-up Parsing

- Shift-Reduce Algorithms
  - **Shift** is the action of moving the next token to the top of the parse stack (and record the state)
  - **Reduce** is the action of replacing the handle on the top of the parse stack with its corresponding LHS

  - Note: In Fischer et. al. the reduce action is a two step process where the LHS is prepended the input stream first and next is shifted to the parse stack (remember the knitting game)

The parse table

- For every state and every terminal
  - either shift x

    Put next input-symbol on the stack and go to state x
  - or reduce production

    On the stack we now have symbols to go backwards in the production – afterwards do a goto
- For every state and every non-terminal
  - Goto x

    Tells us, in which state to be in after a reduce-operation

    (Note as Fischer et. al. prepends non-terminals to input, they have a shift/goto action in their tables)
- Empty cells in the table indicate an error

```
call Stack.PUSH(StartState)
accepted ← false
while not accepted do
    action ← Table[Stack.TOS( )][InputStream.PEEK( )]        ①
    if action = shift s
    then
        call Stack.PUSH(s)                                   ②
        if s ∈ AcceptStates                                  ③
        then   accepted ← true
        else   call InputStream.ADVANCE( )
    else
        if action = reduce A→γ
        then
            call Stack.POP(|γ|)                              ④
            call InputStream.PREPEND(A)                      ⑤
        else
            call ERROR( )                                    ⑥
```

Figure 6.3: Driver for a bottom-up parser.

Example Grammar

- (0) S' → S$
  - This production *augments* the grammar
- (1) S → (S)S
- (2) S → ε

- This grammar generates all expressions of matching parentheses

Example - parse table

	(	)	$	*S'*	*S*
0	s2	r2	r2		g1
1		s3	r0		
2	s2	r2	r2		g3
3		s4			
4	s2	r2	r2		g5
5		r1	r1		

By reduce we indicate the number of the production
r0 = accept
Never a goto by S'

Example – parsing

Stack	Input	Action
$\$_0$	()()$	shift 2
$\$_0(_2$	)()$	reduce S→ε
$\$_0(_2S_3$	)()$	shift 4
$\$_0(_2S_3)_4$	()$	shift 2
$\$_0(_2S_3)_4(_2$	)$	reduce S→ε
$\$_0(_2S_3)_4(_2S_3$	)$	shift 4
$\$_0(_2S_3)_4(_2S_3)_4$	$	reduce S→ε
$\$_0(_2S_3)_4(_2S_3)_4S_5$	$	reduce S→(S)S
$\$_0(_2S_3)_4S_5$	$	reduce S→(S)S
$\$_0S_1$	$	reduce S'→S

- (0) S' → S$
  - This production *augments* the grammar
- (1) S → (S)S
- (2) S → ε

	(	)	$	*S'*	*S*
0	s2	r2	r2		g1
1		s3	r0		
2	s2	r2	r2		g3
3		s4			
4	s2	r2	r2		g5
5		r1	r1		

The resultat

- Read the productions backwards and we get a right derivation:
- $S' \Rightarrow S \Rightarrow (S)S \Rightarrow (S)(S)S$
  $\Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ()()$

- (0) $S' \to S\$$
  - This production *augments* the grammar
- (1) $S \to (S)S$
- (2) $S \to \varepsilon$

LR(0)-DFA

- How do we get the parse table?
- We build a DFA and encode it in a table!
  - Every state is a set of items
  - Transitions are labeled by symbols
  - States must be *closed*
  - New states are constructed from states and transitions

LR(0)-items

Item :

A production with a selected position marked by a point

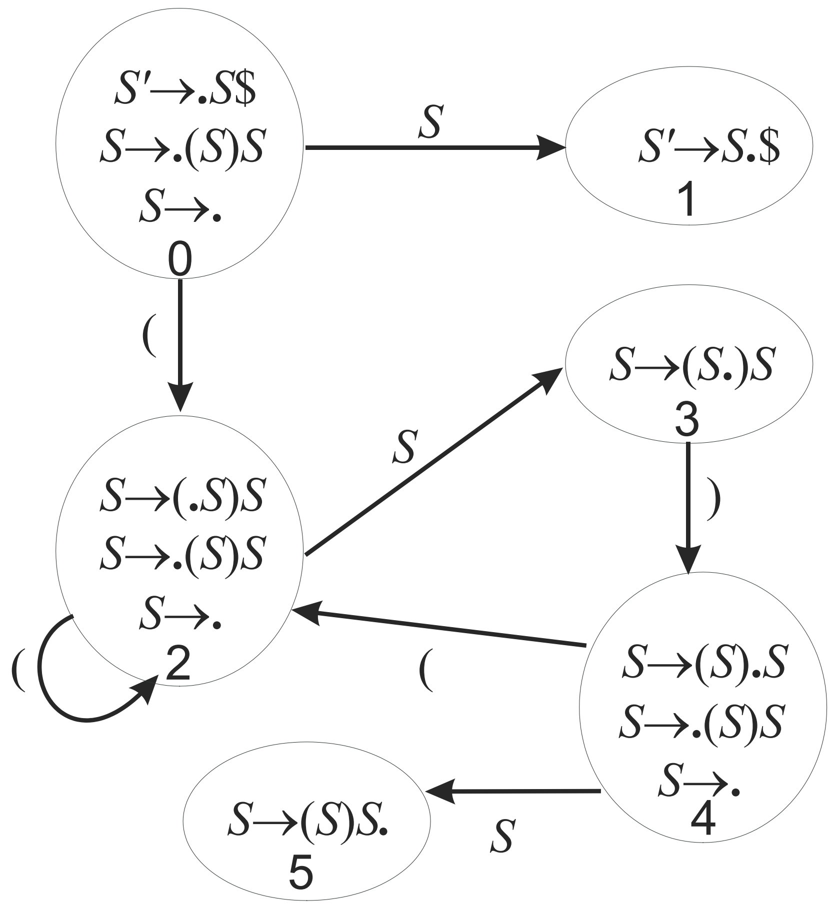
$X \rightarrow \alpha.\beta$ indicates that on the stack we have α and the first of the input can be derived from β

Our example grammar has the following items:

$S' \rightarrow .S\$$	$S' \rightarrow S.\$$	$(S' \rightarrow S\$.)$
$S \rightarrow .(S)S$	$S \rightarrow (.S)S$	$S \rightarrow (S.)S$
$S \rightarrow (S).S$	$S \rightarrow (S)S.$	$S \rightarrow .$

Rules with . at the end are the handles

The DFA for our grammar



```
function ComputeLR0( Grammar ) returns (Set, State)
    States ← ∅
    StartItems ← { Start → • RHS( p ) | p ∈ ProductionsFor( Start ) }    (7)
    StartState ← AddState( States, StartItems )
    while (s ← WorkList.ExtractElement( )) ≠ ⊥ do                           (8)
        call ComputeGoto( States, s )
    return ((States, StartState))
end
function AddState( States, items ) returns State
    if items ∉ States                                                      (9)
    then
        s ← newState(items)                                                (10)
        States ← States ∪ { s }
        WorkList ← WorkList ∪ { s }                                        (11)
        Table[s][★] ← error                                                (12)
    else   s ← FindState(items)
    return (s)
end
function AdvanceDot( state, X ) returns Set
    return ({ A → αX • β | A → α • Xβ ∈ state })                           (13)
end
```

Figure 6.9: LR(0) construction.

```
function Closure(state) returns Set
    ans ← state
    repeat                                                    (14)
        prev ← ans
        foreach A → α • Bγ ∈ ans do                           (15)
            foreach p ∈ ProductionsFor(B) do
                ans ← ans ∪ { B → • RHS(p) }                  (16)
    until ans = prev
    return (ans)
end
procedure ComputeGoto(States, s)
    closed ← Closure(s)                                       (17)
    foreach X ∈ (N ∪ Σ) do                                    (18)
        RelevantItems ← AdvanceDot(closed, X)                 (19)
        if RelevantItems ≠ ∅
        then
            Table[s][X] ← shift AddState(States, RelevantItems)   (20)
end
```

Figure 6.10: LR(0) closure and transitions.

```
procedure CompleteTable(Table, grammar)
    call ComputeLookahead( )
    foreach state ∈ Table do
        foreach rule ∈ Productions(grammar) do
            call TryRuleInState(state, rule)
    call AssertEntry(StartState, GoalSymbol, accept)          (21)
end
procedure AssertEntry(state, symbol, action)
    if Table[state][symbol] = error                           (22)
    then   Table[state][symbol] ← action
    else
        call ReportConflict(Table[state][symbol], action)     (23)
end
```

Figure 6.13: Completing an LR(0) parse table.

```
procedure ComputeLookahead( )
   /*    Reserved for the LALR(k) computation given in Section 6.5.2    */
end
procedure TryRuleInState(s, r)
   if LHS(r) → RHS(r) • ∈ s
   then
       foreach X ∈ (Σ ∪ N) do call AssertEntry(s, X, reduce r)
end
```

Figure 6.14: LR(0) version of TryRuleInState.

Pause

Shift-reduce-conflicts

- What happens, if there is a shift and a reduce in the same cell
  - so we have a shift-reduce-conflict
  - and the grammar is not LR(0)

- Our example grammar is not LR(0)

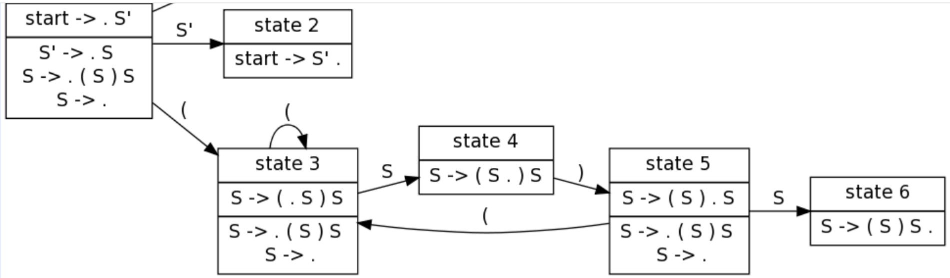
  - (0) S' → S$
    - This production *augments* the grammar
  - (1) S → (S)S
  - (2) S → ε

Shift-reduce-conflicts

	(	)	$	*S'*	*S*
0	**s2/r2**	r2	r2		g1
1	r0	**s3/r0**	r0		
2	**s2/r2**	r2	r2		g3
3		s4			
4	**s2/r2**	r2	r2		g5
5	r1	r1	r1		

http://smlweb.cpsc.ucalgary.ca/

http://smlweb.cpsc.ucalgary.ca/



Grammar

S' → S .
S → (S) S
| .

LR(0) Table

	$	)	(	S'	S
0	r(S → &epsilon)	r(S → &epsilon)	r(S → &epsilon)/s3	s2	s1
1	r(S' → S)	r(S' → S)	r(S' → S)		
2	acc	acc	acc		
3	r(S → &epsilon)	r(S → &epsilon)	r(S → &epsilon)/s3		s4
4		s5			
5	r(S → &epsilon)	r(S → &epsilon)	r(S → &epsilon)/s3		s6
6	r(S → (S) S)	r(S → (S) S)	r(S → (S) S)		

SLR(1) Table

	$	)	(	S'	S
0	r(S → &epsilon)	r(S → &epsilon)	s3	s2	s1
1	r(S' → S)				
2	acc				
3	r(S → &epsilon)	r(S → &epsilon)	s3		s4
4		s5			
5	r(S → &epsilon)	r(S → &epsilon)	s3		s6
6	r(S → (S) S)	r(S → (S) S)			

The grammar is not LR(0) because:

- shift/reduce conflict in state 0.

LR(0) Conflicts

The LR(0) algorithm doesn't always work. Sometimes there are "problems" with the grammar causing LR(0) conflicts.

An LR(0) conflict is a situation (DFA state) in which there is more than one possible action for the algorithm.

More precisely there are two kinds of conflicts:

- Shift-reduce
  - When the algorithm cannot decide between a shift action or a reduce action
- Reduce-reduce
  - When the algorithm cannot decide between two (or more) reductions (for different grammar rules).

LR(0) vs. SLR(1)

- LR(0) - when constructing the parse table, we do not look at the next symbol in the input before we decide whether to shift or to reduce
  - Note that we do use the next symbol in the input when looking up in the parse table

- SLR(1) - here we do look at the next symbol
- the parse table is a bit different:
  - shift and goto as with LR(0)
  - reduce $X \rightarrow \alpha$ only in cells (X,w) with $w \in \text{follow}(X)$
  - this means fewer reduce-actions and therefore this rule removes at lot of potential s/r- or r/r-conflicts

```
procedure TryRuleInState(s, r)
    if LHS(r) → RHS(r) • ∈ s
    then
        foreach X ∈ Follow(LHS(r)) do
            call AssertEntry(s, X, reduce r)
end
```

Figure 6.23: SLR(1) version of TryRuleInState.

LR(1)

- Items are now pairs $(A \rightarrow \alpha.\beta\ , t)$
  - t is a terminal such that $t \in \text{follow}(A)$
  - means that the top of the stack is α and the input can be derived from βt

  - The initial state is generated from $(S' \rightarrow .S\$, ?)$
  - Closure-operation is different
  - Shift and Goto is (more or less) the same
  - state I with item $(A \rightarrow \alpha., z)$ gives a reduce $A \rightarrow \alpha$ in cell (I,z)

  - LR(1)-parse tables are very big

Marker (7): We initialize *StartItems* by including LR(1) items that have \$ as the follow symbol:

$StartItems \leftarrow \{ [\,\mathsf{Start} \rightarrow \bullet\, \mathrm{RHS}(p)\,,\$\,] \mid p \in \textsc{ProductionsFor}(\,\mathsf{Start}\,) \}$

Marker (13): We augment the LR(0) item so that AdvanceDot returns the appropriate LR(1) items:

return $(\{ [\,\mathsf{A} \rightarrow \alpha \mathcal{X} \bullet \beta\,,\mathsf{a}\,] \mid [\,\mathsf{A} \rightarrow \alpha \bullet \mathcal{X} \beta\,,\mathsf{a}\,] \in state \})$

Marker (15): This entire loop is replaced by the following:

```
foreach [ A → α • Bγ , a ] ∈ ans do
    foreach p ∈ ProductionsFor(B) do
        foreach b ∈ First(γa) do                    (31)
            ans ← ans ∪ { [ B → • RHS(p) , b ] }
```

Figure 6.38: Modifications to Figures 6.9 and 6.10 to obtain an LR(1) parser

```
procedure TryRuleInState(s, r)
    if [ LHS(r) → RHS(r) • , w ] ∈ s
    then call AssertEntry(s, w, reduce r)
end
```

Figure 6.39: LR(1) version of TryRuleInState.

Example

0: $S' \rightarrow S\$$

1: $S \rightarrow V=E$

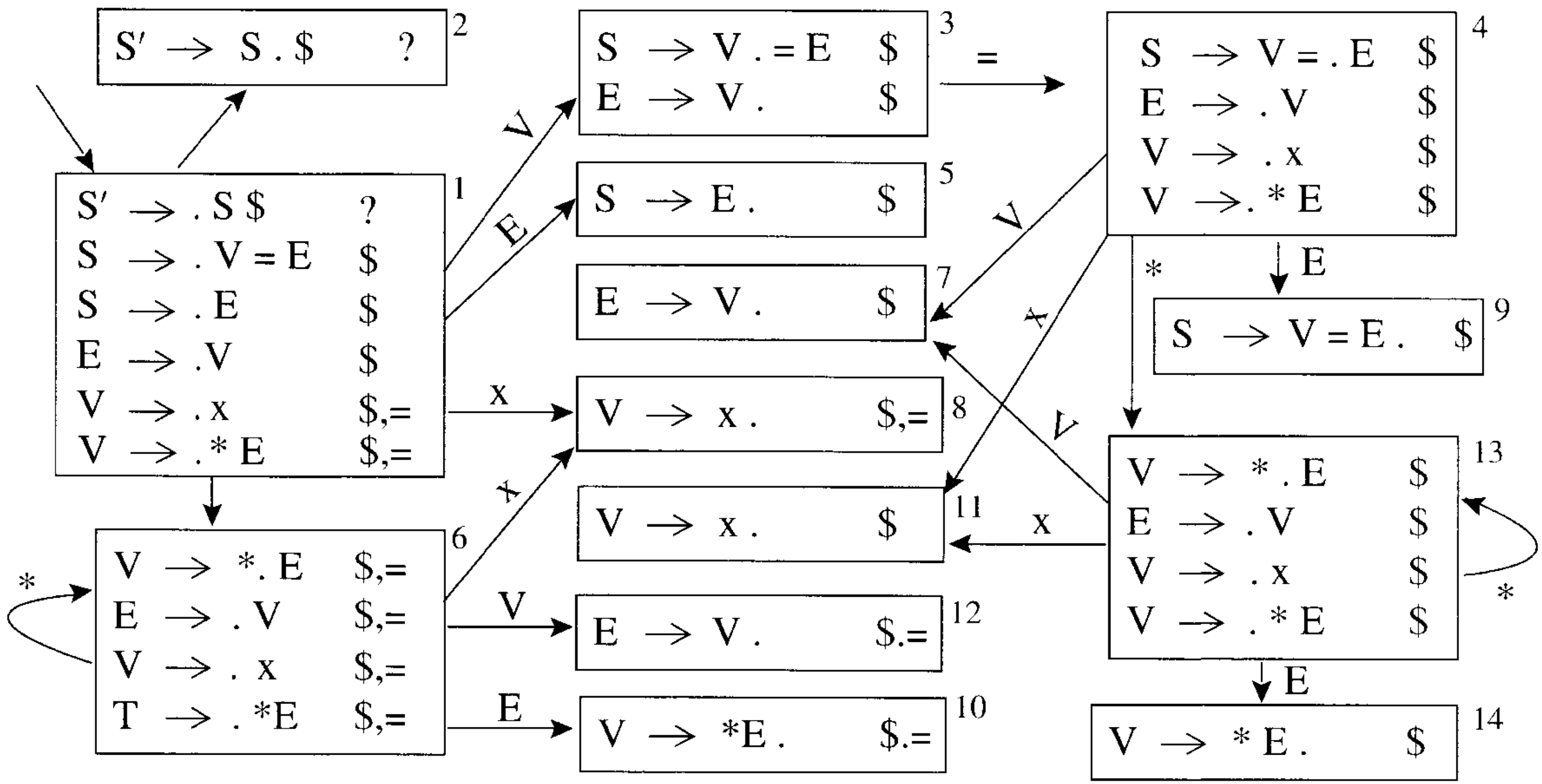
2: $S \rightarrow E$

3: $E \rightarrow V$

4: $V \rightarrow x$

5: $V \rightarrow *E$

LR(1)-DFA



S′ → S . $?
2
S → V . = E $
E → V . $
3
=
S → V = . E $
E → . V $
V → . x $
V → . * E $
4
S′ → . S $?
S → . V = E $
S → . E $
E → .V $
V → . x $,=
V → .* E $,=
1
V
E
S → E . $
5
E → V . $
7
x
V → x . $,=
8
*
E
S → V = E . $
9
V → * . E $
E → . V $
V → . x $
V → . * E $
13
x
V → x . $
11
V → *. E $,=
E → . V $,=
V → . x $,=
T → . *E $,=
6
E → V . $.=
12
V → *E . $.=
10
V → * E . $
14

LR(1)-parse table

	x	*	=	$	S	E	V		x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3	8			r4	r4			
2				acc				9				r1			
3			s4	r3				10			r5	r5			
4	s11	s13				g9	g7	11				r4			
5				r2				12			r3	r3			
6	s8	s6				g10	g12	13	s11	s13				g14	g7
7				r3				14				r5			

LALR(1)

- A variant of LR(1) - gives smaller parse tables

- We allow ourselves in the DFA to combine states, where the items are the same except the *x*.

- In our example we combine the states
  - 6 and 13
  - 7 and 12
  - 8 and 11
  - 10 and 14

```
procedure TryRuleInState(s, r)
    if LHS(r) → RHS(r) •  ∈ s
    then
        foreach 𝒳 ∈ Σ do
            if 𝒳 ∈ ItemFollow((s, LHS(r) → RHS(r) • ))
            then call AssertEntry(s, 𝒳, reduce r)
end
```

Figure 6.27: LALR(1) version of TryRuleInState.

```
procedure ComputeLookahead( )
    call BuildItemPropGraph( )
    call EvalItemPropGraph( )
end
procedure BuildItemPropGraph( )
    foreach s ∈ States do
        foreach item ∈ state do
            v ← Graph.AddVertex((s, item))                                  24
            ItemFollow(v) ← ∅
    foreach p ∈ ProductionsFor(Start) do
        ItemFollow((StartState, Start → • RHS(p))) ← { $ }                  25
    foreach s ∈ States do
        foreach A → α • Bγ ∈ s do                                           26
            v ← Graph.FindVertex((s, A → α • Bγ))
            call Graph.AddEdge(v, (Table[s][B], A → αB • γ))                27
            foreach (w ← (s, B → • δ)) ∈ Graph.Vertices do
                ItemFollow(w) ← ItemFollow(w) ∪ First(γ)                    28
                if AllDeriveEmpty(γ)                                        29
                then call Graph.AddEdge(v, w)
end
procedure EvalItemPropGraph( )
    repeat                                                                  30
        changed ← false
        foreach (v, w) ∈ Graph.Edges do
            old ← ItemFollow(w)
            ItemFollow(w) ← ItemFollow(w) ∪ ItemFollow(v)
            if ItemFollow(w) ≠ old
            then changed ← true
    until not changed
end
```

Figure 6.28: LALR(1) version of ComputeLookahead.

LALR(1)-parse-table

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				acc			
3			s4	r3			
4	s8	s6				g9	g7
5							
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

4 kinds of parsers

- 4 ways to generate the parse table
- LR(0)
  - Easy, but only a few grammars are LR(0)
- SLR(1)
  - Relativey easy, a few more grammars are SLR
- LR(1)
  - Expensive, but alle common languages are LR(1)
- LALR(1)
  - A bit difficult, but simpler and more efficient than LR(1)
  - In practice allmost all grammars are LALR(1)

Parser Conflict Resolution

Most programming language grammars are LR(1). But, in practice, you still encounter grammars which have parsing conflicts.

=> a common cause is an **ambiguous grammar**

Ambiguous grammars always have parsing conflicts (because they are ambiguous this is just unavoidable).

In practice, parser generators still generate a parser for such grammars, using a "resolution rule" to resolve parsing conflicts deterministically.

=> The resolution rule may or may not do what you want/expect

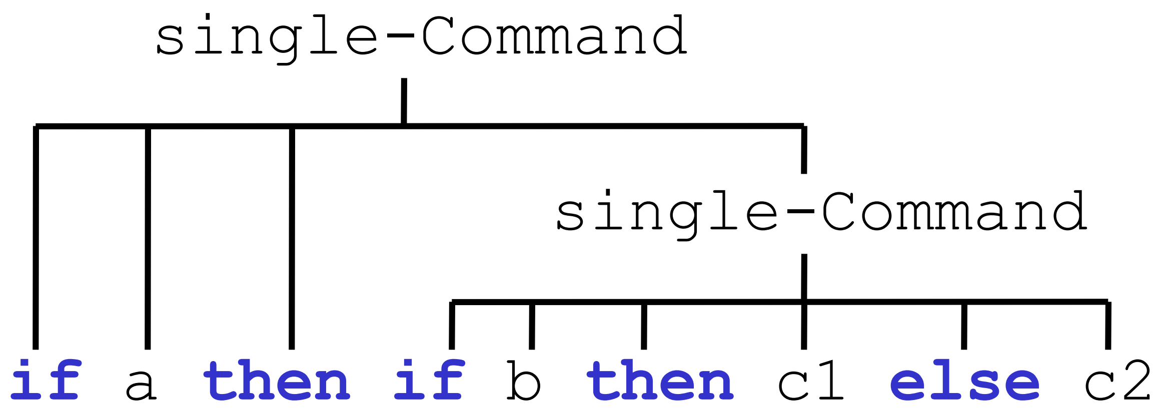
=> You will get a warning message. If you know what you are doing this can be ignored. Otherwise => try to solve the conflict by disambiguating the grammar.

Parser Conflict Resolution

Example: (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                       else single-Command
```

This parse tree?

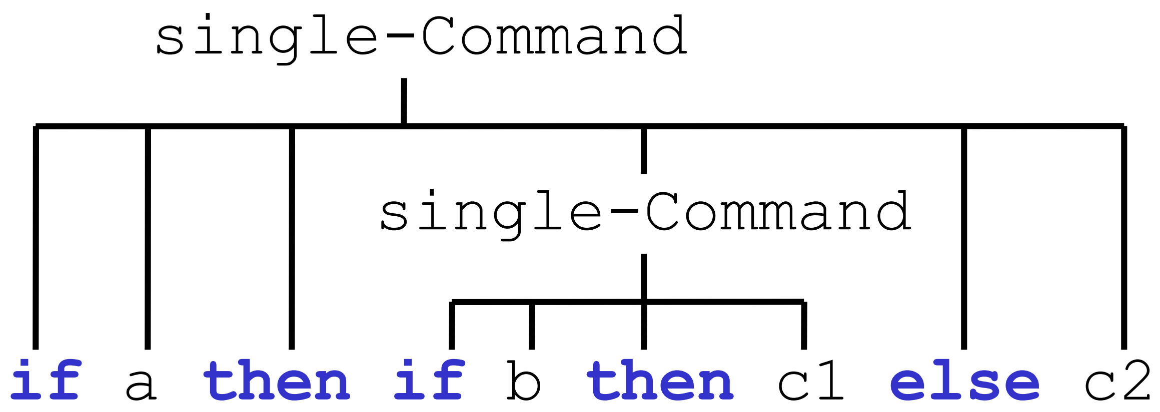


Parser Conflict Resolution

Example: (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                          else single-Command
```

or this one ?



Parser Conflict Resolution

Example: "dangling-else" problem (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                        else single-Command
```

Rewrite Grammar:

```
sC   ::= CsC
       |  OsC
CsC  ::= if E then CsC else CsC
CsC  ::= …
OsC  ::= if E then sC
       |  if E then CsC else OsC
```

Parser Conflict Resolution

Example: "dangling-else" problem (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                           else single-Command
```

LR(1) items (in some state of the parser)

```
sC   ::= if E then sC •  {… else …}
sC   ::= if E then sC •   else sC {…}
```

Shift-reduce conflict!

Resolution rule: shift has priority over reduce.

Q: Does this resolution rule solve the conflict? What is its effect on the parse tree?

Parser Conflict Resolution

There is usually also a default resolution rule for shift-reduce conflicts, for example the rule which appears first in the grammar description has priority.

Reduce-reduce conflicts usually mean there is a real problem with your grammar.

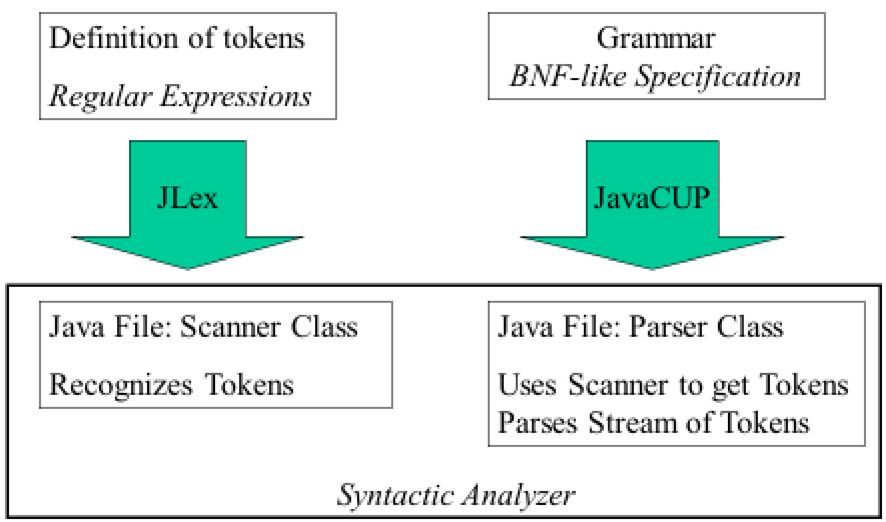
=> You need to fix it! Don't rely on the resolution rule!

Enough background!

- All of this may sound a bit difficult (and it is)
- But it can all be automated!
- Now lets talk about tools
  - CUP (or Yacc for Java)
  - SableCC

Java Cup

- Accepts specification of a CFG and produces an LALR(1) parser (expressed in Java) with action routines expressed in Java
- Similar to yacc in its specification language, but with a few improvements (better name management)
- Usually used together with JLex (or JFlex)



Java Cup Specification Structure

```
java_cup_spec ::= package_spec
                  import_list
                  code_part
                  init_code
                  scan_code
                  symbol_list
                  precedence_list
                  start_spec
                  production_list
```

- What does it mean?
  - Package and import control Java naming
  - Code and init_code allow insertion of code in generated output
  - Scan code specifies how scanner (lexer) is invoked
  - Symbol list and precedence list specify terminal and non-terminal names and their precedence
  - Start and production specify grammar and its start point

Calculator JavaCup Specification (calc.cup)

```
terminal            PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
terminal Integer  NUMBER;
non terminal Integer expr;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
expr  ::= expr PLUS expr
          | expr MINUS expr
          | expr TIMES expr
          | expr DIVIDE expr
          | LPAREN expr RPAREN
          | NUMBER
        ;
```

- Is the grammar ambiguous?
- How can we get PLUS, NUMBER, ...?
  - They are the terminals returned by the scanner.
- How to connect with the scanner?

Ambiguous Grammar Error

- If we enter the grammar

  Expression ::= Expression PLUS Expression;

- without precedence JavaCUP will tell us:

```
Shift/Reduce conflict found in state #4
between Expression ::= Expression PLUS Expression .
and Expression ::= Expression . PLUS Expression
under symbol PLUS
Resolved in favor of shifting.
```

- The grammar is ambiguous!
- Telling JavaCUP that PLUS is left associative helps.

Evaluate the expression

- The previous specification only indicates the success or failure of a parser. No semantic action is associated with grammar rules.
- To calculate the expression, we must add java code in the grammar to carry out actions at various points.
- Form of the semantic action:

```
expr:e1 PLUS expr:e2
 {: RESULT = new Integer(e1.intValue()+ e2.intValue());    :}
```

  - Actions (java code) are enclosed within a pair {:   :}
  - Labels e2, e2: the objects that represent the corresponding terminal or non-terminal;
  - RESULT: The type of RESULT should be the same as the type of the corresponding non-terminals. e.g., expr is of type Integer, so RESULT is of type integer.

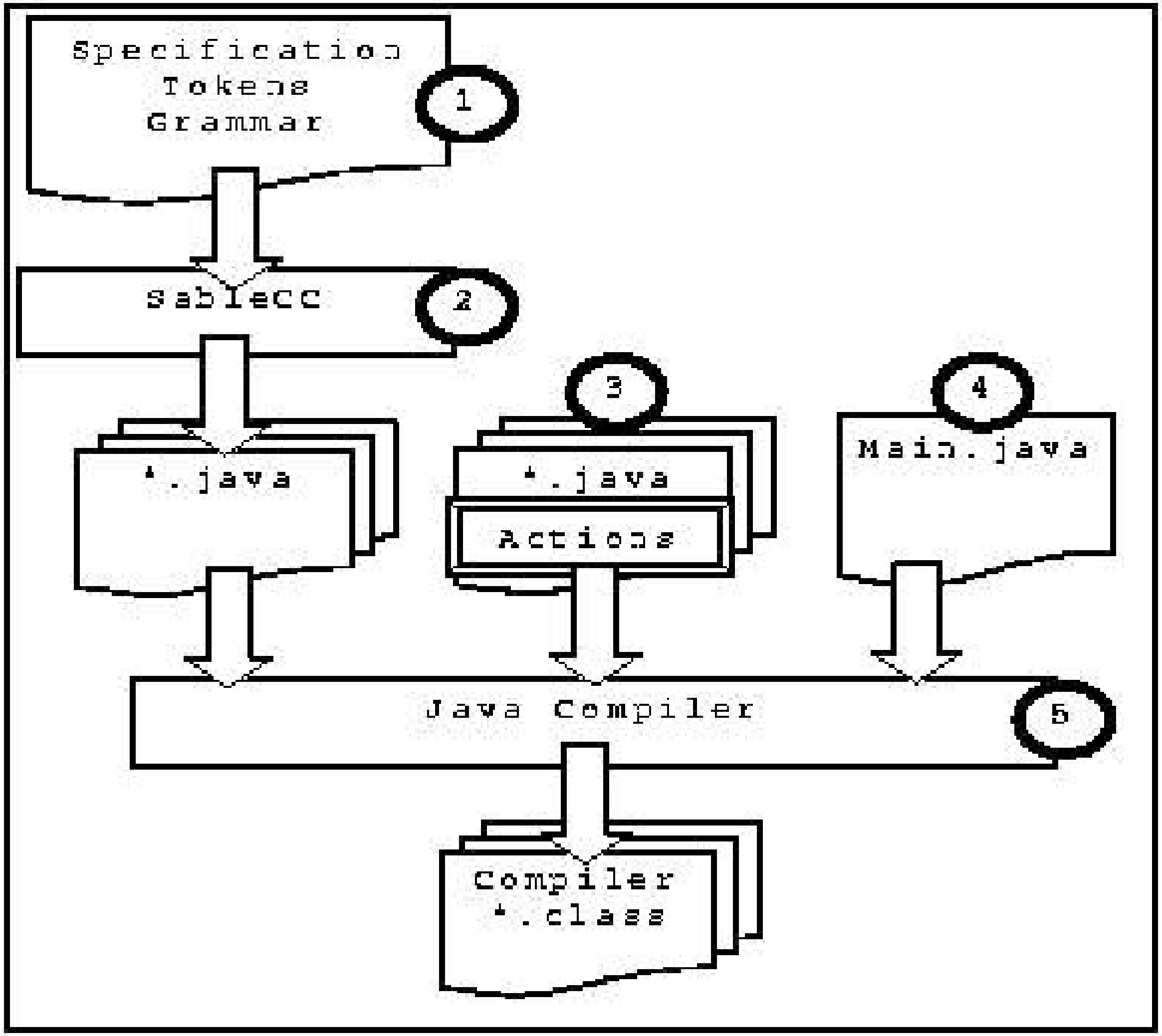
Change the calc.cup

```
terminal           PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
terminal Integer  NUMBER;
non terminal AST expr;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
expr  ::= expr:e1 PLUS expr:e2 {: RESULT = new Computing("+",e1,e2);    :}
        | expr:e1 MINUS expr:e2  {: RESULT = new Computing("-",e1,e2);  :}
        | expr:e1 TIMES expr:e2  {: RESULT = new Computing("*",e1,e2);  :}
        | expr:e1 DIVIDE expr:e2  {: RESULT = new Computing("\",e1,e2);  :}
        | LPAREN expr:e RPAREN {: RESULT = e;    :}
        | NUMBER:e {: RESULT= new IntConsting(e.intValue()); :}
```

SableCC

- Object Oriented compiler framework written in Java
  - There are also versions for C++ and C#
- Front-end compiler compiler like JavaCC and JLex/CUP
- Lexer generator based on DFA
- Parser generator based on LALR(1)
- Object oriented framework generator:
  - Strictly typed Abstract Syntax Tree
  - Tree-walker classes
  - Uses inheritance to implement actions
  - Provides visitors for user manipulation of AST
    - E.g. type checking and code generation

Steps to build a compiler with SableCC



1. Create a SableCC specification file
2. Call SableCC
3. Create one or more working classes, possibly inherited from classes generated by SableCC
4. Create a Main class activating lexer, parser and working classes
5. Compile with Javac

SableCC Example

```
Package Prog
Helpers
  digit = ['0' .. '9'];
  tab = 9;  cr = 13;  lf = 10;
  space = ' ';
  graphic = [[32 .. 127] + tab];

Tokens
  blank = (space | tab | cr | lf)* ;
  comment = '//' graphic* (cr | lf);
  while = 'while';
  begin = 'begin';
  end = 'end';
  do = 'do';
  if = 'if';
  then = 'then';
  else = 'else';
  semi = ';';
  assign = '=';
  int = digit digit*;
  id = ['a'..'z'](['a'..'z']|['0'..'9'])*;

Ignored Tokens
  blank, comment;
```

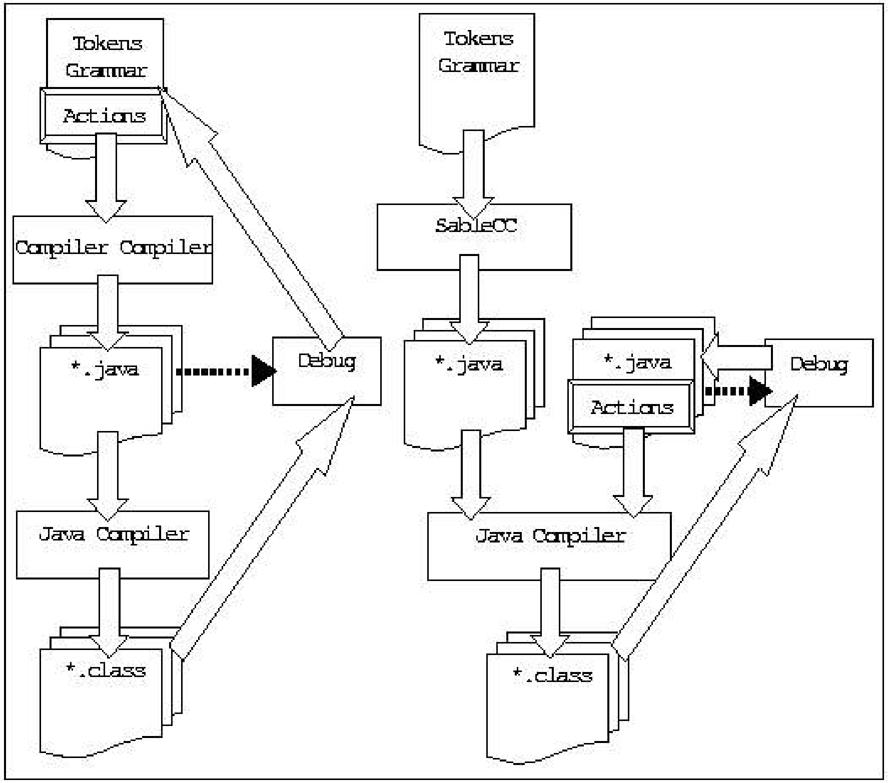
```
Productions
  prog = stmlist;

  stm = {assign} [left:]:id assign [right]:id|
        {while} while id do stm |
        {begin} begin stmlist end |
        {if_then} if id then stm;

stmlist = {stmt} stm |
          {stmtlist} stmlist semi stm;
```

SableCC output

- The *lexer* package containing the Lexer and LexerException classes
- The *parser* package containing the Parser and ParserException classes
- The *node* package contains all the classes defining typed AST
- The *analysis* package containing one interface and three classes mainly used to define AST walkers based on the visitors pattern

JLex/CUP vs. SableCC

- SableCC advantages
  - Automatic AST builder for multi-pass compilers
  - Compiler generator out of development cycle when grammar is stable
  - Easier debugging
  - Access to sub-node by name, not position
  - Clear separation of user and machine generated code
  - Automatic AST pretty-printer
  - Version 3.0 allows declarative grammar transformations



What can you do now in your projects?

- Extract a core of your language
- Define CFG for this core
  – Transform into LL(1)
  – Transform into LALR (probably not necessary)
- Build:
  – Recursive decent parser (and lexer) by hand
  – Try JavaCC and/or ANTLR
  – Try JFlex/CUP
  – Try SableCC
  – (Try other parser tools, e.g. Coco/R, Gold Parser)
- Conclude which one is most appropriate for your project

Languages and Compilers
(SProg og Oversættere)

Lecture 9
Abstract Syntax Trees

Bent Thomsen

Department of Computer Science

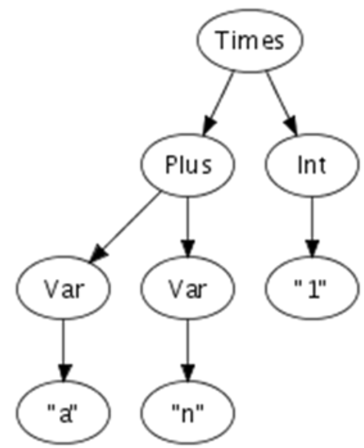
Aalborg University

Learning goals

- To understand the role of the AST in modern compilers
- Knowledge of Attribute Grammars
- Knowledge about single pass vs. multi pas
- Knowledge of different approaches to AST design
- Understand the interplay between CFG and AST
- Be able to design an AST structure
- Knowledge of AST traversal approaches

Remember exercises 2 and 3 from before lecture 1 ?

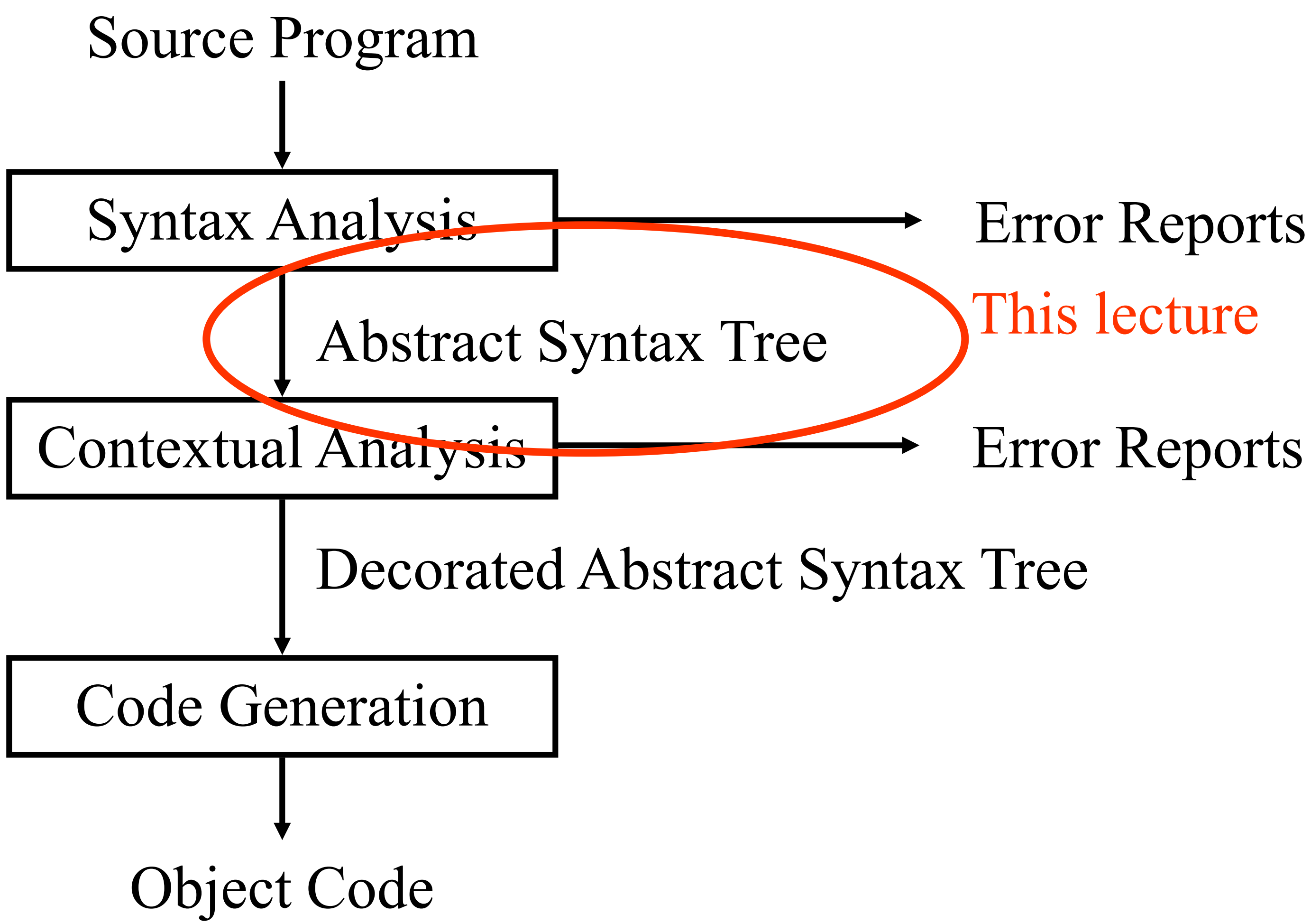
- Write a Java program that implements a data structure for the following tree



- Extend your Java program to traverse the tree depth-first and print out information in nodes and leaves as it goes along.

- Today we shall see several ways of solving this exercise

The "Phases" of a Compiler



Ac in JavaCC with AST

```
AST prog() :
{Prog itsAST = new Prog(new ArrayList<AST >());
 AST dcl;
 AST stm;
}
{(
  dcl = dcl()
  {itsAST.prog.add(dcl);}
  )+
  (stm = stmt()
  {itsAST.prog.add(stm);}
  )*
  {return itsAST;}

}

AST dcl() :
{Token t;}
{
  (< FLOATDCL > t = <ID >)
  {return new FloatDcl(t.image);}
  | (< INTDCL > t = <ID >)
  {return new IntDcl(t.image);}
}

AST stmt() :
{Boolean b = true;
 AST v;
 Computing e = null;
 Token t;
}
{
  (t = < ID ><ASSIGN > v = val() ((e = expr()){b = false;})?)
  {if (b) return v; else { e.child1 = v; return e;}}
| (< PRINT > t = <ID >)
  {return new Printing(t.image);}
}
```

```
AST val() :
{Token t;}
{
  t = < INUM >
  {return new IntConsting(t.image);}
| t = < FNUM >
  {return new FloatConsting(t.image);}
| t= <  ID >
  {return new SymReferencing(t.image);}
}

Computing expr() :
{Boolean b = true;
 AST v;
 Computing e =  null;
}
{
      < PLUS > v = val() (e = expr(){b = false;})?
      {if (b) return new Computing("+",null,v);
      else { e.child1 = v; return new Computing("+",null,e);}}
    | < MINUS > v = val() (e = expr(){b = false;})?
      {if (b) return new Computing("-",null,v);
      else { e.child1 = v; return new Computing("-",null,e);}}

}
```

Action Routines and Attribute Grammars

- Automatic tools can construct lexer and parser for a given context-free grammar
  - *E.g. JavaCC and JLex/CUP (and Lex/Yacc*)
- CFGs cannot describe all of the syntax of programming languages
  - An ad hoc technique is to annotate the grammar with executable rules
  - These rules are known as *action routines*
- Action routines can be formalized *Attribute Grammars*

Semantic Actions and Values

- Semantic actions
  - Associated code sequence that will execute when the production is applied
- Semantic values
  - For production A -> X1…Xn, a semantic value for each symbol
    - Terminals: values originate from the scanner
    - Nonterminals: to compute a value for A based on the values assigned to X1…Xn
      - For yacc Xi: $i A: $0
      - For JavaCUP X:val

Synthesized and Inherited Attributes

- Synthesized attributes
  - Attributes flow from the leaves of a derivation tree toward its root
  - Ex.: evaluating expressions (Fig. 7.1)
  - Better ex.: Inferred Type
- Inherited attributes
  - Attribute values pass from parent to child
  - Ex.: counting the position of each x in a string
  - Better ex.: expected Type

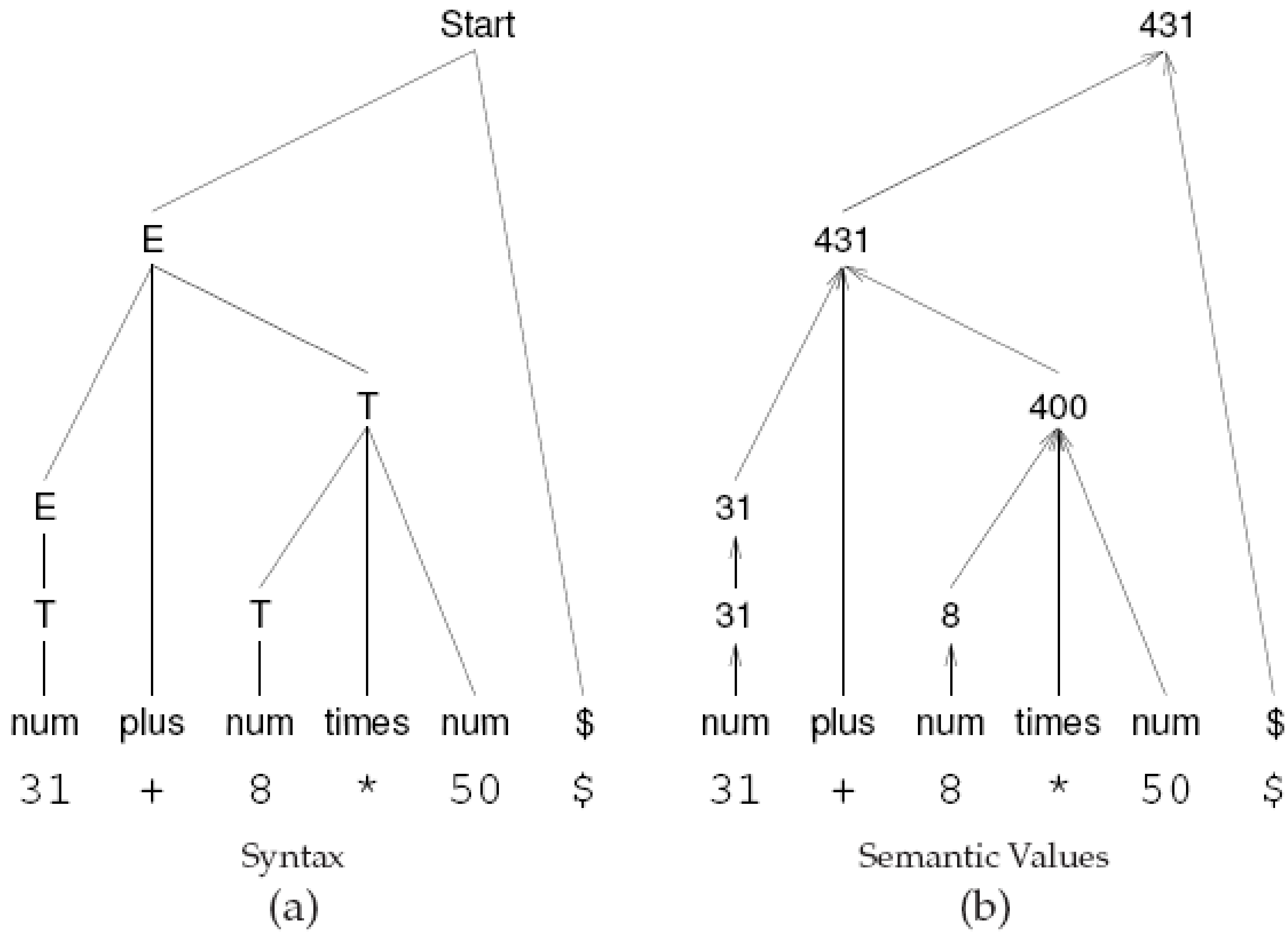


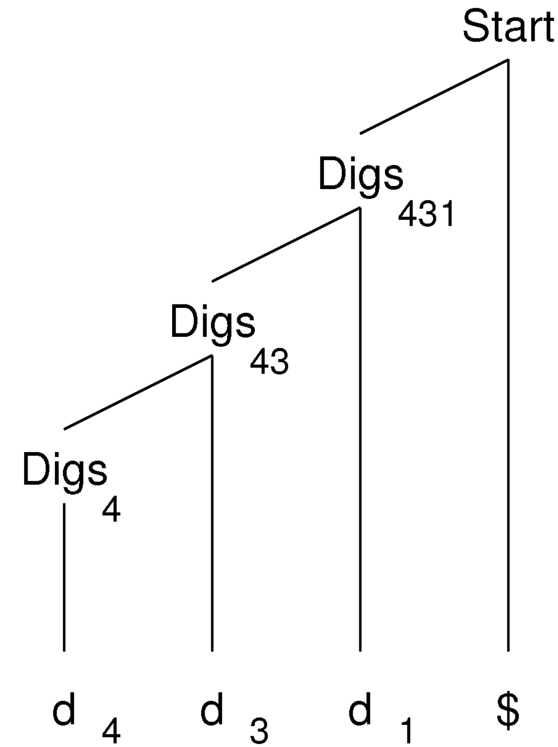
Figure 7.1: (a) Parse tree for the displayed expression;
(b) Synthesized attributes transmit values up the parse tree toward the root.

Example: 4 3 1 $

- Semantic values for nonterminal symbols: computed by semantic actions
- Semantic values for terminal symbols: established by the scanner

1	Start	$\rightarrow$	Digs_{ans} \$
			call PRINT(ans)
2	Digs_{up}	$\rightarrow$	Digs_{below} d_{next}
			$up \leftarrow below \times 10 + next$
3		|	d_{first}
			$up \leftarrow first$

(a)



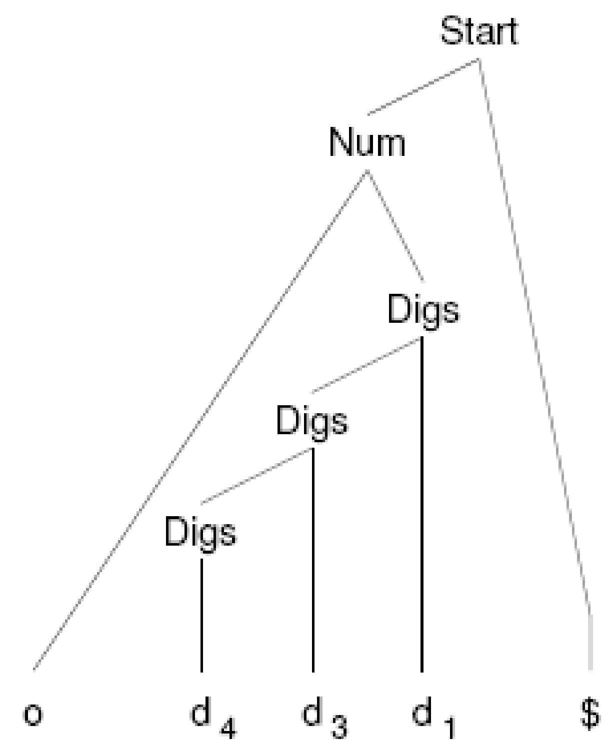
(b)

Figure 7.3: (a) Grammar with semantic actions; (b) Parse tree and propagated semantic values for the input 4 3 1 $.

- Example: o 4 3 1 $ i.e. Base-8 (octal)
  - Problem: the information required at a semantic action is not available from below
    - Semantic actions allowed only on reductions (in bottom up parsers)

1	Start	$\rightarrow$	Num $
2	Num	$\rightarrow$	o Digs
3		|	Digs
4	Digs	$\rightarrow$	Digs d
5		|	d

(a)



(b)

Figure 7.4: (a) Grammar and (b) parse tree for the input o 4 3 1 $.

Rule Cloning

- A similar sequence of input symbols should be treated differently depending on the context
  - Ex.: (Fig. 7.5)
  - Redundancy in productions

1 Start $\rightarrow \text{Num}_{ans}$ \$
  call PRINT(ans)

2 $\text{Num}_{ans} \rightarrow$ o OctDigs_{octans}
  $ans \leftarrow octans$

3 | DecDigs_{decans}
  $ans \leftarrow decans$

4 $\text{DecDigs}_{up} \rightarrow \text{DecDigs}_{below}\ \text{d}_{next}$
  $up \leftarrow below \times 10 + next$

5 | d_{first}
  $up \leftarrow first$

6 $\text{OctDigs}_{up} \rightarrow \text{OctDigs}_{below}\ \text{d}_{next}$
  if $next \geq 8$ ①
  then ERROR("Non-octal digit")
  $up \leftarrow below \times 8 + next$

7 | d_{first}
  if $first \geq 8$ ②
  then ERROR("Non-octal digit")
  $up \leftarrow first$

Figure 7.5: Grammar with cloned productions.

Forcing Semantic Actions

- Introducing *unit productions* of the form A→X
  - Semantic actions can be associated with the reduction of A→X
  - If a semantic action is desired between two symbols Xm and Xn,
    - a production of the form A→λ can be introduced
  - Ex.: (Fig. 7.6)

1	Start	→	Num_{ans} \$
			call PRINT(ans)
2	Num_{ans}	→	SignalOctal Digs_{octans}
			$ans \leftarrow octans$
3		|	SignalDecimal Digs_{decans}
			$ans \leftarrow decans$
4	SignalOctal	→	o
			$base \leftarrow 8$
5	SignalDecimal	→	λ
			$base \leftarrow 10$
6	Digs_{up}	→	Digs_{below} d_{next}
			$up \leftarrow below \times base + next$
7		|	d_{first}
			$up \leftarrow first$

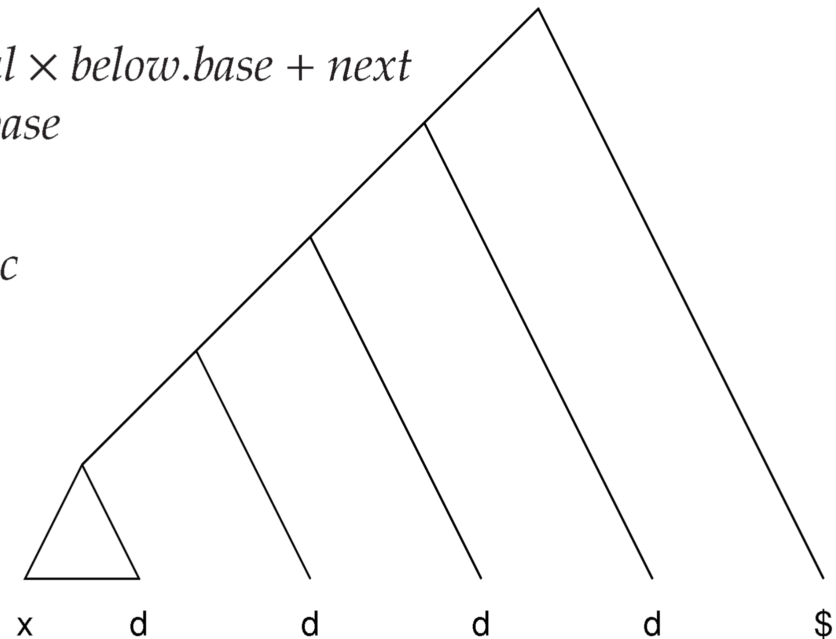
Figure 7.6: Use of λ-rules to force semantic action.

Aggressive Grammar Restructuring

- Reasons to avoid using global variables
  - Grammar rules are often invoked recursively, and the global variables can introduce unwanted interactions
  - Global variables can make semantic actions difficult to write and maintain
  - Global variables may require setting or resetting
- More robust solution
  - Sketch the parse tree without global variables
  - Revise the grammar to achieve the desired parse tree
  - Verify the revised grammar is still suitable for parser construction (e.g. LALR(1))
  - Verify the revised grammar still generates the same language
  - (Fig. 7.8)
    - Keep the base in the semantic values
    - Propagate the value up the parse tree

1	Start	$\rightarrow$	Digs_{ans} \$
			call PRINT($ans.val$)
2	Digs_{up}	$\rightarrow$	$\text{Digs}_{below}\ \text{d}_{next}$
			$up.val \leftarrow below.val \times below.base + next$
			$up.base \leftarrow below.base$
3		|	$\text{SetBase}_{basespec}$
			$up.base \leftarrow basespec$
			$up.val \leftarrow 0$
4	Setbase_{n}	$\rightarrow$	λ
			$n \leftarrow 10$
5		|	$\text{x}\ \text{d}_{num}$
			$n \leftarrow num$

(a)



(b)

Figure 7.8: (a) Grammar that avoids global variables; (b) Parse tree reorganized to facilitate bottom-up attribute propagation.

Top-Down Syntax-Directed Translation

- Using the recursive-descent parsers
- Semantic actions can be written directly into the parser
  - Ex.: Lisp-like expressions (Fig. 7.9)
    - (plus 31 (prod 10 2 20)) $
- Inherited values: parameters passed into a method
- Synthesized values: returned by methods
  - (Fig. 7.10)

```
1  Start  → Value $
2  Value  → num
3         | lparen Expr rparen
4  Expr   → plus Value Value
5         | prod Values
6  Values → Value Values
7         | λ
```

Figure 7.9: Grammar for Lisp-like expressions.

```
procedure START( )
    switch (...)
        case ts.PEEK( ) ∈ { num, lparen }
            ans ← VALUE( )
            call MATCH($)
            call PRINT(ans)                                  (5)
end
function VALUE( ) returns int
    switch (...)
        case ts.PEEK( ) ∈ { num }
            call MATCH(num)
            ans ← num.VALUEOF( )
            return (ans)
        case ts.PEEK( ) ∈ { lparen }
            call MATCH(lparen)
            ans ← EXPR( )
            call MATCH(rparen)
            return (ans)
end
function EXPR( ) returns int
    switch (...)
        case ts.PEEK( ) ∈ { plus }
            call MATCH(plus)
            op1 ← VALUE( )                                   (6)
            op2 ← VALUE( )                                   (7)
            return (op1 + op2)                               (8)
        case ts.PEEK( ) ∈ { prod }
            call MATCH(prod)
            ans ← VALUES(1)                                  (9)
            return (ans)
end
function VALUES(thusfar) returns int
    case ts.PEEK( ) ∈ { num, lparen }
        next ← VALUE( )                                      (10)
        ans ← VALUES(thusfar × next)                         (11)
        return (ans)
    case ts.PEEK( ) ∈ { rparen }
        return (thusfar)                                     (12)
end
```

Figure 7.10: Recursive-descent parser with semantic actions. The variable *ts* is the token stream produced by the scanner.

General structure

Production: X -> a Y Z

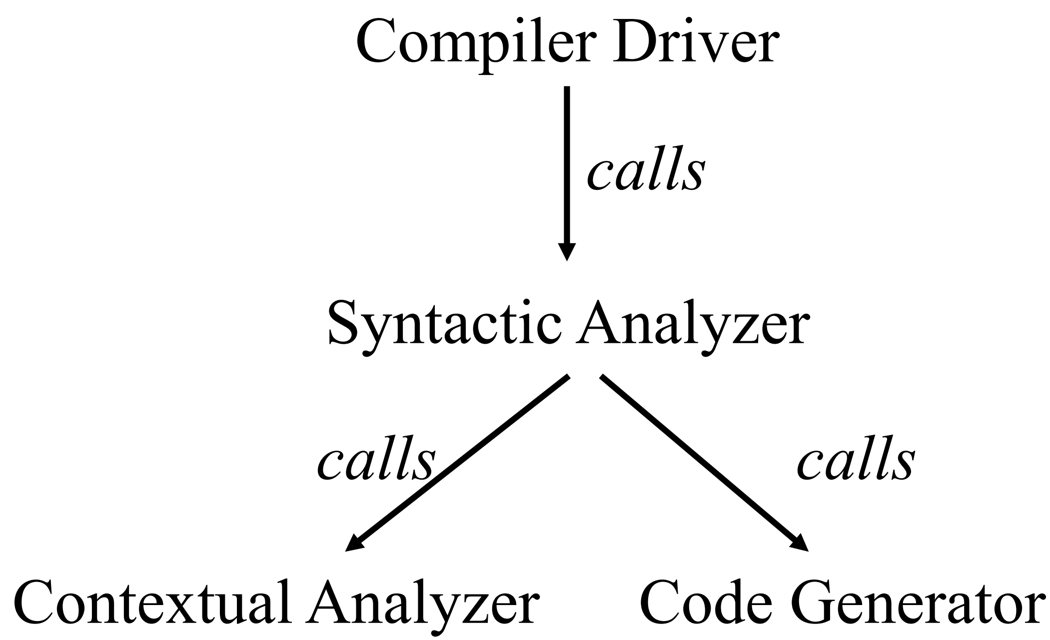
```
parseX(); checkToken(a); parseY(); parseZ();
```

```
parseX(); CODE checktoken(a); CODE parseY(); CODE parseZ();
```

Single Pass Compiler

A single pass compiler makes a single pass over the source text, parsing, analyzing and generating code all at once.

Dependency diagram of a typical Single Pass Compiler:



Ac Single Pass Compiler

Production: X -> a Y Z

```
parseX(); checkToken(a); parseY(); parseZ();
```

```
parseX(); CODE checktoken(a); CODE parseY(); CODE parseZ();
```

CODE include code for typechecking, codegeneration, …

Ac Parser (without action code)

```
public void Stmt() {
    if (ts.peek() == ID) {
        expect(ID);
        expect(ASSIGN);
        Val();
        Expr();
    }
    else if (ts.peek() == PRINT) {
        expect(PRINT);
        expect(ID);
    }
    else error("expected id or print");

}

public void Expr() {
    if (ts.peek() == PLUS) {
        expect(PLUS);
        Val();
        Expr();
    }
    else if (ts.peek() == MINUS) {
        expect(MINUS);
        Val();
        Expr();

    }
    else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
        // Do nothing for lambda-production
    }
    else error("expected plus, minus, id, print, or eof");

}
```

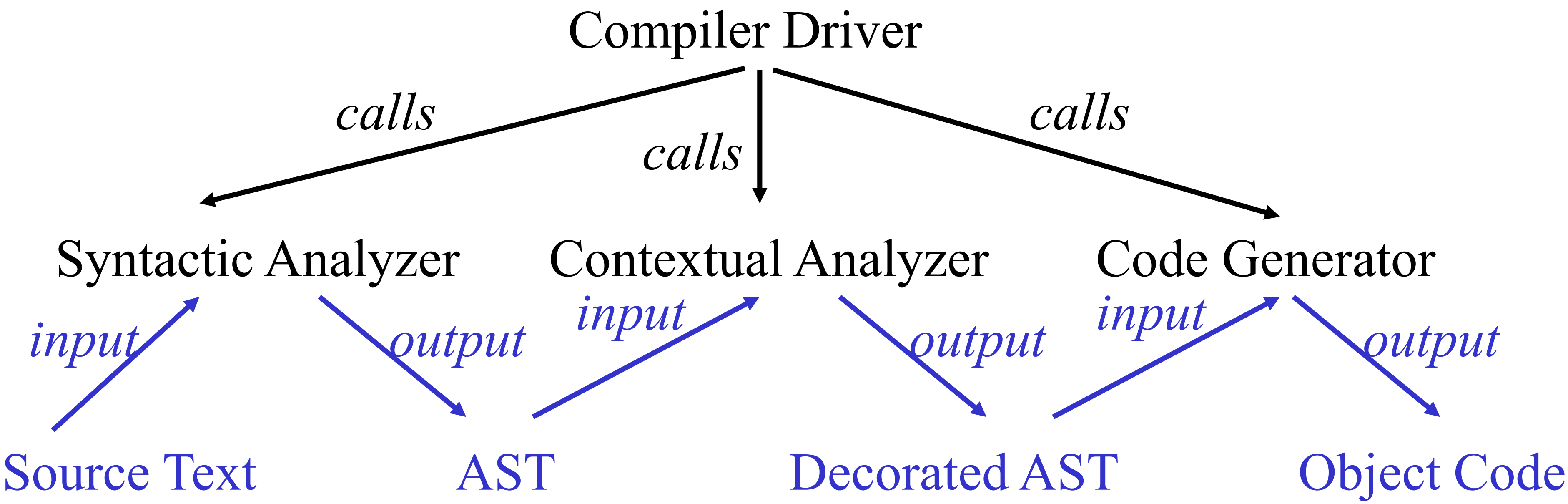
Ac Parser for Single Pass Comp.

```
public void Stmt() {
    if (ts.peek() == ID) {
        Token t = expect(ID);
        expect(ASSIGN);
        int tid = SymbolTable.get(t.val);
        int vt = Val();
        if (tid == FLTTYPE && vt == INTTYPE) {emit(" 5 k "); vt = FLTTYPE;};
        int et = Expr(vt);
        if (tid == INTTYPE && et == FLTTYPE) error("Illegal type conversion");
        emit(" s");
        emit(t.val);
        emit(" 0 k ");
    }
    else if (ts.peek() == PRINT) {
        expect(PRINT);
        Token t = expect(ID);
        emit("l");
        emit(t.val);
        emit(" p ");
        emit("si ");
    }
    else error("expected id or print");

}

public int Expr(int te) {
    int ty = -1;
    if (ts.peek() == PLUS) {
        expect(PLUS);
        int vt = Val();
        if (te == FLTTYPE && vt == INTTYPE) {emit(" 5 k "); vt = FLTTYPE;};
        int et = Expr(vt);
        emit(" + ");
        ty = et;
    }
    else if (ts.peek() == MINUS) {
        expect(MINUS);
        int vt = Val();
        if (te == FLTTYPE && vt == INTTYPE) {emit(" 5 k "); vt = FLTTYPE;};
        int et = Expr(vt);
        emit(" - ");
        ty = et;

    }
    else if (ts.peek() == ID || ts.peek() == PRINT || ts.peek() == EOF) {
        // Do nothing for lambda-production
        ty = te;
    }
```

Multi Pass Compiler

A multi pass compiler makes several passes over the program. The output of a preceding phase is stored in a data structure and used by subsequent phases.

Dependency diagram of a typical Multi Pass Compiler:



Abstract Syntax Trees

- The central data structure for all post-parsing activities
  - AST must be concise
  - AST must be sufficiently flexible
- Concrete vs. abstract trees
  - (Fig. 7.3 & 7.4)
  - (Fig. 7.11)

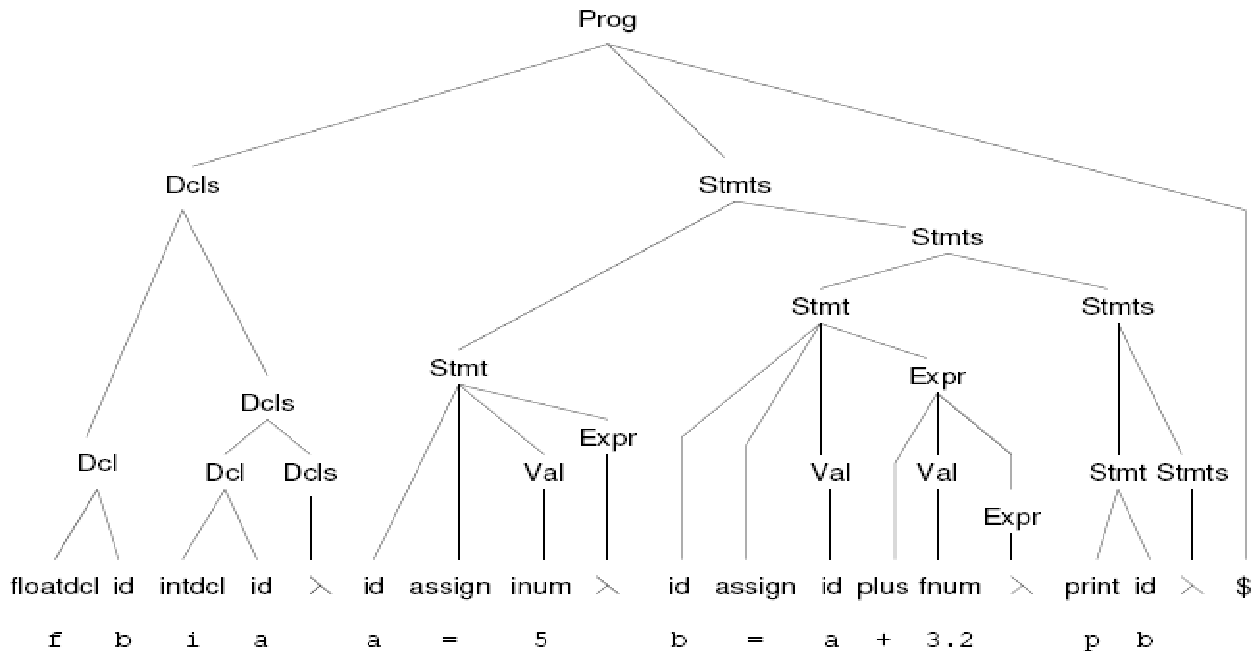


Figure 2.4: An ac program and its parse tree.

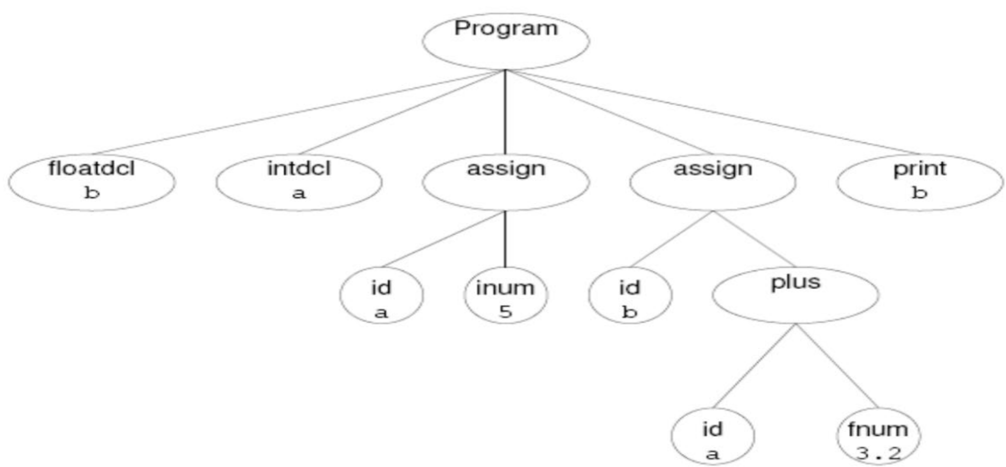


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Abstract Syntax Trees

- Like a parse tree, but with some details omitted
- Note we could use the parse tree
  - but often, the parse tree keeps unnecessary details
  - E.g. SableCC AST is equivalent to the parse tree if you do not specify grammar transformation rules!
  - ANTLR4 gives you the parse tree !
    - You have to convert this to an AST yourself

An Efficient AST Data Structure

- Considering
  - AST is typically constructed bottom-up
  - Lists of siblings are typically generated by recursive rules
  - Some AST nodes have a fixed number of children, but some may require an arbitrarily large number of children
- (Fig. 7.12)

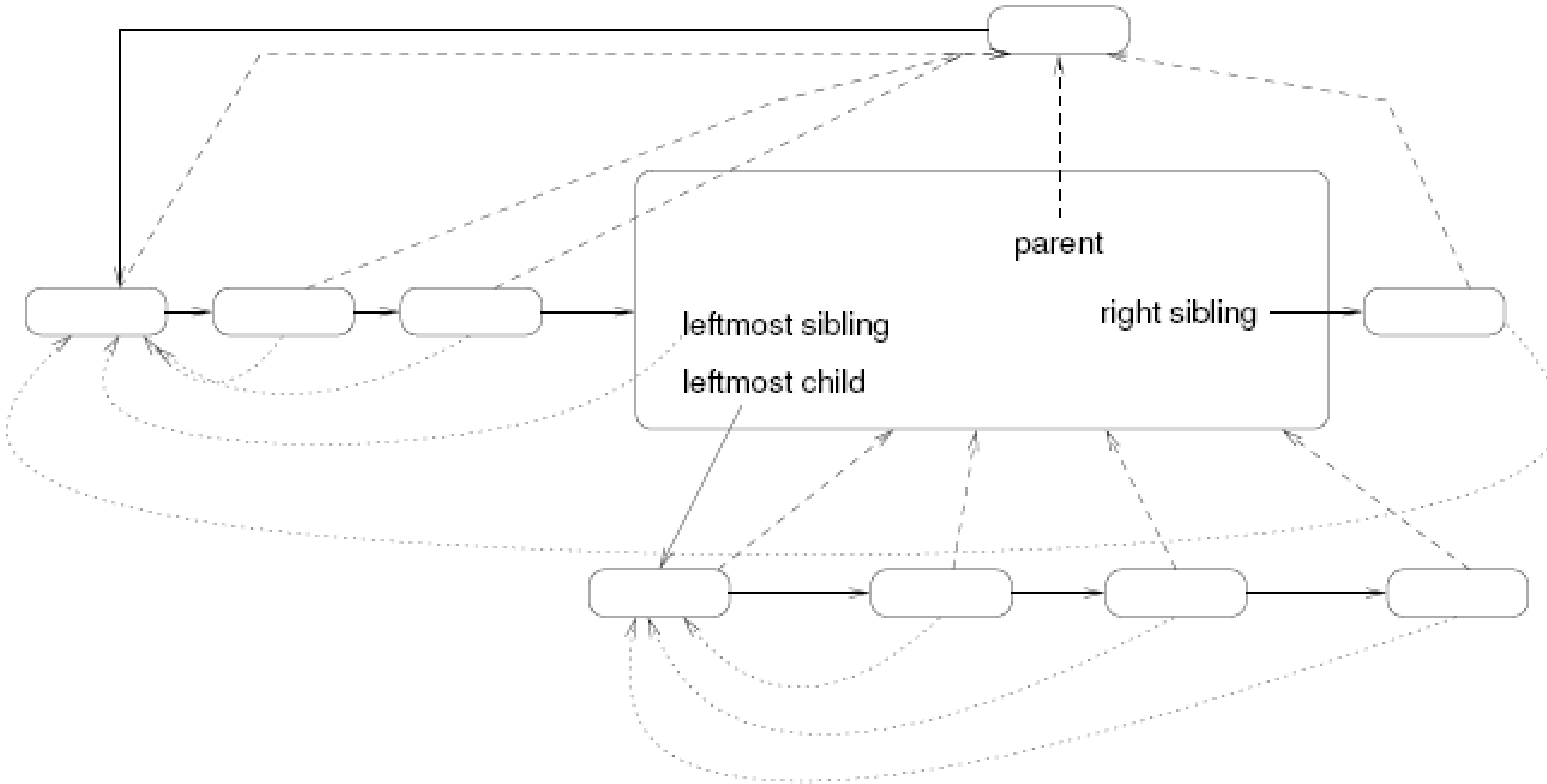


Figure 7.12: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

```
/*  Assert: y ≠ null                                                  */
function makeSiblings( y ) returns Node
    /*  Find the rightmost node in this list                          */
    xsibs ← this
    while xsibs.rightSib ≠ null do xsibs ← xsibs.rightSib
    /*  Join the lists                                                */
    ysibs ← y.leftmostSib
    xsibs.rightSib ← ysibs
    /*  Set pointers for the new siblings                             */
    ysibs.leftmostSib ← xsibs.leftmostSib
    ysibs.parent ← xsibs.parent
    while ysibs.rightSib ≠ null do
        ysibs ← ysibs.rightSib
        ysibs.leftmostSib ← xsibs.leftmostSib
        ysibs.parent ← xsibs.parent
    return (ysibs)
end

/*  Assert: y ≠ null                                                  */
function adoptChildren( y ) returns Node
    if this.leftmostChild ≠ null
    then this.leftmostChild.makeSiblings( y )
    else
        ysibs ← y.leftmostSib
        this.leftmostChild ← ysibs
        while ysibs ≠ null do
            ysibs.parent ← this
            ysibs ← ysibs.rightSib
end
```

Figure 7.13: Methods for building an AST.

```
1  Start → Stmt $
2  Stmt  → id assign E
3        | if lparen E rparen Stmt else Stmt fi
4        | if lparen E rparen Stmt fi
5        | while lparen E rparen do Stmt od
6        | begin Stmts end
7  Stmts → Stmts semi Stmt
8        | Stmt
9  E     → E plus T
10       | T
11 T     → id
12       | num
```

Figure 7.14: Grammar for a simple language.

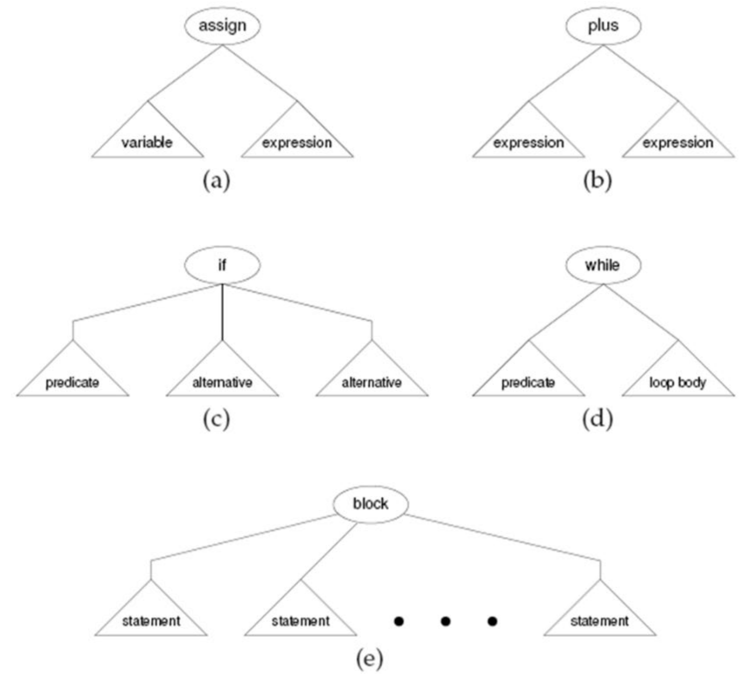


Figure 7.15: AST structures: A specific node is designated by an ellipse. Tree structure of arbitrary complexity is designated by a triangle.

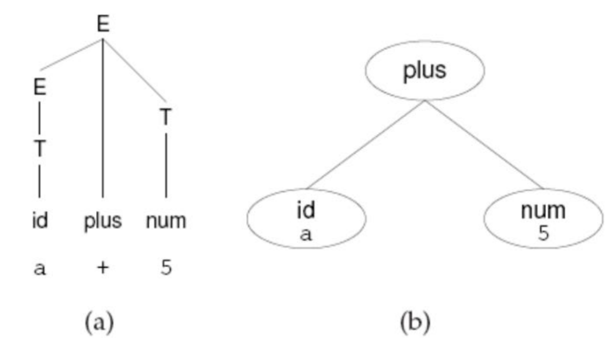


Figure 7.16: (a) Derivation of a + 5 from E;
(b) Abstract representation of a + 5.

1	Start	$\rightarrow$	Stmt$_{ast}$ \$	
			return (ast)	(13)
2	Stmt$_{result}$	$\rightarrow$	id$_{var}$ assign E$_{expr}$	
			$result \leftarrow$ MAKEFAMILY(assign$, var, expr)$	(14)
3		|	if lparen E$_p$ rparen Stmt$_s$ fi	
			$result \leftarrow$ MAKEFAMILY(if$, p, s,$ MAKENODE( ))	(15)
4		|	if lparen E$_p$ rparen Stmt$_{s1}$ else Stmt$_{s2}$ fi	
			$result \leftarrow$ MAKEFAMILY(if$, p, s1, s2)$	(16)
5		|	while lparen E$_p$ rparen do Stmt$_s$ od	
			$result \leftarrow$ MAKEFAMILY(while$, p, s)$	(17)
6		|	begin Stmts$_{list}$ end	
			$result \leftarrow$ MAKEFAMILY(block$, list)$	(18)
7	Stmts$_{result}$	$\rightarrow$	Stmts$_{sofar}$ semi Stmt$_{next}$	
			$result \leftarrow sofar.$MAKESIBLINGS$(next)$	(19)
8		|	Stmt$_{first}$	
			$result \leftarrow first$	(20)
9	E$_{result}$	$\rightarrow$	E$_{e1}$ plus T$_{e2}$	
			$result \leftarrow$ MAKEFAMILY(plus$, e1, e2)$	(21)
10		|	T$_e$	
			$result \leftarrow e$	(22)
11	T$_{result}$	$\rightarrow$	id$_{var}$	
			$result \leftarrow$ MAKENODE(var)	(23)
12		|	num$_{val}$	
			$result \leftarrow$ MAKENODE(val)	(24)

Figure 7.17: Semantic actions for grammar in Figure 7.14.

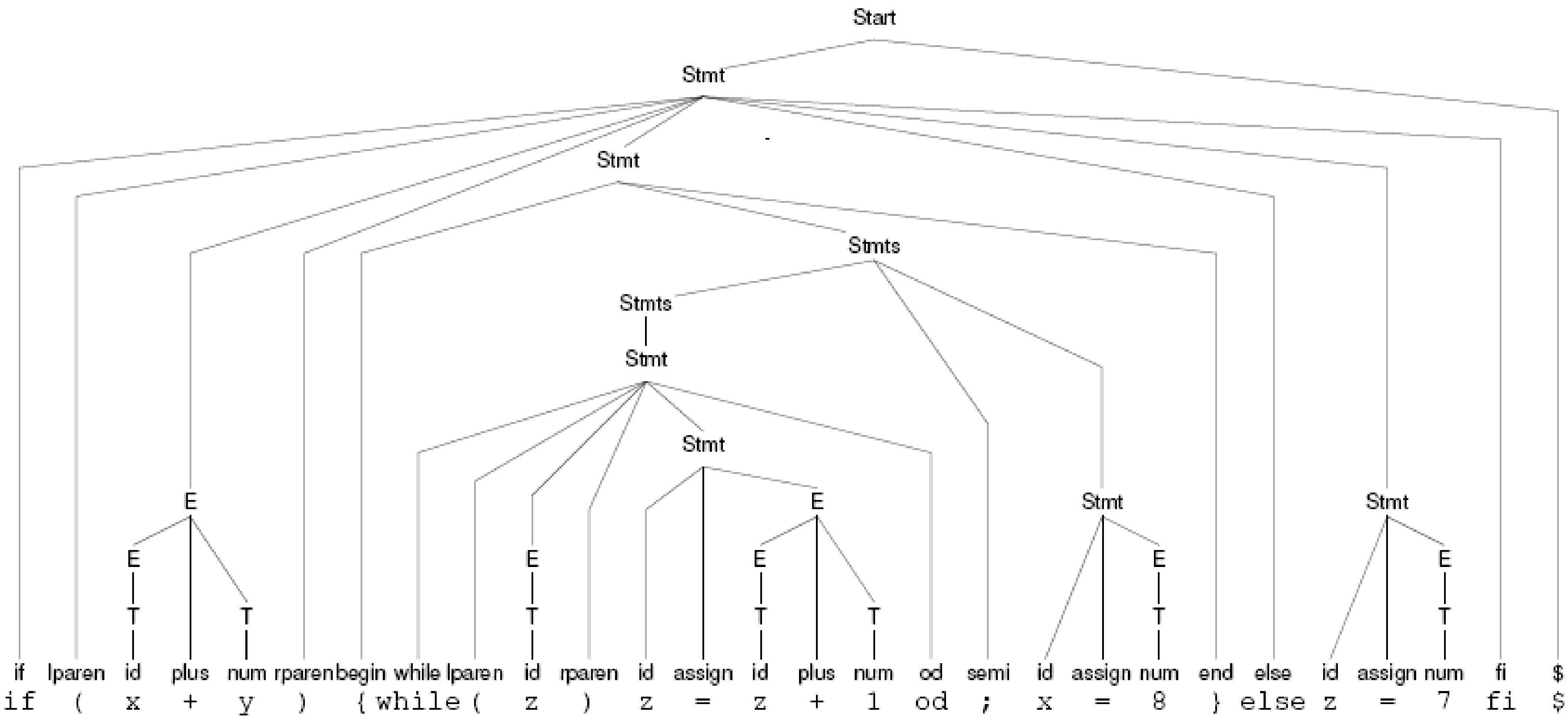


Figure 7.18: Concrete syntax tree.

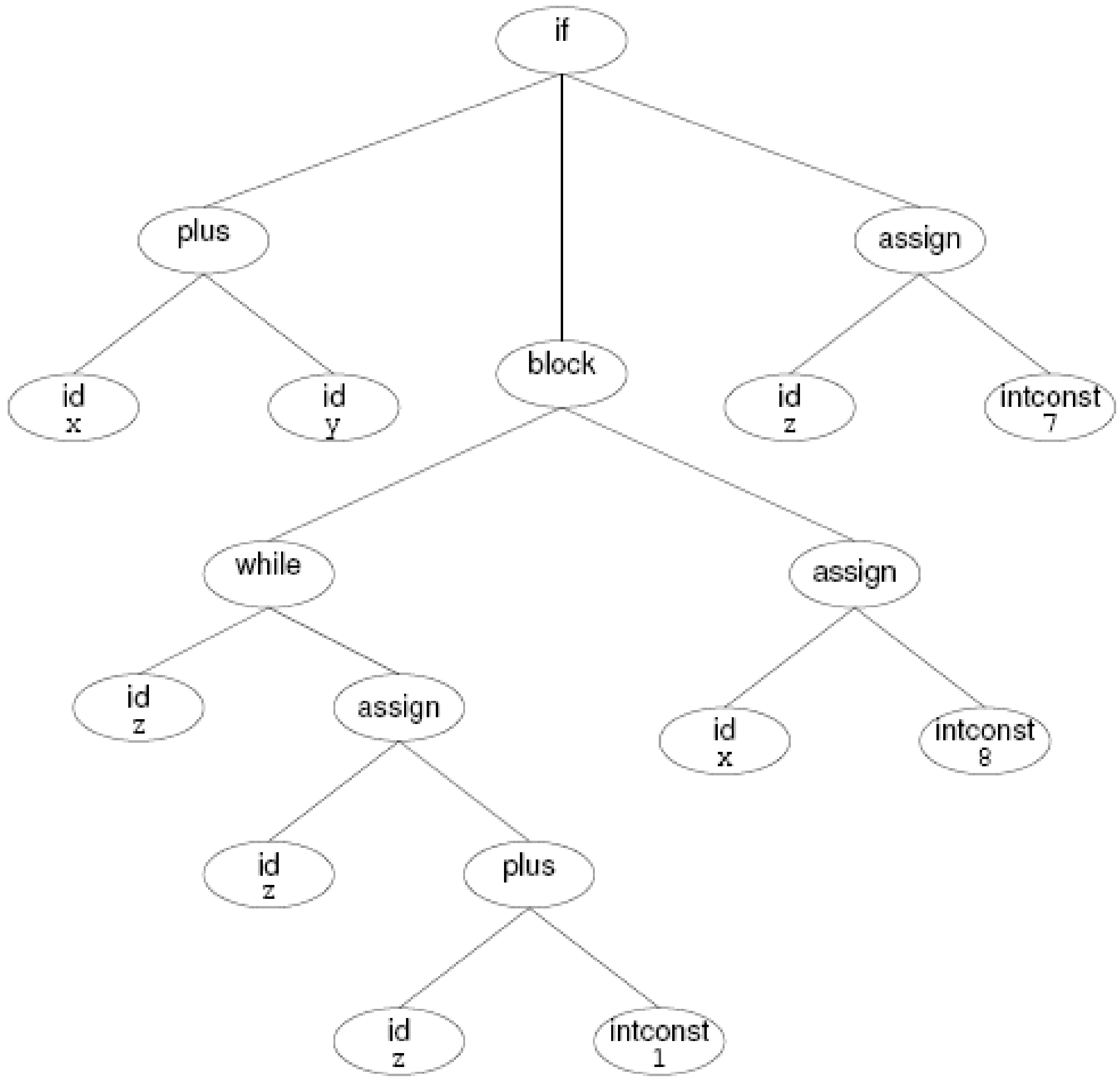


Figure 7.19: AST for the parse tree in Figure 7.18.

AST Design and Construction

- Important forces that influence the design of an AST
  - It should be possible to *unparse* an AST
    - i.e. reconstitute the program from an AST
    - AST must hold sufficient information
  - The implementation of an AST should be decoupled from the essential information represented within the AST
  - Different views from different phases of a compiler

- Process of the design of an appropriate AST structure
  - An unambiguous grammar for L is devised
  - An AST for L is devised
  - Semantic actions are placed in the grammar to construct the AST
  - Passes of the compiler are designed using the visitor design pattern

This is what Fischer et. Al. Says –
however sometimes an ambigous grammer may be the right thing
For devising the AST – just think of SableCC version 3.0

```
class Visitor
    /*    Generic visit                                   */
    procedure visit( AbstractNode n )                        (28)
        n.accept( this )                                     (29)
    end
end
class TypeChecking extends Visitor                           (30)
    procedure visit( IfNode i )
    end
    procedure visit( PlusNode p )
    end
    procedure visit( MinusNode m )
    end
end

class IfNode extends AbstractNode
    procedure accept( Visitor v )                            (31)
        v.visit( this )
    end
    ...
end
class PlusNode extends AbstractNode
    procedure accept( Visitor v )                            (32)
        v.visit( this )                                      (33)
    end
    ...
end
class MinusNode extends AbstractNode
    procedure accept( Visitor v )                            (34)
        v.visit( this )
    end
    ...
end
```

Figure 7.23: Visitor pattern

Pause

Abstract Syntax Trees

- The examples of AST design and construction in Fischer et. Al. are some what abstract
- Now we will look at very concrete example taken from Brown&Watt's book: Programming Language Processors in Java:
  - MiniTriangle language
  - how to represent AST as data structures.
  - how to refine a recursive decent parser to construct an AST data structure.

You may need more than one Grammar

- Concrete Syntax
  - The grammar we use as specification for building a parser
  - Must be unambiguous
  - Usually LL(1), LL(*) or LALR(1)

- Lexical elements (Syntax given as Regular Expressions)
  - Identifiers e.g. Id := [a-z]([a-z]|[0-9])*
  - Keywords (or reserved words)

- Abstract Syntax
  - To communicate the essentials of the language
  - To serve in the formal specification of the semantics
  - May be ambiguous
  - To serve as design pattern for AST

Concrete Syntax of Commands

```
single-Command
        ::= V-name := Expression
          | Identifier ( Expression )
          | if Expression then single-Command
                           else single-Command
          | while Expression do single-Command
          | let Declaration in single-Command
          | begin Command end
Command ::= single-Command
            | Command ; single-Command
```

Abstract Syntax of Commands

Command		
::=	V-name **:=** Expression	AssignCmd
|	Identifier **(** Expression **)**	CallCmd
|	**if** Expression **then** Command **else** Command	IfCmd
|	**while** Expression **do** Command	WhileCmd
|	**let** Declaration **in** Command	LetCmd
|	Command **;** Command	SequentialCmd

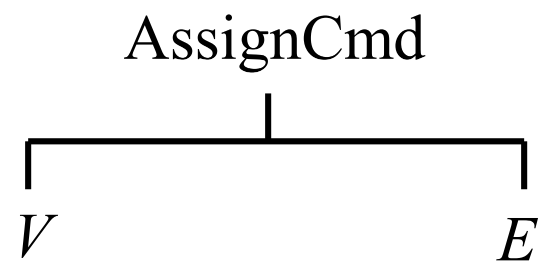
Even more Abstract Syntax of Commands

```
Command
  ::= V-name Expression                    AssignCmd
    | Identifier Expression                CallCmd
    | Expression Command Command           IfCmd
    | Expression Command                   WhileCmd
    | Declaration Command                  LetCmd
    | Command Command                      SequentialCmd
```

The possible form of AST structures can be completely determined by the AST grammar

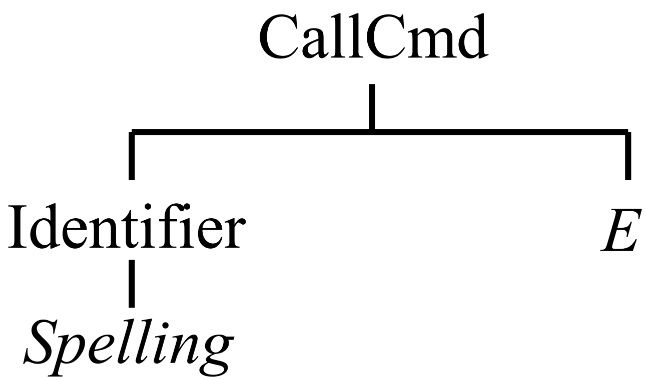
AST Representation: Possible Tree Shapes

```
Command ::= VName := Expression        AssignCmd
          | ...
```



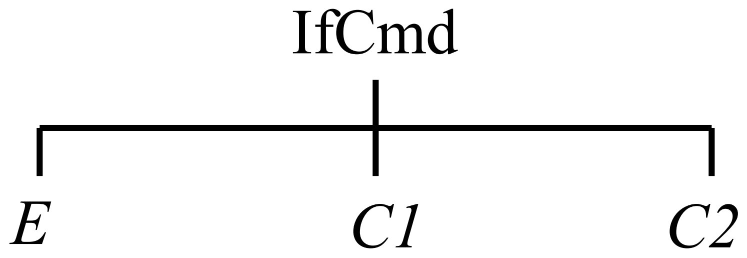
AST Representation: Possible Tree Shapes

```
Command ::=
      ...
      | Identifier ( Expression )          CallCmd
      ...
```



AST Representation: Possible Tree Shapes

```
Command ::=
      ...
      | if Expression then Command
                       else Command        IfCmd
      ...
```



AST Representation: Java Data Structures

Example: Java classes to represent Mini Triangle AST's

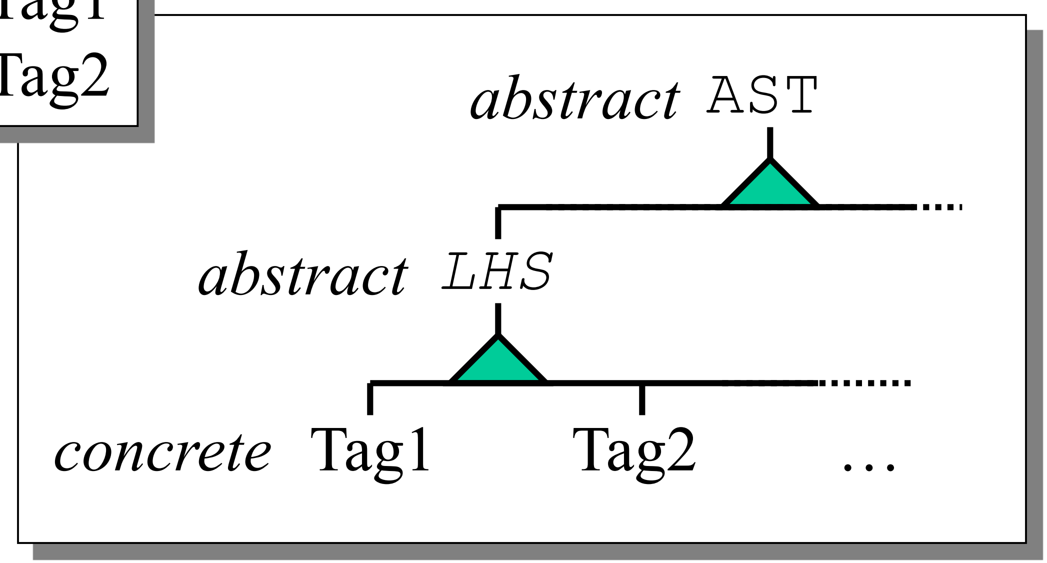
1) A common (abstract) super class for all AST nodes

```
public abstract class AST { ... }
```

2) A Java class for each "type" of node.

- abstract as well as concrete node types

```
LHS ::= ...          Tag1
      | ...          Tag2
```



Example: Mini Triangle Commands ASTs

```
Command
 ::= V-name := Expression                      AssignCmd
   | Identifier ( Expression )                 CallCmd
   | if Expression then Command
                    else Command               IfCmd
   | while Expression do Command               WhileCmd
   | let Declaration in Command                LetCmd
   | Command ; Command                         SequentialCmd
```

```
public abstract class Command extends AST { ... }

public class AssignCommand extends Command { ... }
public class CallCommand extends Command { ... }
public class IfCommand extends Command { ... }
etc.
```

Example: Mini Triangle Command ASTs

```
Command ::= V-name := Expression        AssignCmd
    | Identifier ( Expression )         CallCmd
    | ...
```

```
public class AssignCommand extends Command {
   public Vname V;          // assign to what variable?
   public Expression E;     // what to assign?
   ...
}

public class CallCommand extends Command {
   public Identifier I;     //procedure name
   public Expression E;     //actual parameter
   ...
}
...
```

AST Terminal Nodes

```
public abstract class Terminal extends AST {
   public String spelling;
   ...
}

public class Identifier extends Terminal { ... }

public class IntegerLiteral extends Terminal { ... }

public class Operator extends Terminal { ... }
```

AST Construction

First, every concrete AST class needs a constructor.

Examples:

```
public class AssignCommand extends Command {
   public Vname V;          // Left side variable
   public Expression E;     // right side expression
   public AssignCommand(Vname V; Expression E) {
      this.V = V; this.E=E;
   }
   ...
}

public class Identifier extends Terminal {
   public class Identifier(String spelling) {
      this.spelling = spelling;
   }
   ...
}
```

AST Construction

We will now show how to refine our recursive descent parser to actually construct an AST.

Remember:

N ::= *X*

```
private void parseN() {
    parse X
}
```

AST Construction

We will now show how to refine our recursive descent parser to actually construct an AST.

```
N  ::=  X
```

```
private N parseN() {
  N itsAST;
  parse X at the same time constructing itsAST
  return itsAST;
}
```

Example: “**Generation” of parseCommand**

Command ::= single-Command (; single-Command)*

```
private void parseCommand() {
  parseSingleCommand();
  while (currentToken.kind==Token.SEMICOLON) {
     acceptIt();
     parseSingleCommand();
  }
}
```

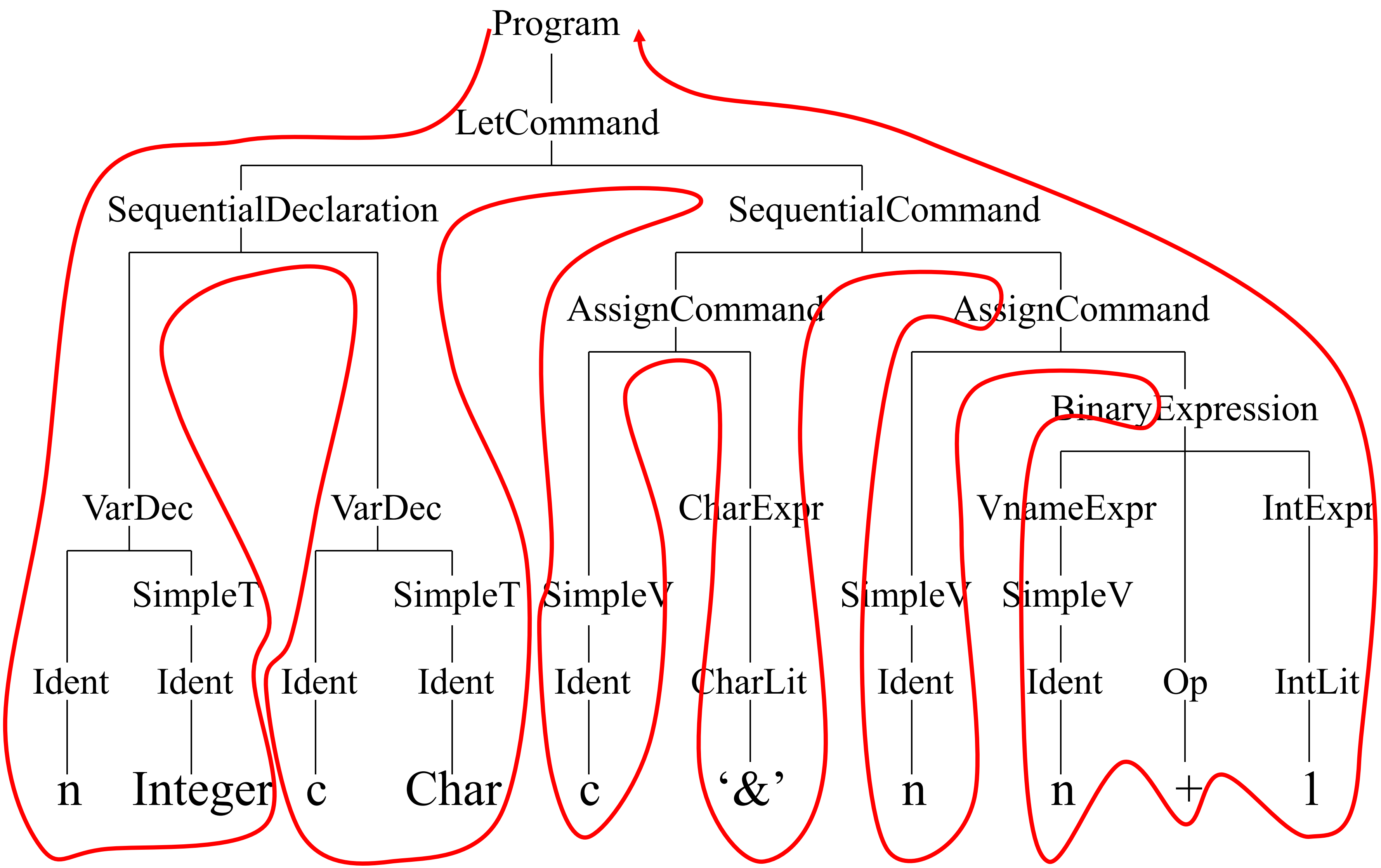
Example: Construction of Mini Triangle ASTs

Command ::= single-Command (; single-Command)*

```
// AST-generating version
private Command parseCommand() {
  Command itsAST;
  itsAST = parseSingleCommand();
  while (currentToken.kind==Token.SEMICOLON) {
     acceptIt();
     Command extraCmd = parseSingleCommand();
     itsAST = new SequentialCommand(itsAST,extraCmd);
  }
  return itsAST;
}
```

Contextual Analysis

Identification and type checking are combined into a depth-first traversal of the AST.



Depth-First Traversal

Depth-first traversal depends on the structure of the AST - it depends on the number and kind of descendants of each node. Organize it as a collection of functions: analyze*NodeType*

```
analyzeProgram(Program P) {
   … analyzeCommand(P.C) … }
```

```
analyzeIfCommand(IfCommand C) {
   … analyzeExpression(C.E) …
   … analyzeCommand(C.C1)… analyzeCommand(C.C2)… }
```

Depth-First Traversal

It turns out (later in the course) that code generation also requires a traversal of the AST. So we expect the code generator to be organized similarly:

```
generateProgram(Program P) {
   … generateCommand(P.C) … }
```

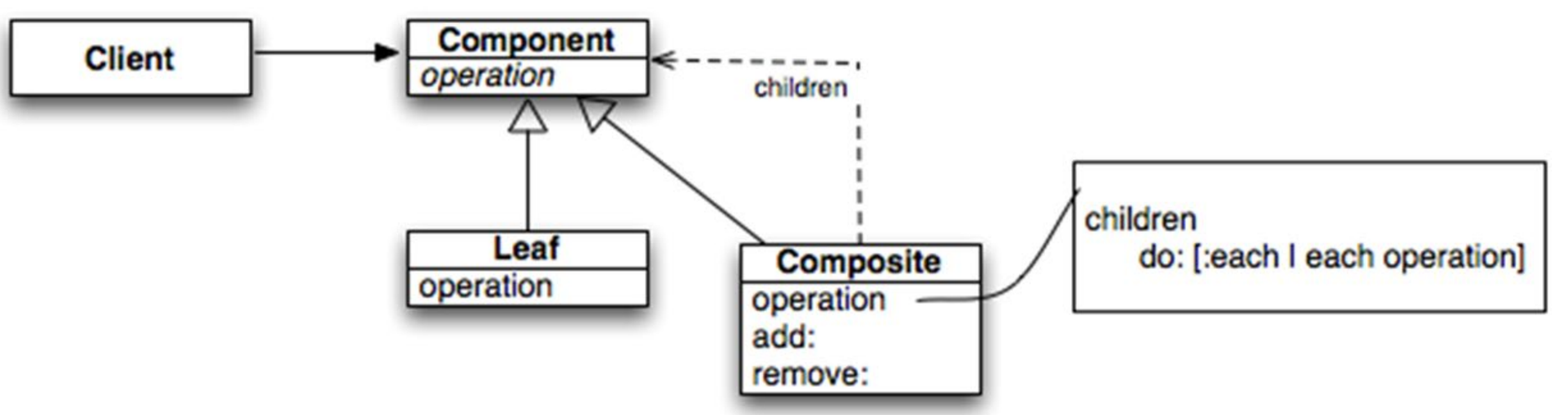
```
generateIfCommand(IfCommand C) {
   … generateExpression(C.E) …
   … generateCommand(C.C1)… generateCommand(C.C2)… }
```

Implementing Tree Traversal

- "Traditional" OO approach
- Visitor approach
  - GOF
  - Using static overloading
  - Reflective
  - (dynamic)
  - (SableCC style)
- "Functional" approach
- Active patterns in Scala (or F#)
- (Aspect oriented approach)

Implementing Tree Traversal: Traditional

- "Traditional" OO approach add a method to each class, so for each node in the AST we have a method that knows how to traverse its children.
- Note the AST is a composit
  - thus we can use the composit pattern
  - Composite lets clients treat individual objects and compositions of objects uniformly
-



ac traditional OO AST traversal

```java
package acASTtraditionalOO;

import java.util.ArrayList;;

public class Prog extends AST {

    ArrayList<AST> prog;

    Prog(ArrayList<AST> prg){
        prog = prg;
    }


    public void prettyprint(){
        for(AST ast : prog){
            ast.prettyprint();
        };
        System.out.println();
    }

    public void symbolTableFilling() {
        // TODO Auto-generated method stub
        for(AST ast : prog){
            ast.symbolTableFilling();
        };

    }

    public void typeChecking() {
        // TODO Auto-generated method stub
        for(AST ast : prog){
            ast.typeChecking();
        };

    }

    public void codeGeneration() {
        // TODO Auto-generated method stub
        for(AST ast : prog){
            ast.codeGeneration();
        };
        System.out.println(code);

    }

}
```

```java
package acASTtraditionalOO;

public class Computing extends AST {
    String operation;
    AST child1;
    AST child2;

    Computing(String op, AST ch1, AST ch2){
        child1 = ch1;
        child2 = ch2;
        operation = op;

    }


    public void prettyprint(){
        child1.prettyprint();
        System.out.print(operation);
        child2.prettyprint();
        }

    public void symbolTableFilling() {
        child1.symbolTableFilling();
        child2.symbolTableFilling();

    }

    public void typeChecking() {
        child1.typeChecking();
        child2.typeChecking();
        int m = generalize(child1.type,child2.type);
        child1 = convert(child1,m);
        child2 = convert(child2,m);
        type = m;
    }

    public void codeGeneration() {
        child1.codeGeneration();
        child2.codeGeneration();
        emit(operation);

    }
}
```

Implementing Tree Traversal: Traditional

- "Traditional" OO approach add a method to each class, so for each node in the AST we have a method that knows how to traverse its children.
- Note the AST is a composit, thus we can use the composit pattern

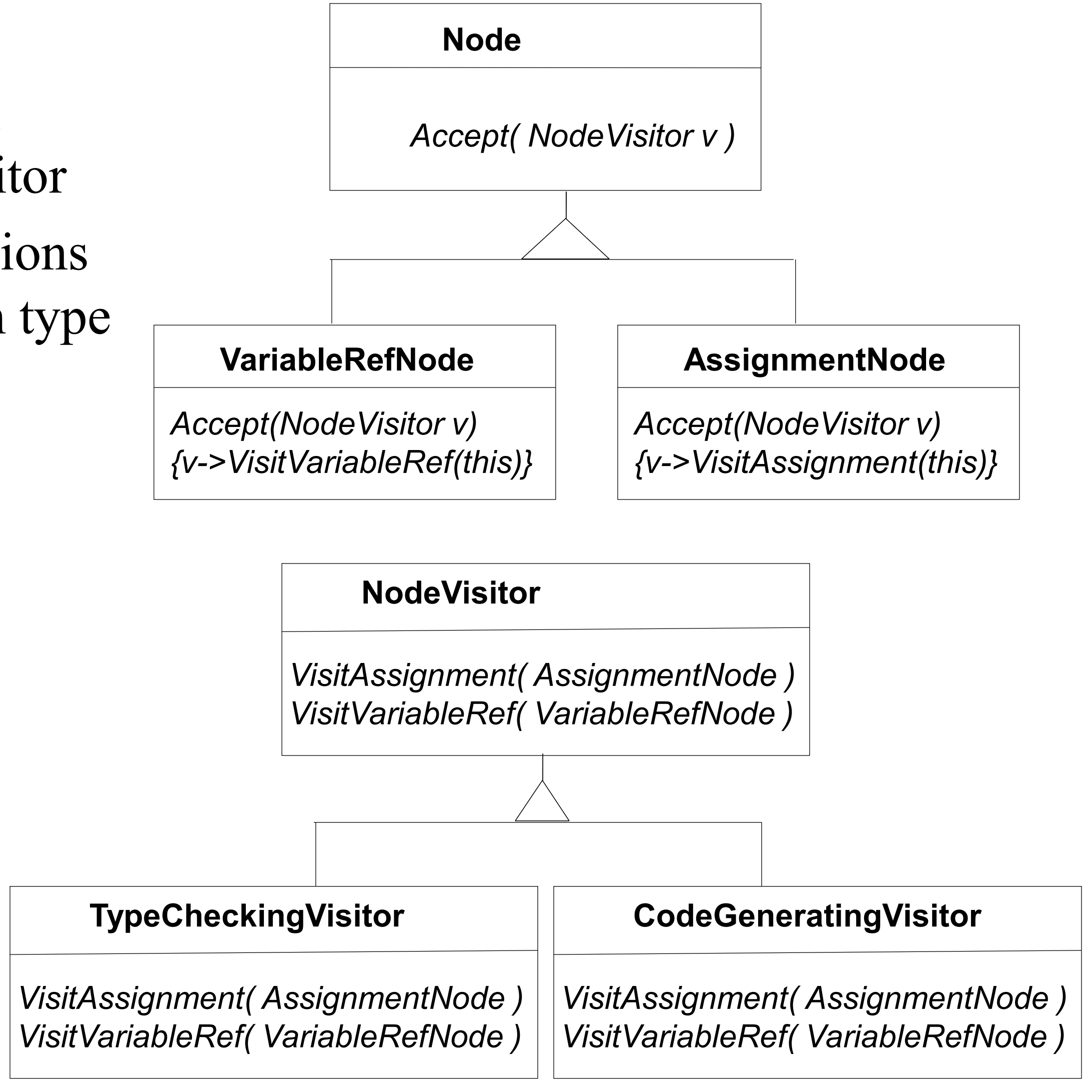
- Scatters code over a large number of classes
- Requires recompilation of AST classes each time a method needs changing
- Could be preferable as long as we are changing the AST often.
- Solution could later be refactored to Visitor pattern

Implementing Tree Traversal: Visitor

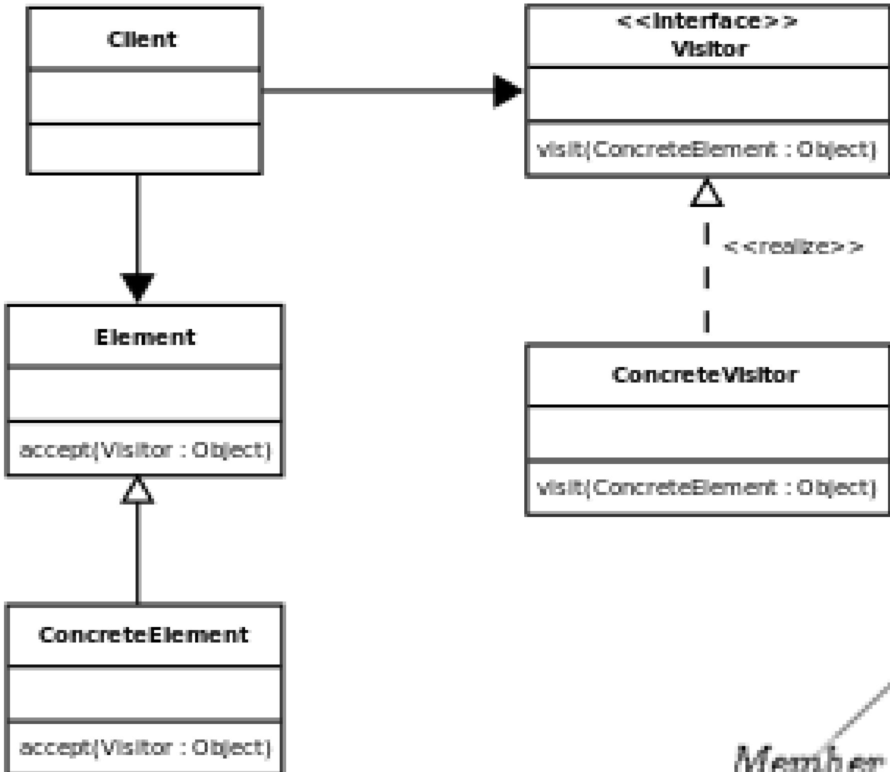
- Solution using Visitor:
  - Visitor is an interface or an abstract class that has a different method for each type of object on which it operates
  - Each operation is a subclass of Visitor and overloads the type-specific methods
  - Objects that are operated on, accept a Visitor and call back their type-specific method passing themselves as operands
  - Object types are independent of the operations that apply to them
  - New operations can be added without modifying the object types

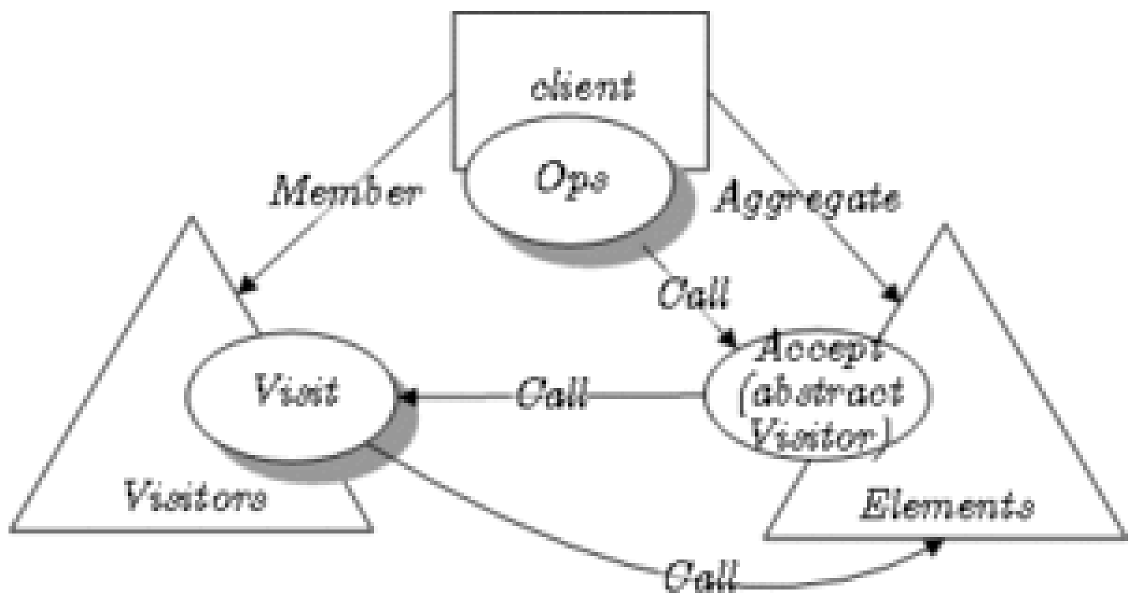
Visitor Solution

- Nodes accept visitors and call appropriate method of the visitor
- Visitors implement the operations and have one method for each type of node they visit



Double Dispatch





AST.java

```java
package acASTVisitor;

import java.util.Hashtable;

public abstract class AST {

    public final static int
    FLTTYPE    = 0,
    INTTYPE    = 1;

    public static Hashtable<String,Integer> SymbolTable = new Hashtable<String,Integer>();

    AST(){
        //for(int ch = 'a'; ch <= 'z'; ch++){AST.SymbolTable.put("" + ch,null);};
    }

    public Integer type = null;


    public abstract void accept(Visitor v);

}

```

Computing.java

```java
package acASTVisitor;

public class Computing extends AST {
    String operation;
    AST child1;
    AST child2;

    Computing(String op, AST ch1, AST ch2){
        child1 = ch1;
        child2 = ch2;
        operation = op;

    }

    public void accept(Visitor v){v.visit(this);}


}

```

Visitor.java

```java
package acASTVisitor;

public abstract class Visitor {
    public void visit(AST n){
        //System.out.println ("In  AST visit\t"+n);

        n.accept(this);
    }

    abstract void visit(Assigning n);
    abstract void visit(Computing n);
    abstract void visit(ConvertingToFloat n);
    abstract void visit(FloatConsting n);
    abstract void visit(IntConsting n);
    abstract void visit(Printing n);
    abstract void visit(Prog n);
    abstract void visit(SymDeclaring n);
    abstract void visit(FloatDcl n);
    abstract void visit(IntDcl n);
    abstract void visit(SymReferencing n);



}

```

TypeChecker.java

```java
public class TypeChecker extends Visitor {

    @Override
    void visit(Assigning n) {
        // TODO Auto-generated method stub
        n.child1.accept(this);
        int m = AST.SymbolTable.get(n.id);
        int t = generalize(n.child1.type,m);
        n.child1 = convert(n.child1,m);
        n.type = t;
    }

    @Override
    void visit(Computing n) {
        // TODO Auto-generated method stub
        n.child1.accept(this);
        n.child2.accept(this);
        int m = generalize(n.child1.type,n.child2.type);
        n.child1 = convert(n.child1,m);
        n.child2 = convert(n.child2,m);
        n.type = m;
    }

    void visit(ConvertingToFloat n){
        n.child.accept(this);
        n.type = AST.FLTTYPE;
    }

    @Override
    void visit(FloatConsting n) {
        // TODO Auto-generated method stub
        n.type = AST.FLTTYPE;

    }

    @Override
    void visit(IntConsting n) {
```

Flavours of the Visitor Pattern

- GOF style as on previous slides
  - acASTGOFVisitor
- Reflective Visitor
  - acASTreflective
- Exploiting static overloading
  - acASTVisitor

Implementing Tree Traversal: `instanceof`

Another possibility is to use a "functional" approach and implement a case-analysis on the class of an object.

```
Type check(Expr e) {
   if (e instanceof IntLitExpr)
      return representation of type int
   else if (e instanceof BoolLitExpr)
      return representation of type bool
   else if (e instanceof EqExpr) {
      Type t = check(((EqExpr)e).left);
      Type u = check(((EqExpr)e).right);
      if (t == representation of type int &&
          u == representation of type int)
         return representation of type bool
   ...
```

ac with functional AST traversal

```
public static void functionalprettyprinter (AST ast){
    if (ast instanceof Assigning) {
        Assigning n = (Assigning)ast;
        System.out.print(n.id + " = " );
        functionalprettyprinter(n.child1);
        System.out.print(" ");

    }
    else if (ast instanceof Computing) {
        Computing n = (Computing)ast;
        functionalprettyprinter(n.child1);
        System.out.print(" " + n.operation + " ");
        functionalprettyprinter(n.child2);

    }
    else if(ast instanceof ConvertingToFloat){
        ConvertingToFloat n = (ConvertingToFloat) ast;
        System.out.print(" i2f ");
        functionalprettyprinter(n.child);
    }
    else if(ast instanceof FloatConsting) {
        // TODO Auto-generated method stub
        FloatConsting n = (FloatConsting) ast;
        System.out.print(n.val);

    }
    else if(ast instanceof IntConsting) {
        // TODO Auto-generated method stub
        IntConsting n = (IntConsting) ast;
        System.out.print(n.val);

    }
    else if(ast instanceof Printing) {
        // TODO Auto-generated method stub
        Printing n = (Printing) ast;
        System.out.print("p " + n.id + " ");

    }
```

Implementing Tree Traversal: `instanceof`

This approach leads to a messy nested **`if`**, which can't be converted into a **`switch`** because Java has no mechanism for switching on the class of an object.

Also this technique is not very object-oriented: instead of explicitly using **`instanceof`**, we prefer to arrange for analysis of an object's class to be done via the built-in mechanisms of overloading and dynamic method dispatch.

Scala active patterns

```
sealed abstract class AST
case class Prog(prog:List[AST]) extends AST
case class Assigning(id:String,child1:AST) extends AST
case class Computing(operation:String,child1:AST,child2:AST) extends AST
case class ConvertingToFloat(child:AST) extends AST
case class Printing(id:String) extends AST
case class FloatConsting(v:String) extends AST
case class FloatDcl(id:String) extends AST
case class Intconsting(v:String) extends AST
case class IntDcl(id:String) extends AST
case class SymDeclaring(id:String) extends AST
case class SymReferencing(id:String) extends AST

def prettyprint(t: AST): void = t match {
  case Prog(prog) => prog.map(prettyprint)
  case Assigning(id,child1) => print(id + " = ");prettyprint(child1);print(" ")
  case Computing(op, ch1,ch2) => prettyprint(ch1);print(" " + op + " ")
  case Converting(ch) => print(" i2f ");prettyprint(ch)
  case Printing(id) => print("p " + id + " ")
  case FloatConsting(v) => print(v)
  case FloatDcl(id) => print("f " + id + " ")
  case IntConsting(v) => print(v)
  case IntDcl(id) => print("f " + id + " ")
  case SymReferencing(id) => print(id)
  }
```

Summary

- The AST is a central data structure in modern compilers
  - Generic very general AST structure
  - Designed based on (Abstract) grammar
- Parser builds AST
  - Action code, e.g. JavaCC, CUP/Yacc/C#CUP (, ANTLR)
  - Done by tool, e.g. SableCC, JavaCC+JJT or JBT (, ANTLR)
- AST traversal
  - Traditional OO
  - Visitor Pattern
  - Functional style

What can you do in your project now?

- Start deciding on an AST design for your compiler
  - Generic vs. Abstract Syntax based (classic OOP)
  - Experiment with AST traversal strategies
- Compare approaches
  - By hand
  - By tool

Languages and Compilers (SProg og Oversættere)

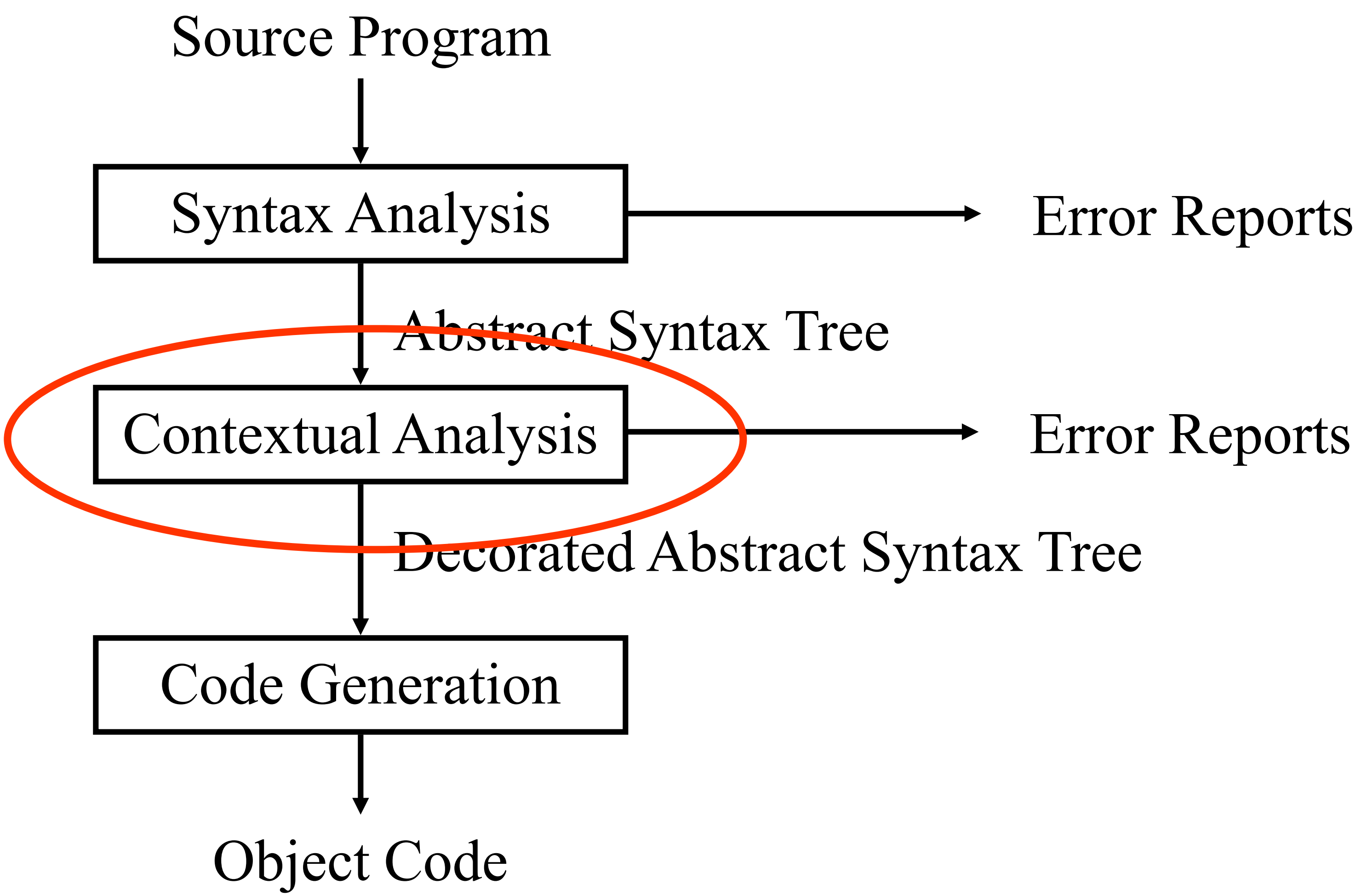
Lecture 10
Scopes and Symbol Tables

Bent Thomsen
Department of Computer Science
Aalborg University

Learning Goals

- Understand the purpose of the Contextual Analysis phase of the compiler
- Knowledge about scope and type rules
- Knowledge about Symbol Tables
- Knowledge about strategies for implementing this phase

The “Phases” of a Compiler



Programming Language Specification

- A language specification has (at least) three parts:
  - Syntax of the language: usually formal: EBNF
  - **Contextual constraints:**
    - **scope rules**
      - » **often written in English, but can be formal**
      - » **(see chapter 6 on p. 86-93 in Transitions and Trees)**
    - **type rules**
      - » **formal or informal**
      - » **See chapter 13 on p.185-210 in Transitions and Trees)**
  - Semantics:
    - defined by the implementation
    - informal descriptions in English
    - formal using operational or denotational semantics
      - » See Transitions and Trees

Contextual Constraints

Syntax rules alone are not enough to specify the format of well-formed programs.

Example 1:

```
let const m~2;
in  m + x
```

Undefined! → Scope Rules

Example 2:

```
let const m~2 ;
    var   n:Boolean
in begin
   n := m<4;
   n := n+1
end
```

Type error! → Type Rules

Scope Rules

Scope rules regulate visibility of identifiers. They relate every **applied occurrence** of an identifier to a **binding occurrence**

Example 1

Binding occurrence

```
let const m~2;
    var   r:Integer
in
    r := 10*m
```

Applied occurrence

Example 2:

?

```
let const m~2
in  m + x
```

Terminology:

Static binding vs. *dynamic binding*

Static scope/block structured scope vs. dynamic scope

Implicit vs. explicit binding *(see p. 86-93 in Transitions and Trees)*

Type Rules

- In order to "tame" the behaviour of programs we can make more or less restrictive type rules
- The validity of these rules is controlled by the type cheking algorithm
- Details depend upon the type system
  - Type systems can be very complicated
    - Lets look at them later
      - Simple type system (next lecture)
      - More complex type systems (later lecture)

Type Rules

Type rules regulate the expected types of arguments and types of returned values for the operations of a language.

Examples

Type rule of **<** :

E1 **<** *E2* is type correct and of type **Boolean**
if *E1* and *E2* are type correct and of type **Integer**

Type rule of **while**:

while *E* **do** *C* is type correct
if *E* of type **Boolean** and *C* type correct

Terminology:

Static typing vs. *dynamic typing*

See Chapter 13 in Trans. & Trees

[SUBS$_{\text{EXP}}$]	$\dfrac{E \vdash e_1 : \mathsf{Int} \quad E \vdash e_2 : \mathsf{Int}}{E \vdash e_1 - e_2 : \mathsf{Int}}$	[NUM$_{\text{EXP}}$]	$E \vdash n : \mathsf{Int}$
[ADD$_{\text{EXP}}$]	$\dfrac{E \vdash e_1 : \mathsf{Int} \quad E \vdash e_2 : \mathsf{Int}}{E \vdash e_1 + e_2 : \mathsf{Int}}$	[VAR$_{\text{EXP}}$]	$\dfrac{E(x) = T}{E \vdash x : T}$
[MULT$_{\text{EXP}}$]	$\dfrac{E \vdash e_1 : \mathsf{Int} \quad E \vdash e_2 : \mathsf{Int}}{E \vdash e_1 * e_2 : \mathsf{Int}}$	[PAREN$_{\text{EXP}}$]	$\dfrac{E \vdash e_1 : T}{E \vdash (e_1) : T}$
[EQUAL$_{\text{EXP}}$]	$\dfrac{E \vdash e_1 : T \quad E \vdash e_2 : T}{E \vdash e_1 = e_2 : \mathsf{Bool}}$		
[AND$_{\text{EXP}}$]	$\dfrac{E \vdash e_1 : \mathsf{Bool} \quad E \vdash e_2 : \mathsf{Bool}}{E \vdash e_1 \wedge e_2 : \mathsf{Bool}}$	[NEG$_{\text{EXP}}$]	$\dfrac{E \vdash e_1 : \mathsf{Bool}}{E \vdash \neg e_1 : \mathsf{Bool}}$

Table 13.3 *Type rules for* ***Bump*** *expressions*

[SKIP$_{\text{STM}}$]	$E \vdash \texttt{skip} : \mathsf{ok}$
[ASS$_{\text{STM}}$]	$\dfrac{E \vdash x : T \quad E \vdash a : T}{E \vdash x := a : \mathsf{ok}}$
[IF$_{\text{STM}}$]	$\dfrac{E \vdash e : \mathsf{Bool} \quad E \vdash S_1 : \mathsf{ok} \quad E \vdash S_2 : \mathsf{ok}}{E \vdash \texttt{if } e \texttt{ then } S_1 \texttt{ else}; S_2 : \mathsf{ok}}$
[WHILE$_{\text{STM}}$]	$\dfrac{E \vdash e : \mathsf{Bool} \quad E \vdash S : \mathsf{ok}}{E \vdash \texttt{while } e \texttt{ do } S : \mathsf{ok}}$
[COMP$_{\text{STM}}$]	$\dfrac{E \vdash S_1 : \mathsf{ok} \quad E \vdash S_2 : \mathsf{ok}}{E \vdash S_1; S_2 : \mathsf{ok}}$
[BLOCK$_{\text{STM}}$]	$\dfrac{E \vdash D_V : \mathsf{ok} \quad E_1 \vdash D_P : \mathsf{ok} \quad E_2 \vdash S : \mathsf{ok}}{E \vdash \texttt{begin } D_V \; D_P \; S \texttt{ end} : \mathsf{ok}}$ where $E_1 = E(D_V, E)$ and $E_2 = E(D_P, E_1)$
[CALL$_{\text{STM}}$]	$\dfrac{E \vdash p : (x : T \to \mathsf{ok}) \quad E \vdash e : T}{E \vdash \texttt{call } p(e) : \mathsf{ok}}$

Table 13.5 *Type rules for* ***Bump*** *statements*

[EMPTY$_{\text{DEC}}$]	$E \vdash \varepsilon : \mathsf{ok}$
[VAR$_{\text{DEC}}$]	$\dfrac{E[x \mapsto T] \vdash D_V : \mathsf{ok} \quad E \vdash a : T}{E \vdash \texttt{var } T\ x := a; D_V : \mathsf{ok}}$
[PROC$_{\text{DEC}}$]	$\dfrac{E[p \mapsto (x : T \to \mathsf{ok})] \vdash D_P : \mathsf{ok}}{E \vdash \texttt{proc } p(T\ x) \texttt{ is } S; D_P : \mathsf{ok}}$

Table 13.4 *Type rules for variable and procedure declarations in* ***Bump***

Typechecking

- Static typechecking
  - All type errors are detected at compile-time
  - Pascal and C are *statically typed*
  - Most modern languages have a large emphasis on static typechecking
- Dynamic typechecking
  - Scripting languages such as JavaScript, PhP, Perl and Python do run-time typechecking
- Mix of Static and Dynamic
  - object-oriented programming requires some runtime typechecking: e.g. Java has a lot of compile-time typechecking but it is still necessary for some potential runtime type errors to be detected by the runtime system

- Static typechecking involves calculating or *inferring* the types of expressions (by using information about the types of their components) and checking that these types are what they should be (e.g. the condition in an *if* statement must have type *Boolean*).

Contextual Analysis Phase

- Purposes:
  - Finish syntax analysis by deriving context-sensitive information
    - Scoping
    - (static) type checking
  - Start to interpret meaning of program based on its syntactic structure
  - Prepare for the final stage of compilation: Code generation

Contextual Analyzer

- Which contextual constraints might the compiler add?
  - Is identifier x declared before it is used?
  - Which declaration of x does an occurrence of x refer to?
  - Is x an Integer, Boolean, array or a function?
  - Is an expression type-consistent?
  - Are any names declared but not used?
  - Has x been initialized before it is being accessed?
  - Is an array reference out of bounds?
  - Does a function `bar` produce a constant value?
  - Where can x be stored? (heap, stack, …)

Why contextual analysis can be hard

- Questions and answers involve non-local information
- Answers mostly depend on values, not syntax
- Answers may involve computations

Solution alternatives:

- Abstract syntax tree
  - specify non-local computations by walking the tree
- Identification tables (sometimes called symbol tables)
  - central store for facts + checking code
- Language design
  - simplify language

To simplify the language design or not?

- Syntax vs. types
  - Bool expressions and Int expressions as syntactic categories
  - One syntactic category of Expressions with types

```
Bexp      :=    true
          |     false
          |     Bexp Bop Bexp

Bop       :=    & | or | …

IntExp    :=    Literal
          |     IntExp Iop IntExp

Iop       :=    + | - | * | / | …
```

vs

```
Exp       :=    Literal
          |     Exp op Exp

Op        :=    & | or | + | - | * | / | …
```

- Psychology of syntax errors vs. type errors
  - Most C programmers accept syntax errors as their fault, but regard typing errors as annoying constraints imposed on them

Language Issues

Example **Pascal:**

Pascal was explicitly designed to be easy to implement with a single pass compiler:

- Every **identifier** must be **declared before** its first **use.**

```
var n:integer;

procedure inc;
begin
   n:=n+1
end
```

?

```
procedure inc;
begin
   n:=n+1
end;

var n:integer;
```

Undeclared Variable!

Language Issues

Example **Pascal:**

- Every **identifier** must be **declared before** it is **used.**
- How to handle mutual recursion then?

```
procedure ping(x:integer)
begin
    ... pong(x-1); ...
end;

procedure pong(x:integer)
begin
    ... ping(x); ...
end;
```

C was designed for a single pass compiler

Mutual recursion problem**:**

- Every **identifier** must be **declared before** it is **used.**
- How to handle mutual recursion then?

```
void ping(int x)
{
   pong(x-1); ...
}
void pong(int x)
{
   ping(x); ...
}
```

```
void pong(int x);

void ping(x:integer)

{

   pong(x-1); ...

}

Void pong(int x)

{

   ping(x); ...

}
```

OK!

Language Issues

Example **Pascal:**

- Every **identifier** must be **declared before** it is **used.**
- How to handle mutual recursion then?

```
forward procedure pong(x:integer)

procedure ping(x:integer)
begin
   ... pong(x-1); ...
end;

procedure pong(x:integer)
begin
   ... ping(x); ...
end;
```

OK!

Language Issues

Example **SML:**

- Every **identifier** must be **declared before** it is **used.**
- How to handle mutual recursion then?

```
fun ping(x:int)=
    ... pong(x-1) ...

and pong(x:int)=
    ... ping(x) ...
;
```

OK!

Language Issues

Example **Java:**

- **identifiers** can be **declared before** they are **used.**
- thus a Java compiler needs at least two passes

```
Class Example {
      void inc() { n = n + 1; }
      int n;
      void use() { n = 0 ; inc(); }
}
```

Scope of Variable

- Range of program that can reference that variable (ie access the corresponding data object by the variable's name)
- Variable is *local* to program or block if it is declared there
- Variable is *non-local* to program unit if it is visible there but not declared there
- Static vs. Dynamic scope

Static Scoping

- Scope computed at compile time, based on program text
- To determine the name of a used variable we must find statement declaring variable
- Subprograms and blocks generate hierarchy of scopes
  – Subprogram or block that declares current subprogram or contains current block is its *static parent*
- General procedure to find declaration:
  – First see if variable is local; if yes, done
  – If non-local to current subprogram or block recursively search static parent until declaration is found
  – If no declaration is found this way, undeclared variable error detected

Example

```
program main;
   var x : integer;
   procedure sub1;
     var x : integer;
     begin { sub1 }
       … x …
     end; { sub1 }
begin { main }
  … x …
  end; { main }
```

Example (from p. 88 in Transitions and Trees)

```
begin
        var x:= 0;
        var y:= 42

        proc p is x:= x+3;
        proc q is call p;

        begin
                var x:=9;
                proc p is x := x+1;
                call q;
                y := x
        end
end
```

Assuming static scope for procedures and variables,
What is the value assigned to y ?

Value of y is 9, assuming static scope for procedures and variables

Dynamic Scope

- Now generally thought to have been a mistake
- Main example of use: original versions of LISP
  - APL, PostScript
  - (Note: Scheme uses static scope)
  - Perl allows variables to be declared to have dynamic scope
- Determined by the calling sequence of program units, not static layout
- Name bound to corresponding variable most recently declared among still active subprograms and blocks

Example

```
program main;
   var x : integer;
   procedure sub1;
     begin { sub1 }
       … x …
     end; { sub1 }
procedure sub2;
   var x : integer;
     begin { sub2 }
     … call sub1 …
     end; { sub2 }
 … call  sub2…
 end; { main }
```

Example (from p. 88 in Transitions and Trees)

```
begin
        var x:= 0;
        var y:= 42

        proc p is x:= x+3;
        proc q is call p;

        begin
                var x:=9;
                proc p is x := x+1;
                call q;
                y := x
        end
end
```

Assuming dynamic scope for procedures and variables,
What is the value assigned to y ?

Value of y is 10, assuming dynamic scope for procedures and variables

Value of y is 12, assuming static scope for procedures and dynamic of variables

Formal rules

(from p. 89-93 in Transitions and Trees)

[PROC-BIP$_{BSS}$]
$$\frac{env_V \vdash \langle D_P, env_P[p \mapsto (S, env_V, env_P)]\rangle \to_{DP} env'_P}{env_V \vdash \langle \texttt{proc}\ p\ \texttt{is}\ S\ ; D_P, env_P\rangle \to_{DP} env'_P}$$

[PROC-EMPTY-BIP$_{BSS}$] $\quad env_V \vdash \langle \varepsilon, env_P\rangle \to_{DP} env_P$

Table 6.8 *Transition rules for procedure declarations assuming fully static scope rules*

[PROC-BIP$_{BSS}$]
$$\frac{env_V \vdash \langle D_P, env_P[p \mapsto S]\rangle \to_{DP} env'_P}{env_V \vdash \langle \texttt{proc}\ p\ \texttt{is}\ S\ ; D_P, env_P\rangle \to_{DP} env'_P}$$

[PROC-EMPTY-BIP$_{BSS}$] $\quad env_V \vdash \langle \varepsilon, env_P\rangle \to_{DP} env_P$

Table 6.4 *Transition rules for procedure declarations (assuming fully dynamic scope rules)*

[CALL-STAT-STAT$_{BSS}$]
$$\frac{env'_V[\text{next} \mapsto l], env'_P \vdash \langle S, sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{call}\ p, sto\rangle \to sto'}$$
where $env_P\, p = (S, env'_V, env'_P)$
and $l = env_V\ \text{next}$

Table 6.9 *Transition rules for procedure calls assuming fully static scope rules*

[CALL-DYN-DYN$_{BSS}$]
$$\frac{env_V, env_P \vdash \langle S, sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{call}\ p, sto\rangle \to sto'}$$
where $env_P\, p = S$

Table 6.5 *Transition rule for procedure calls (assuming fully dynamic scope rules)*

[EMPTY$_{DEC}$] $\quad E \vdash \varepsilon : \mathsf{ok}$

[VAR$_{DEC}$]
$$\frac{E[x \mapsto T] \vdash D_V : \mathsf{ok} \quad E \vdash a : T}{E \vdash \texttt{var}\ T\ x := a; D_V : \mathsf{ok}}$$

[PROC$_{DEC}$]
$$\frac{E[p \mapsto (x : T \to \mathsf{ok})] \vdash D_P : \mathsf{ok}}{E \vdash \texttt{proc}\ p(T\ x)\ \texttt{is}\ S; D_P : \mathsf{ok}}$$

Table 13.4 *Type rules for variable and procedure declarations in **Bump***

Pause

Organization of a Compiler

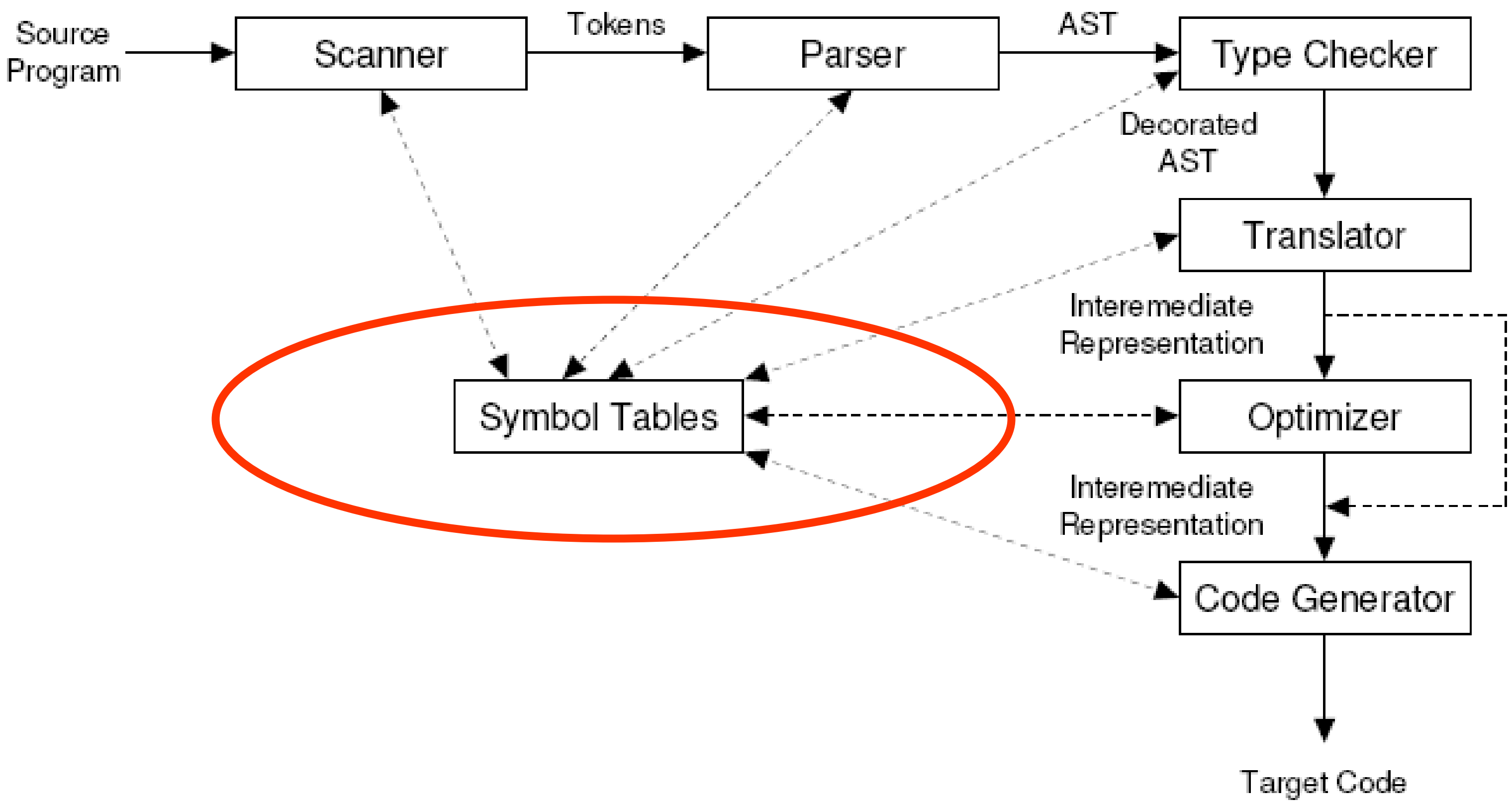


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

Identification Table

- The identification table (also often called symbol table) is a dictionary-style data structure in which we somehow store identifier names and relate each identifier to its corresponding **attributes**.
- Typical operations:
  - Empty the table
  - Add an entry (Identifier -> Attribute)
  - Find an entry for an identifier
  - (open and close scope)

Identification Table

- The organization of the identification table depends on the programming language.
- Different kinds of "block structure" in languages:
  - Monolithic block structure: e.g. ac, BASIC, COBOL
  - Flat block structure: e.g. Fortran (and functions in C)
  - Nested block structure => Modern "block-structured" PLs (e.g. Algol, Pascal, C, C++, Scheme, Java,…)

a **block =** an area of text in the program that corresponds to some kind of boundary for the visibility of identifiers.

block structure = the textual relationship between blocks in a program.

C# scope definition

10.7 Scopes

The *scope* of a name is the region of program text within which it is possible to refer to the entity declared by the name without qualification of the name. Scopes can be *nested*, and an inner scope can redeclare the meaning of a name from an outer scope. [*Note*: This does not, however, remove the restriction imposed by §10.3 that within a nested block it is not possible to declare a local variable or local constant with the same name as a local variable or local constant in an enclosing block. *end note*] The name from the outer scope is then said to be *hidden* in the region of program text covered by the inner scope, and access to the outer name is only possible by qualifying the name.

- The scope of a namespace member declared by a *namespace-member-declaration* (§16.5) with no enclosing *namespace-declaration* is the entire program text.
- The scope of a namespace member declared by a *namespace-member-declaration* within a *namespace-declaration* whose fully qualified name is N, is the *namespace-body* of every *namespace-declaration* whose fully qualified name is N or starts with N, followed by a period.
- The scope of a name defined by an *extern-alias-directive* (§16.3) extends over the *using-directives*, *global-attributes* and *namespace-member-declarations* of the *compilation-unit* or *namespace-body* in which the *extern-alias-directive* occurs. An *extern-alias-directive* does not contribute any new members to the underlying declaration space. In other words, an *extern-alias-directive* is not transitive, but, rather, affects only the *compilation-unit* or *namespace-body* in which it occurs.
- The scope of a name defined or imported by a *using-directive* (§16.4) extends over the *global-attributes* and *namespace-member-declarations* of the *compilation-unit* or *namespace-body* in which the *using-directive* occurs. A *using-directive* can make zero or more namespace or type names available within a particular *compilation-unit* or *namespace-body*, but does not contribute any new members to the underlying declaration space. In other words, a *using-directive* is not transitive, but, rather, affects only the *compilation-unit* or *namespace-body* in which it occurs.
- The scope of a member declared by a *class-member-declaration* (§17.2) is the *class-body* in which the declaration occurs. In addition, the scope of a class member extends to the *class-body* of those derived classes that are included in the accessibility domain (§10.5.2) of the member.
- The scope of a member declared by a *struct-member-declaration* (§18.2) is the *struct-body* in which the declaration occurs.
- The scope of a member declared by an *enum-member-declaration* (§21.3) is the *enum-body* in which the declaration occurs.
- The scope of a parameter declared in a *method-declaration* (§17.5) is the *method-body* of that *method-declaration*.
- The scope of a parameter declared in an *indexer-declaration* (§17.8) is the *accessor-declarations* of that *indexer-declaration*.
- The scope of a parameter declared in an *operator-declaration* (§17.9) is the *block* of that *operator-declaration*.
- The scope of a parameter declared in a *constructor-declaration* (§17.10) is the *constructor-initializer* and *block* of that *constructor-declaration*.
- The scope of a label declared in a *labeled-statement* (§15.4) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *local-variable-declaration* (§15.5.1) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *switch-block* of a `switch` statement (§15.7.2) is the *switch-block*.
- The scope of a local variable declared in a *for-initializer* of a `for` statement (§15.8.3) is the *for-initializer*, the *for-condition*, the *for-iterator*, and the contained *statement* of the `for` statement.
- The scope of a local constant declared in a *local-constant-declaration* (§15.5.2) is the *block* in which the declaration occurs. It is a compile-time error to refer to a local constant in a textual position that precedes its *constant-declarator*.

Within the scope of a namespace, class, struct, or enumeration member it is possible to refer to the member in a textual position that precedes the declaration of the member. [*Example*:

```
class A
{
   void F() {
      i = 1;
   }
   int i = 0;
}
```

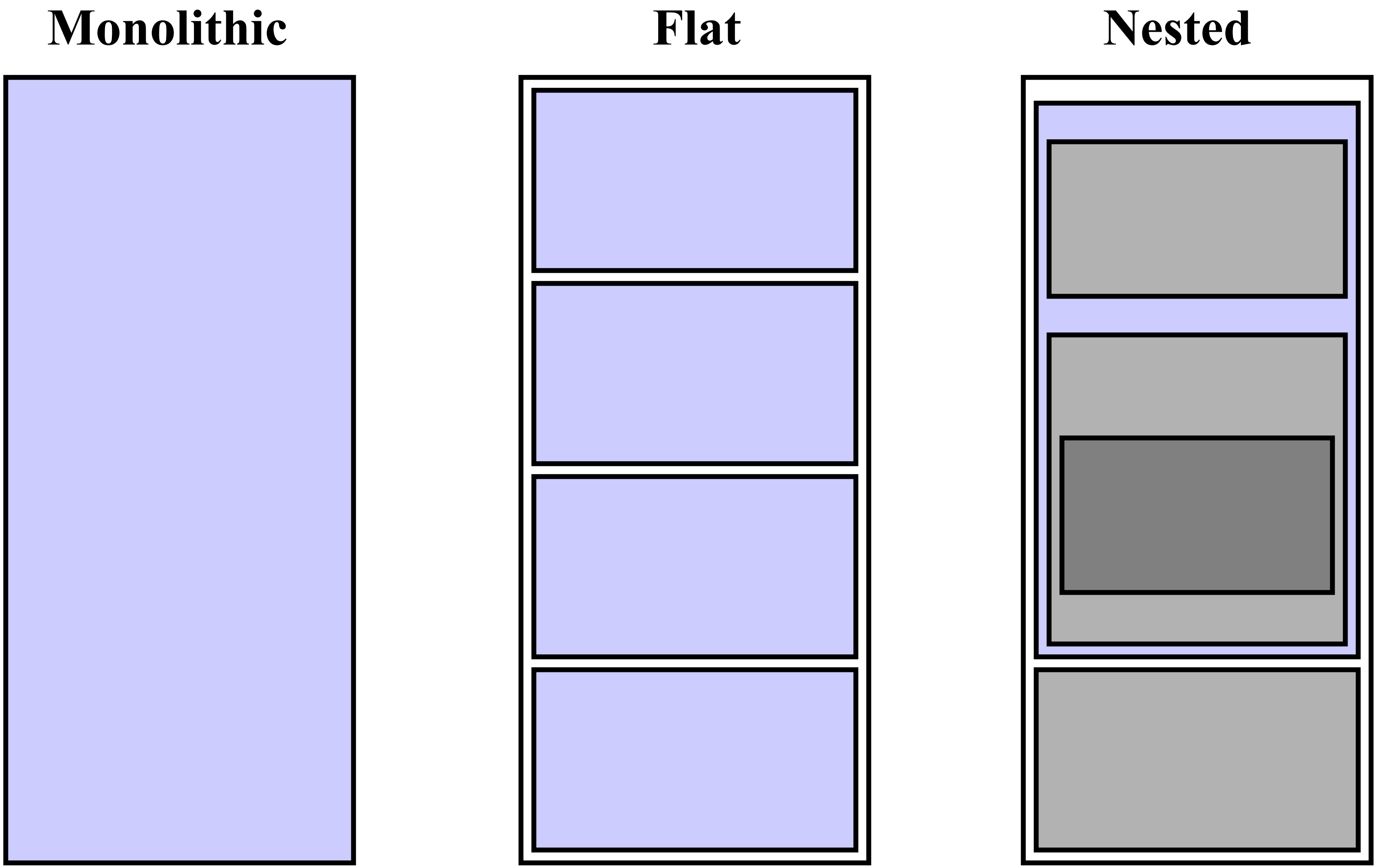
Here, it is valid for `F` to refer to `i` before it is declared. *end example*]

Within the scope of a local variable, it is a compile-time error to refer to the local variable in a textual position that precedes the *local-variable-declarator* of the local variable. [*Example*:

```
class A
{
   int i = 0;
   void F() {
      i = 1;              // Error, use precedes declaration
      int i;
      i = 2;
   }
   void G() {
      int j = (j = 1);    // Valid
   }
   void H() {
      int a = 1, b = ++a; // Valid
   }
}
```

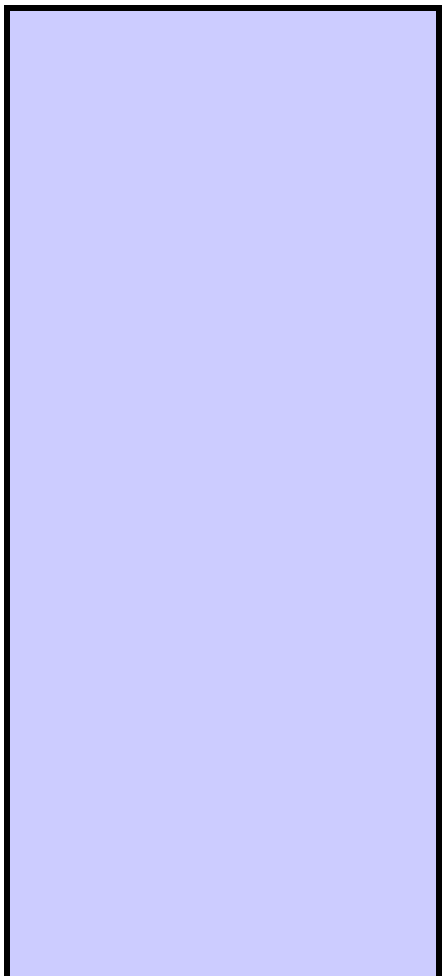
In the `F` method above, the first assignment to `i` specifically does not refer to the field declared in the outer scope. Rather, it refers to the local variable and it results in a compile-time error because it textually precedes the declaration of the variable. In the `G` method, the use of `j` in the initializer for the declaration of `j` is valid because the use does not precede the *local-variable-declarator*. In the `H` method, a subsequent *local-variable-declarator* correctly refers to a local variable declared in an earlier *local-variable-declarator* within the same *local-variable-declaration*. *end example*]

Different kinds of Block Structure... a picture



Monolithic Block Structure

Monolithic



A language exhibits **monolithic block structure** if the only block is the entire program.

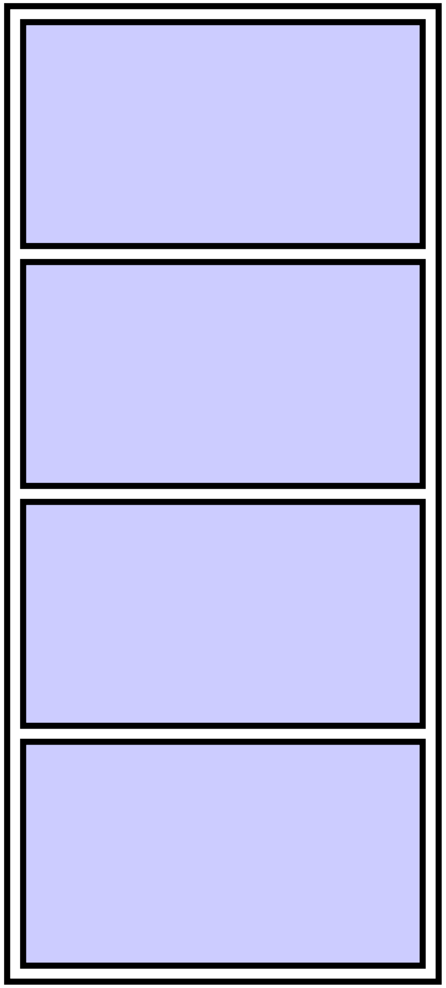
=> Every identifier is visible throughout the entire program

Very simple scope rules:

- No identifier may be declared more than once
- For every applied occurrence of an identifier *I* there must be a corresponding declaration.

Flat Block Structure

Flat



A language exhibits **flat block structure** if the program can be subdivided into several disjoint blocks

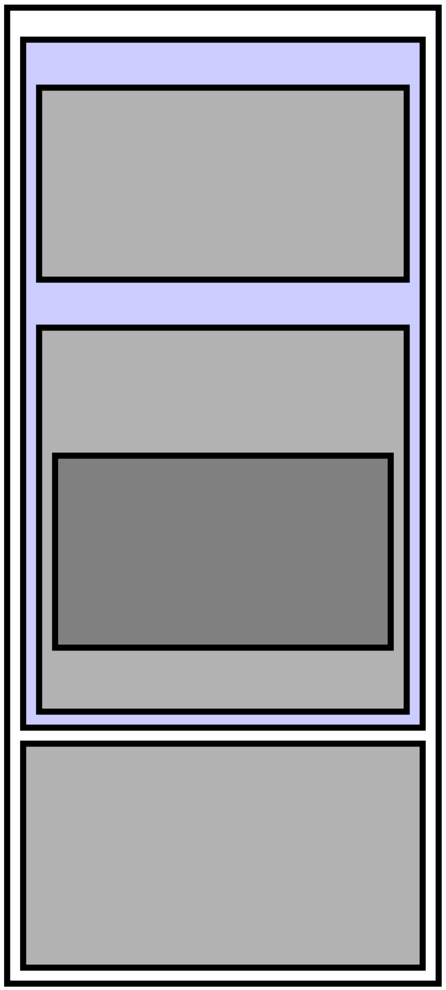
There are two scope levels: global or local.

Typical scope rules:

- a globally defined identifier may be redefined locally
- several local definitions of a single identifier may occur in different blocks (but not in the same block)
- For every applied occurrence of an identifier there must be either a local declaration within the same block or a global declaration.

Nested Block Structure

Nested



A language exhibits **nested block structure** if blocks may be nested one within another (typically with no upper bound on the level of nesting that is allowed).

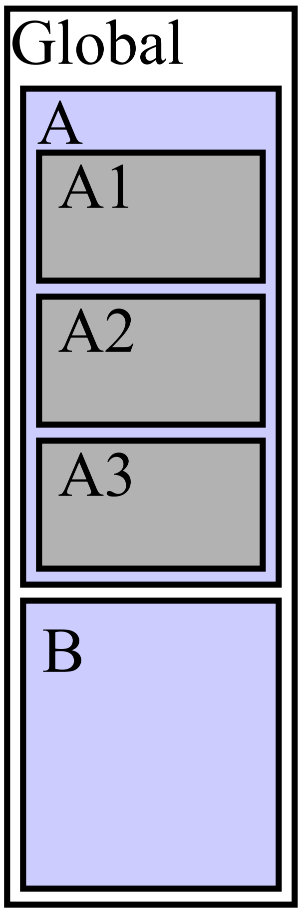
There can be any number of scope levels (depending on the level of nesting of blocks):

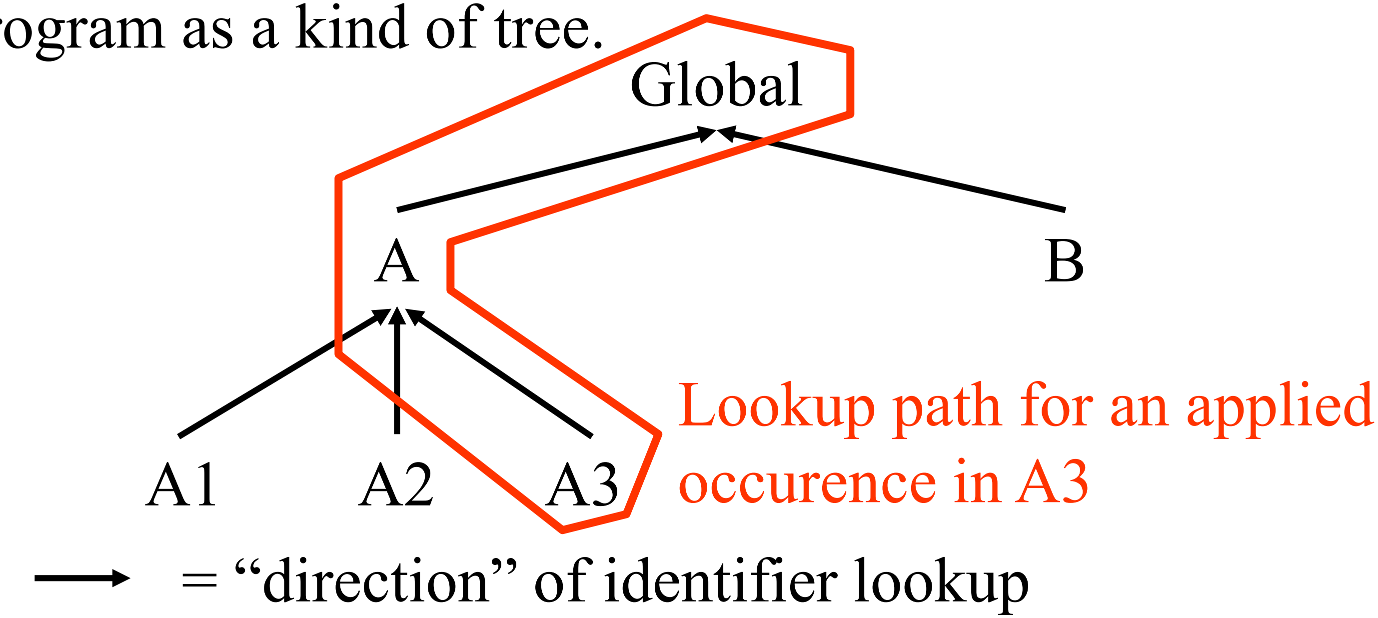
Typical scope rules:

- no identifier may be declared more than once within the same block (at the same level).
- for any applied occurrence there must be a corresponding declaration, either within the same block or in a block in which it is nested.

Identification Table

For a typical programming language, i.e. statically scoped language and with nested block structure we can visualize the structure of all scopes within a program as a kind of tree.





At any one time (in analyzing the program) only a single path on the tree is accessible.
=> We don't necessarily need to keep the whole "scope" tree in memory all the time.

A Symbol Table Interface

- Methods
  - OpenScope()
  - CloseScope()
  - EnterSymbol(name, type)
  - RetreiveSymbol(name)
  - DeclaredLocally(name)
- Ex.
  - (Fig. 8.2) Code to build the symbol table for the AST in Fig. 8.1

```
procedure buildSymbolTable( )
    call processNode( ASTroot )
end

procedure processNode( node )
    switch (kind( node ))
        case Block
            call symtab.openScope( )                                    (1)
        case Dcl
            call symtab.enterSymbol( node.name, node.type )
        case Ref
            sym ← symtab.retrieveSymbol( node.name )
            if sym = null
            then  call error( "Undeclared symbol : ", sym )
    foreach c ∈ node.getChildren( ) do  call processNode( c )
    if kind( node ) = Block
    then
        call symtab.closeScope( )                                       (2)
end
```

Figure 8.2: Building the symbol table

Ac SymbolTableFilling

```
@Override
void visit(Prog n) {
    // TODO Auto-generated method stub
    for(AST ast : n.prog){
        ast.accept(this);
    };

}

@Override
void visit(SymDeclaring n) {
    // TODO Auto-generated method stub

}

@Override
void visit(FloatDcl n) {
    // TODO Auto-generated method stub
    if (AST.SymbolTable.get(n.id) == null) AST.SymbolTable.put(n.id,AST.FLTTYPE);
    else error("variable " + n.id + " is already declared");

}

@Override
void visit(IntDcl n) {
    // TODO Auto-generated method stub
    if (AST.SymbolTable.get(n.id) == null) AST.SymbolTable.put(n.id,AST.INTTYPE);
    else error("variable " + n.id + " is already declared");
```

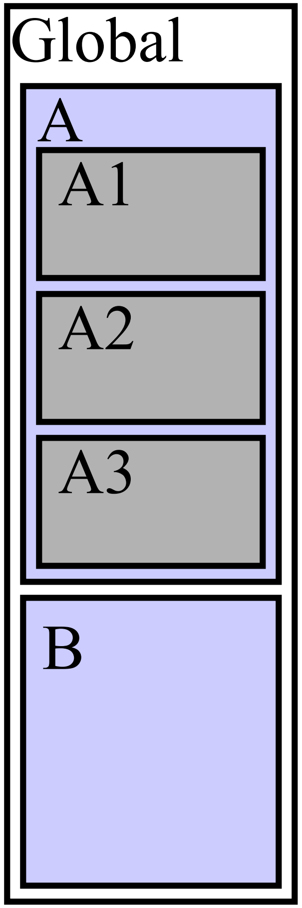
One Symbol Table or Many?

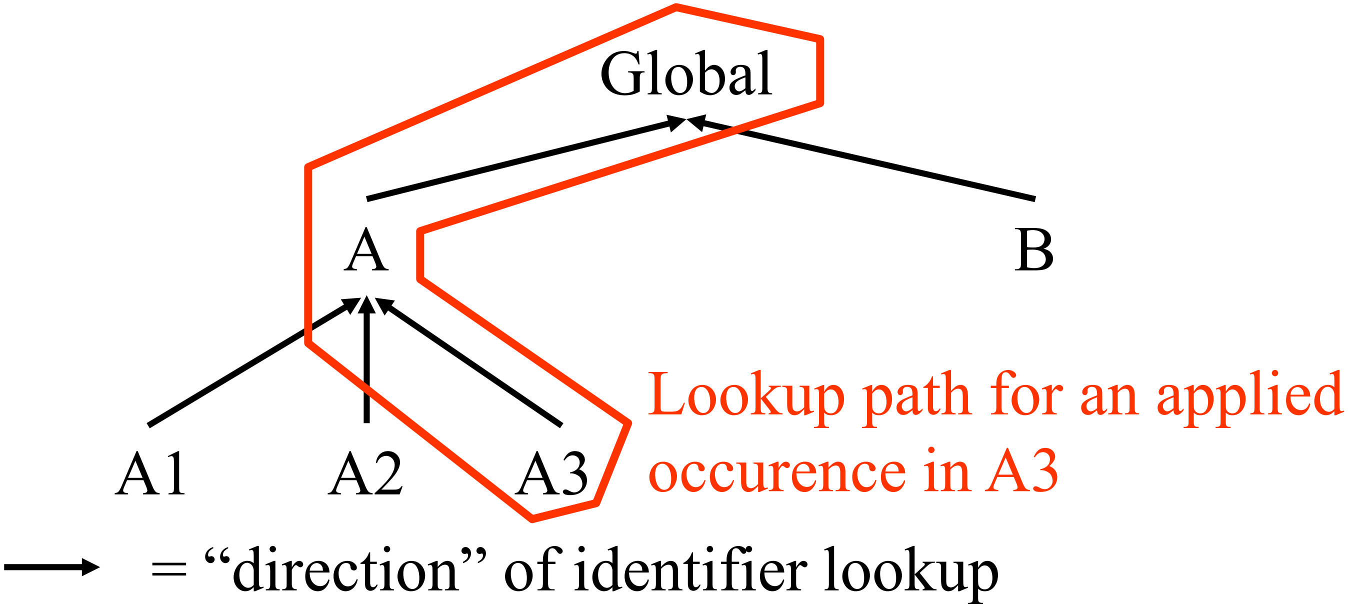
- Two common approaches to implementing block-structured symbol tables
  - A symbol table associated with each scope
  - Or a single, global table

An Individual Table for Each Scope

- Because name scope are opened and closed in a last-in first-out (LIFO) manner, a stack is an appropriate data structure for a search
  - The innermost scope appears at the top of stack
  - OpenScope(): pushes a new symbol table
  - CloseScope(): pop
- Disadvantage
  - Need to search a name in a number of symbol tables
  - Cost depending on the number of nonlocal references and the depth of nesting

Individual Table for each scope





At any one time (in analyzing the program) only a single path on the tree is accessible.
=> We can keep a stack of identification tables, one for each "active" scope.

One Symbol Table

- All names in the same table
  - Uniquely identified by the scope name or depth
- RetrieveSymbol() need not chain through scope tables to locate a name

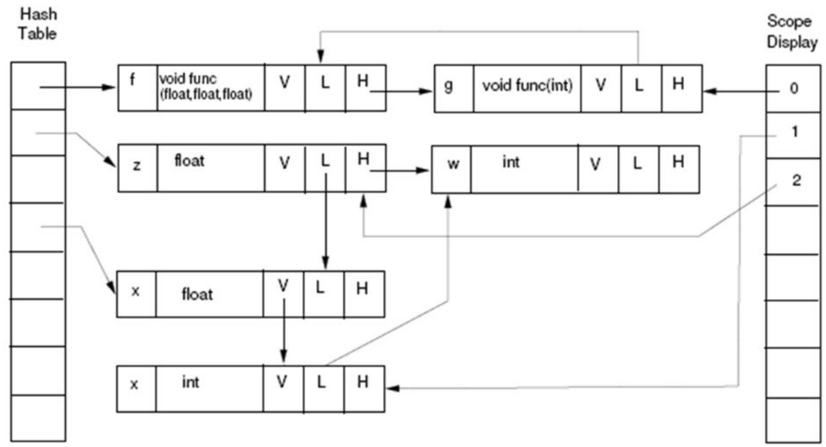


Figure 8.8: Detailed layout of the symbol table for Figure 8.1. The V, L, and H fields abbreviate the Var, Level, and Hash fields, respectively

Entering and Finding Names

- Examine the time needed to insert symbols, retrieve symbols, and maintain scopes
  - In particular, we pay attention to the cost of retrieving symbols
  - Names can be declared no more than once in each scope, but typically referenced multiple times

- Various approaches
  - Unordered list
    - Insertion: fast, Retrieval: linear scan, Impractically slow
  - Ordered list
    - Fast retrieval , but expensive insertion
  - Binary search trees
    - Insert, search: O(log n),
  - Balanced trees
    - Insert, search: O(log n) – avoids worst case for binary trees
  - Hash tables
    - Insert, search: O(1), given sufficiently large table, a good hash function and appropriate collision-handling techniques

Advanced Features

- Extensions of the simple symbol table framework to accommodate advanced features of modern programming languages
  - Name augmentation (overloading)
  - Name hiding and promotion
  - Modification of search rules

Implicit Declarations

- In some languages, the appearance of a name in a certain context serves to declare the name as well
  - E.g.: labels in C
  - In Fortran: inferred from the identifier's first letter
  - In Ada: an index is implicitly declared to be of the same type as the range specifier
  - A new scope is opened for the loop so that the loop index cannot clash with an existing variable
    - E.g. for (int i=1; i<10; i++) { … }
  - Variables in dynamic languages like Python

Symbol Table Summary

- The symbol table organization in this chapter efficiently represents scope-declared symbols in a block-structured language
- Most languages include rules for symbol promotion to a global scope
- Issues such as inheritance, overloading, and aggregate data types must be considered
  - Records, objects and classes

Declaration Processing Fundamentals

- Attributes in the symbol table
  - Internal representations of declarations
  - Identifiers are used in many different ways in a modern programming language
    - Variables, constants, types, procedures, classes, and fields
    - Every identifier will not have the same set of attributes
  - We need a data structure to store the variety of information
    - Using a struct that contains a tag, and a union for each possible value of the tag
    - Using object-based approach, Attributes and appropriate subclasses

Type Descriptor Structures

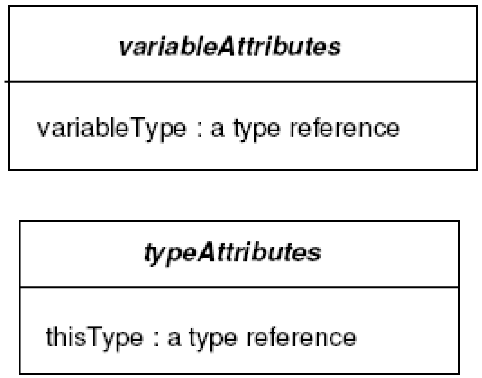


Figure 8.9: Attribute Descriptor Structures

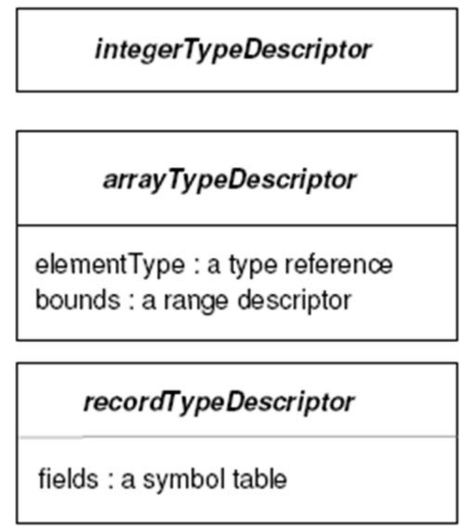
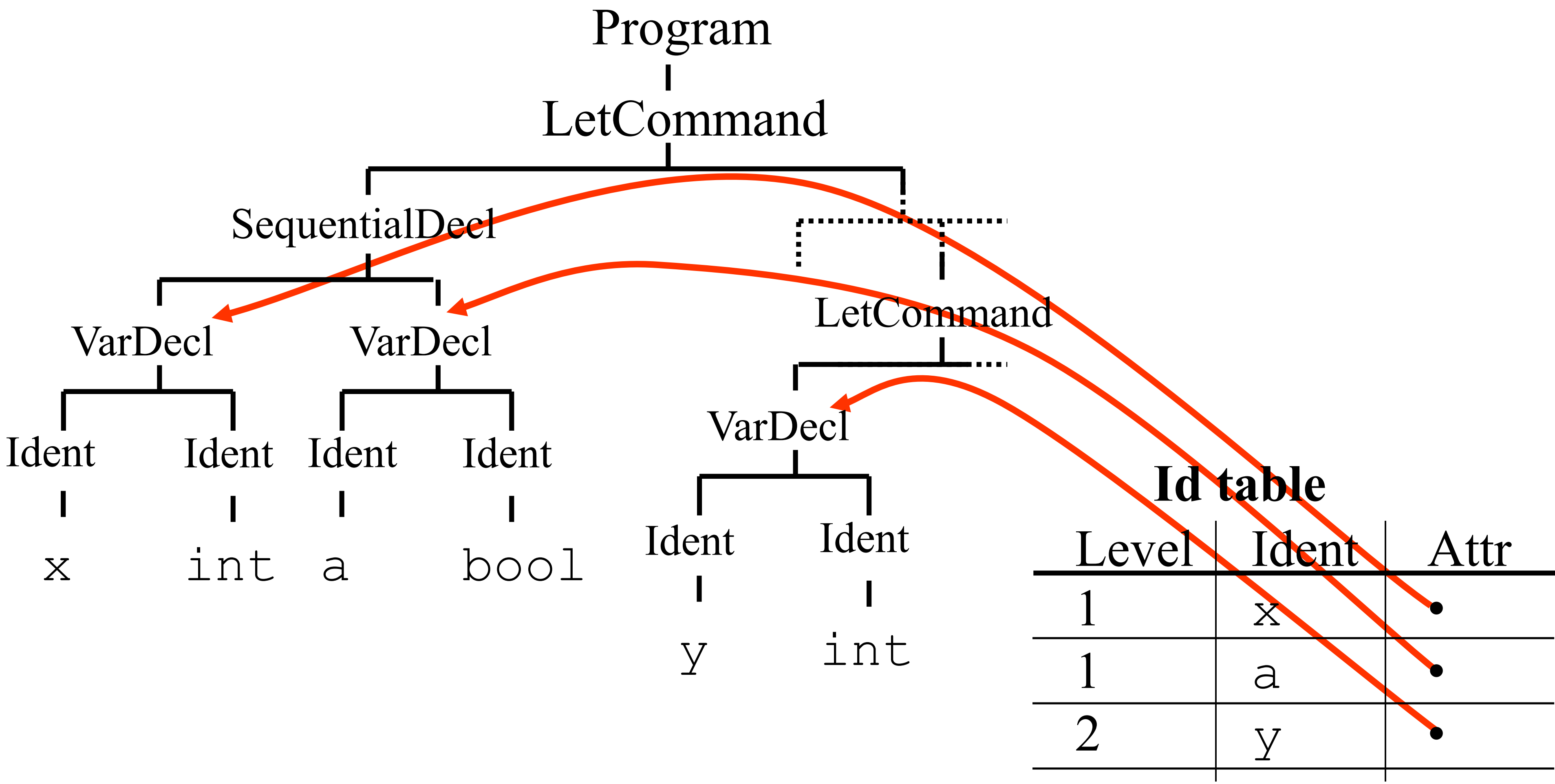


Figure 8.10: Type Descriptor Structures

Attributes as pointers to Declaration AST's



The Standard Environment

- Most programming languages have a set of predefined functions, operators etc.
- We call this the **standard environment**

At the start of identification the ID table is not empty but... needs to be initialized with entries representing the standard environment.

Scope for Standard Environment

Should the scope level for the standard environment be the same as the globals (level 1) or outside the globals (level 0)?

- C: level 1
- Mini Triangle: level 0

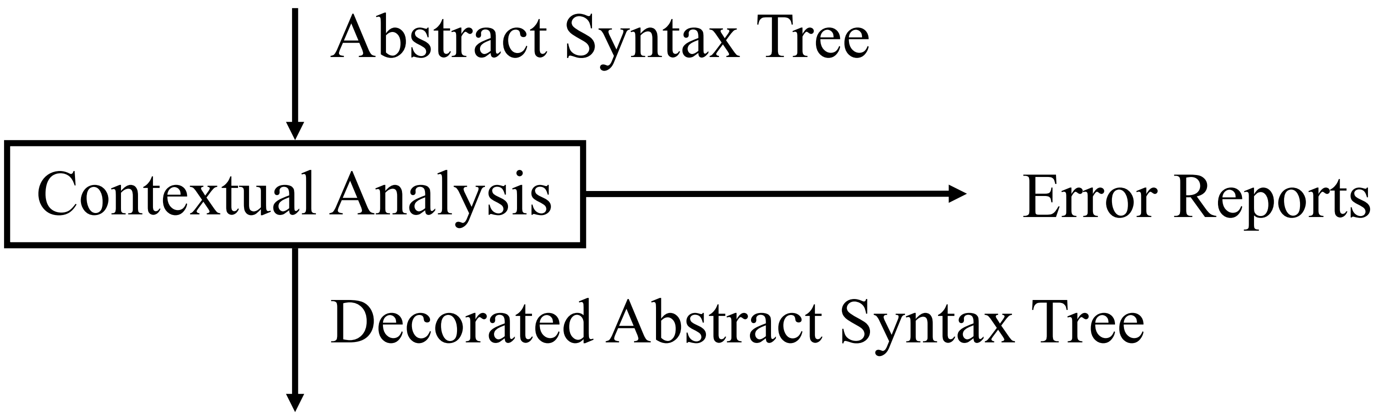
- Consequence:

```
let
  var false : Integer
in
  begin
    false := 3;
    putint ( false )
  end
```

is a perfectly correct Mini Triangle program

- Similar with Integer or putint. . .

Contextual Analysis -> Decorated AST



Contextual analysis:

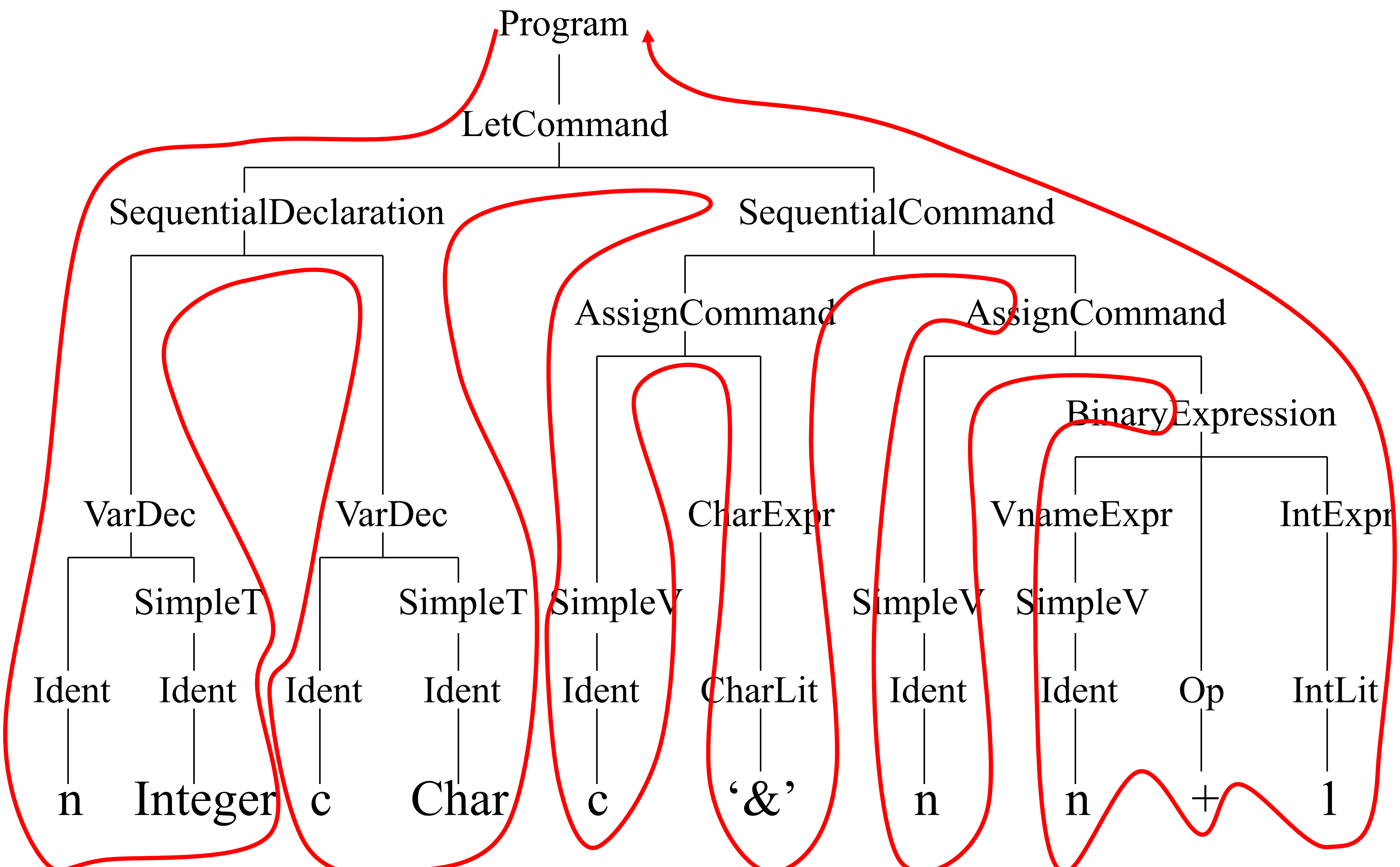
- Scope checking: verify that all applied occurrences of identifiers are declared
- Type checking: verify that all operations in the program are used according to their type rules.

Annotate AST:

- Applied identifier occurrences => declaration
- Expressions => Type

Contextual Analysis

Identification and type checking are combined into a depth-first traversal of the AST.



Implementing Tree Traversal

- “Traditional” OO approach
- Visitor approach
  - GOF
  - Using static overloading
  - Reflective
  - (dynamic)
  - (SableCC style)
- “Functional” approach
- Active patterns in Scala (or F#)
- (Aspect oriented approach)

What can you do in your project now?

- Start designing and defining:
  - Scope rules for your language
    - Informal (in structured English)
    - Formally (when you have read chapter 6 in Trans. & Trees)
- Start thinking about designing and defining
  - the type system for your language
    - Informal (in structured English)
    - Formally (when you have read chapter 13 in Trans. & Trees)
- Start thinking about implementing
  - Symbol table(s)
  - Scope cheking
  - (simple) type cheking

Languages and Compilers (SProg og Oversættere)

Lecture 11
Type Checking

Bent Thomsen
Department of Computer Science
Aalborg University

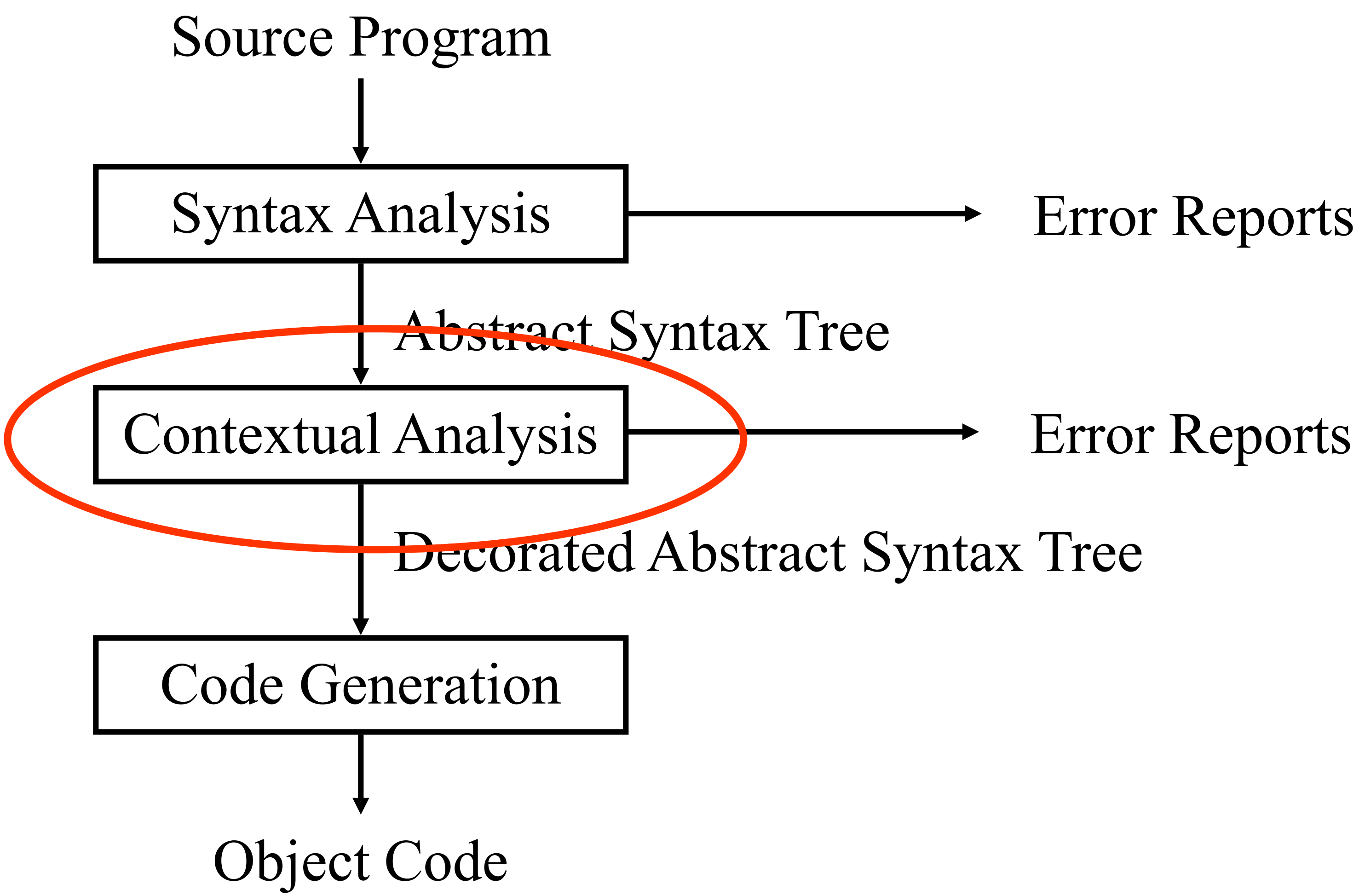
Learning Goals

- Understand how (simpel) type checking is implemented
- Understand that type checking is language dependent and thus different from language to language
- Understand that similar principles apply to many different languages

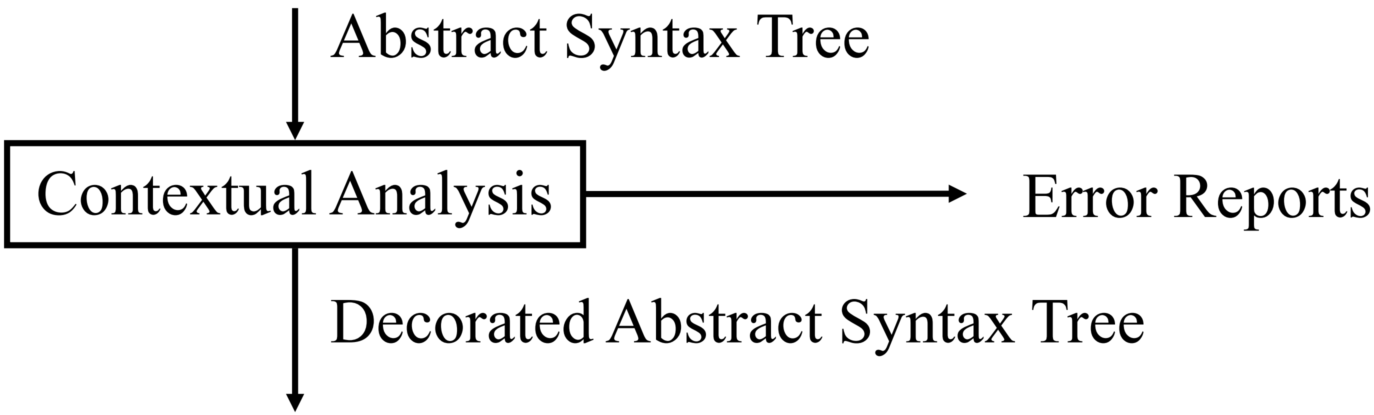
Programming Language Specification

- A language specification has (at least) three parts:
  - Syntax of the language: usually formal: EBNF
  - **Contextual constraints:**
    - **scope rules**
      - » **often written in English, but can be formal**
      - » **(see p. 86-93 in Transitions and Trees)**
    - **type rules**
      - » **formal or informal**
      - » **See p.185-210 in Transitions and Trees)**
  - Semantics:
    - defined by the implementation
    - informal descriptions in English
    - formal using operational or denotational semantics
      - » See Transitions and Trees

The "Phases" of a Compiler



Contextual Analysis -> Decorated AST



Contextual analysis:

- Scope checking: verify that all applied occurrences of identifiers are declared
- Type checking: verify that all operations in the program are used according to their type rules.

Annotate AST:

- Applied identifier occurrences => Type or ref to declaration
- Expressions => Type

Type Checking

- In a statically typed language every expression *E* either:
  - Is ill-typed
  - Or has a static type that can be computed without actually evaluating *E*
- When an expression *E* has static type *T* this means that when *E* is evaluated then the returned value will **always** have type *T*
- => This makes static type checking possible!

- Note in languages with subtyping the value returned by E with static type T maybe of type T′ where T' is a subtype of T, written T' < T

Type Checking: How Does It Work

For most statically typed programming languages, type checking is a bottom up algorithm over the AST:

- Types of expression AST leaves are known immediately:
  - literals => obvious
  - variables => from the ID table
  - named constants => from the ID table
- Types of internal nodes are inferred from the type of the children and the type rule for that kind of expression

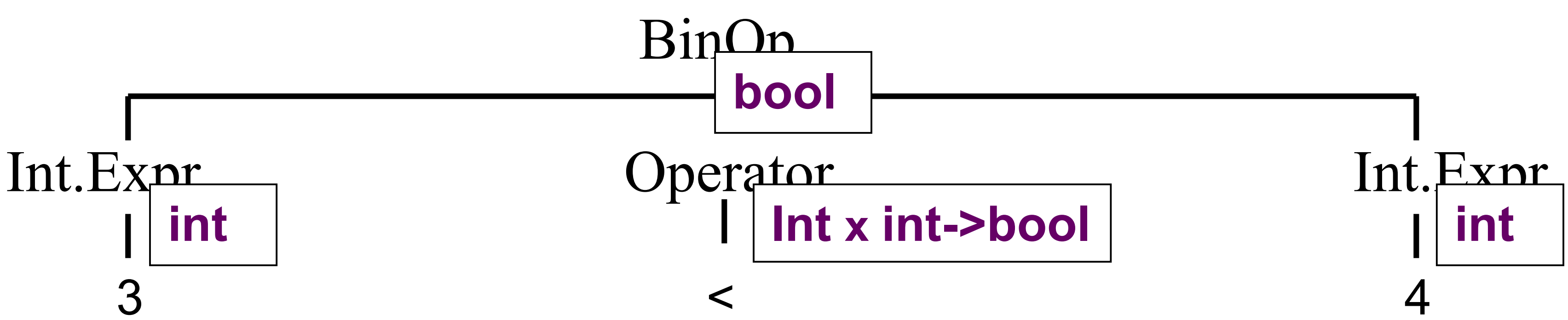
Type Checking: How Does It Work

Example: the type of **<** operation

Type rule of **<** :

E1 **<** *E2* is type correct and of type **`Boolean`**

if *E1* and *E2* are type correct and of type **`Integer`**

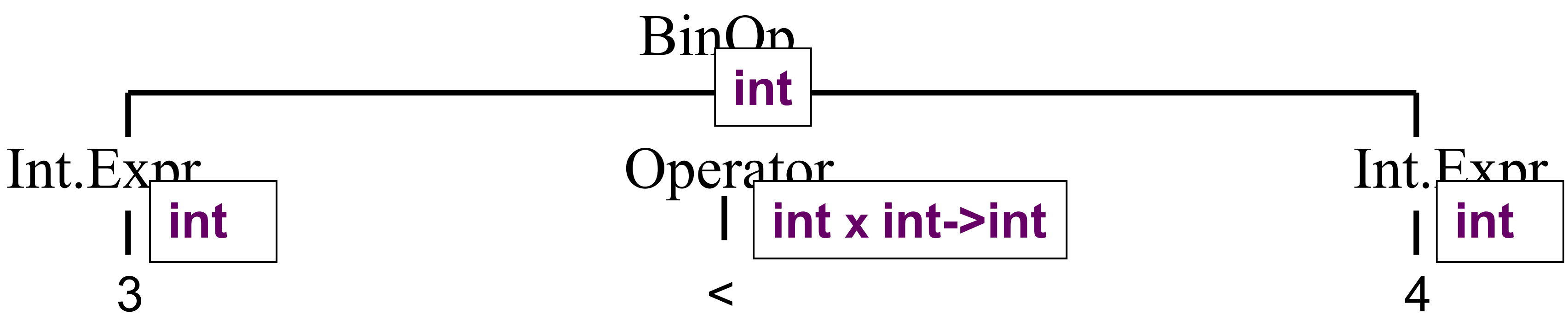


Type Checking: How Does It Work

Example: the type of **+** operation

Type rule of **+** :

E1 **+** *E2* is type correct and of type **`Integer`**
if *E1* and *E2* are type correct and of type **`Integer`**



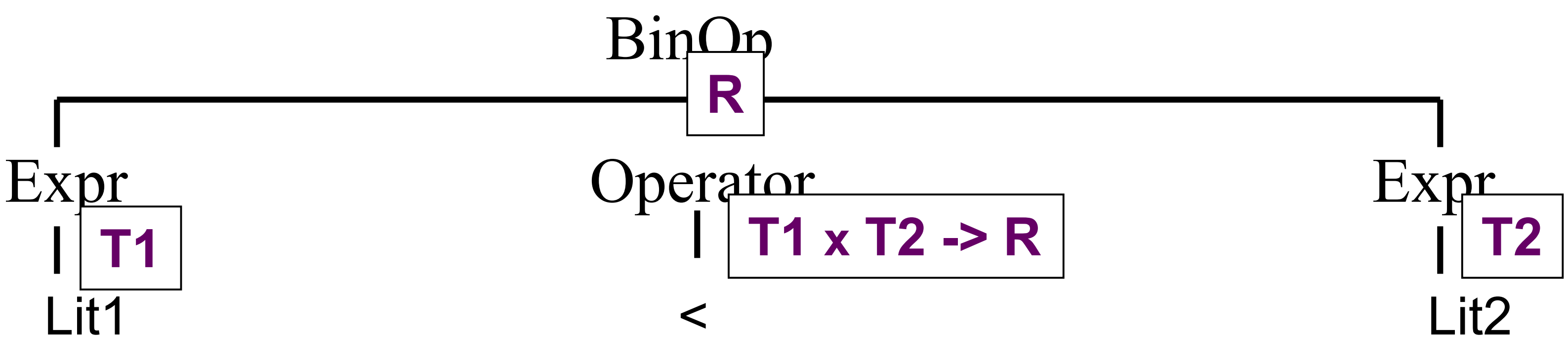
$$[\text{ADDEXP}] \qquad \frac{E \vdash e_1 : \mathsf{Int} \quad E \vdash e_2 : \mathsf{Int}}{E \vdash e_1 + e_2 : \mathsf{Int}}$$

Type Checking: How Does It Work

General: the type of a binary operation expression

Type rule:

If ***op*** is an operation of type ***T1xT2->R*** then ***E1 op E2*** is type correct and of type ***R*** if ***E1*** and ***E2*** are type correct and have types compatible with ***T1*** and ***T2*** respectively

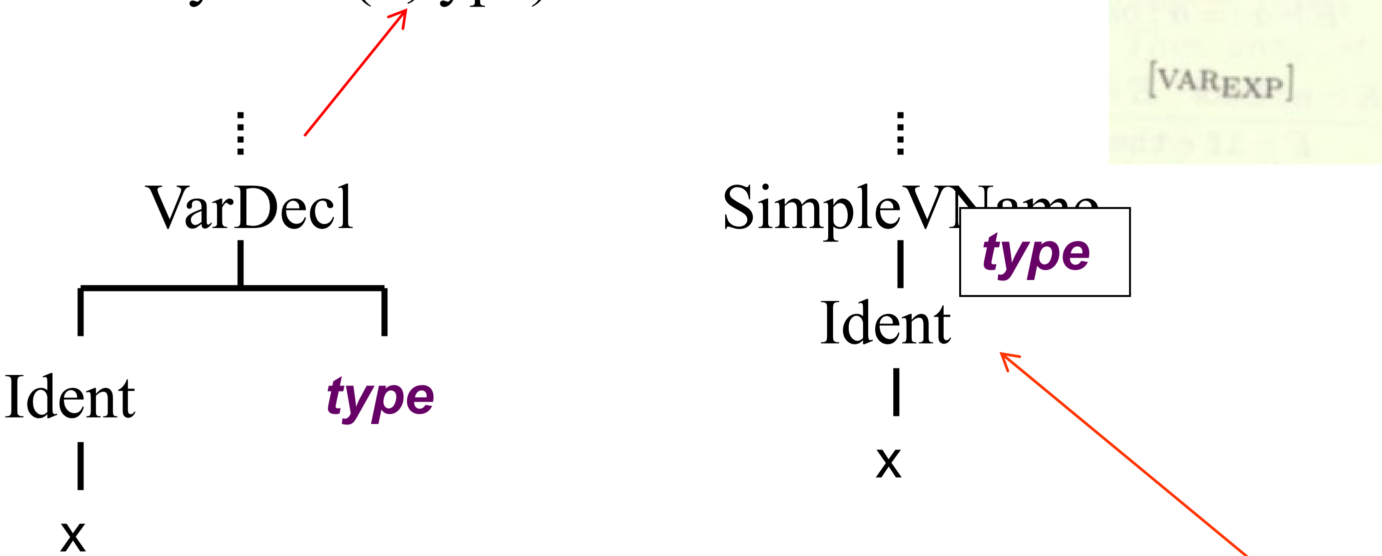


Type Checking: How Does It Work

Example: Type of a variable (applied occurrence)

During Identification/SymbolTableFilling:
EnterSymbol(x,type)

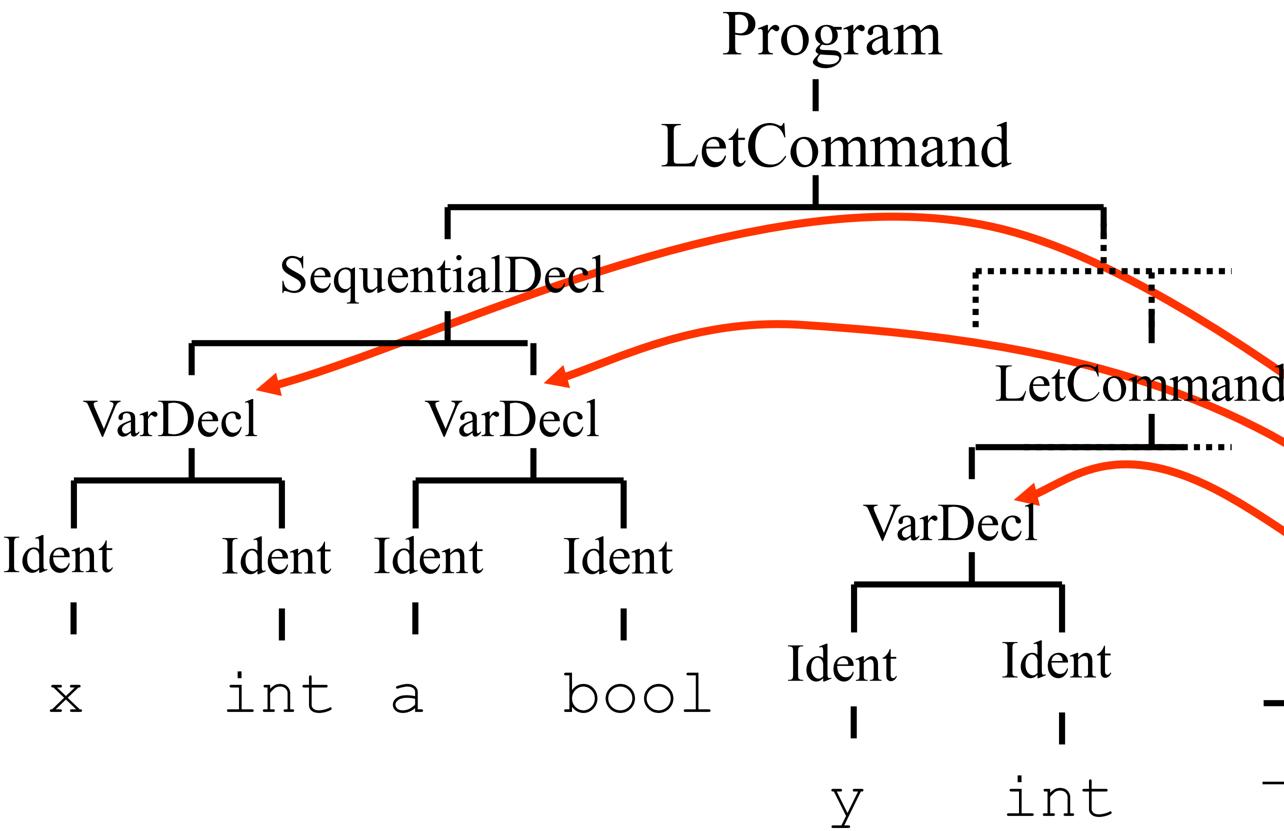
$$[\text{VAR}_{\text{EXP}}] \qquad \frac{E(x) = T}{E \vdash x : T}$$



$$[\text{VAR}_{\text{DEC}}] \qquad \frac{E[x \mapsto T] \vdash D_V : \mathsf{ok} \quad E \vdash a : T}{E \vdash \mathsf{var}\ T\ x := a; D_V : \mathsf{ok}}$$

During typeChecking:
RetreiveSymbol(x) -> type

Attributes as pointers to Declaration AST's

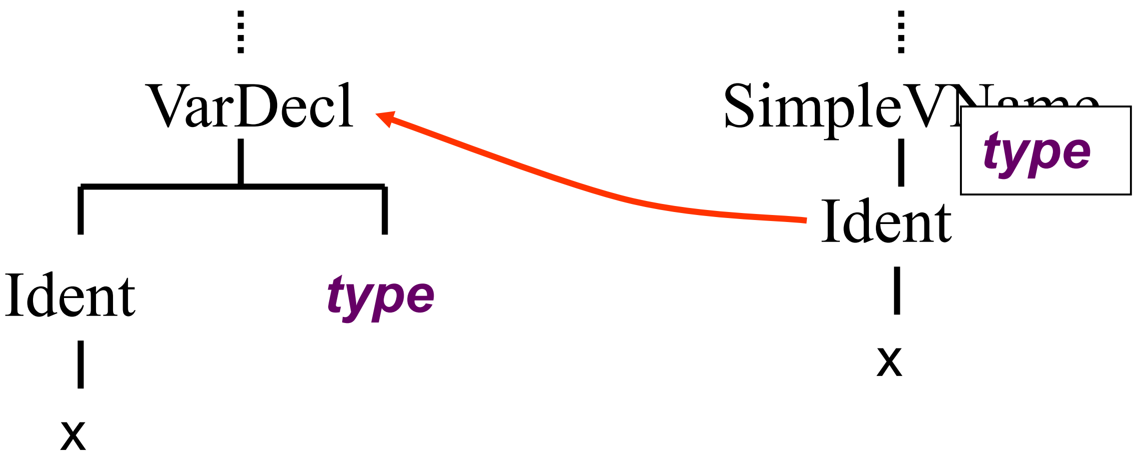


Id table

Level	Ident	Attr
1	x	•
1	a	•
2	y	•

Type Checking: How Does It Work

Example: Type of a variable (applied occurrence)



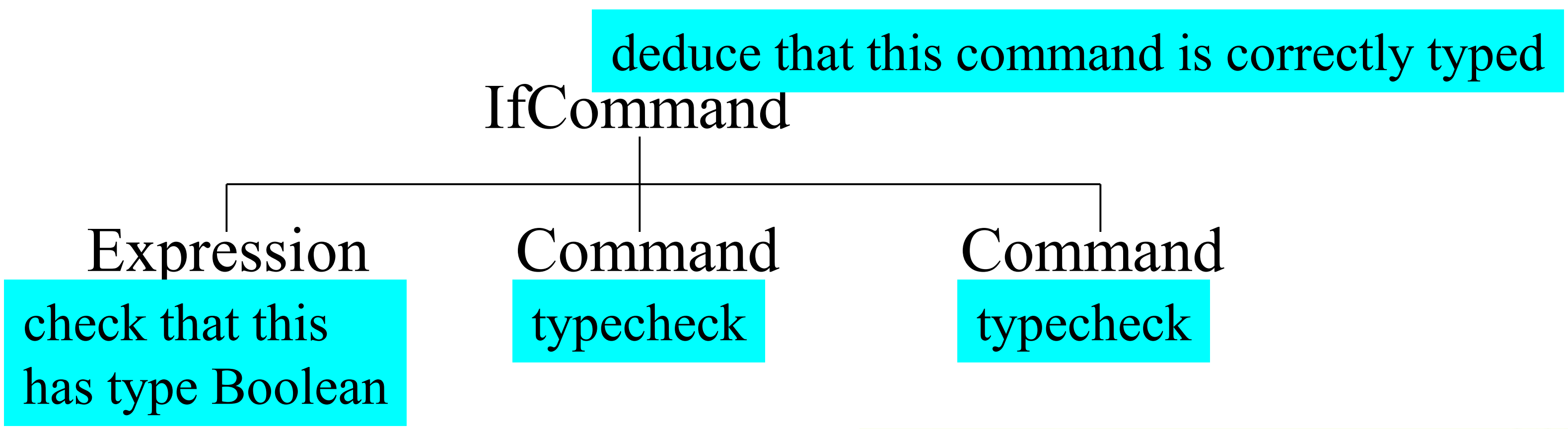
Type checking

Commands which contain expressions:

Type rule of **IfCommand**:

if *E* **do** *C1* **else** *C2* is type correct

if *E* of type **Boolean** and *C1* and *C2* are type correct

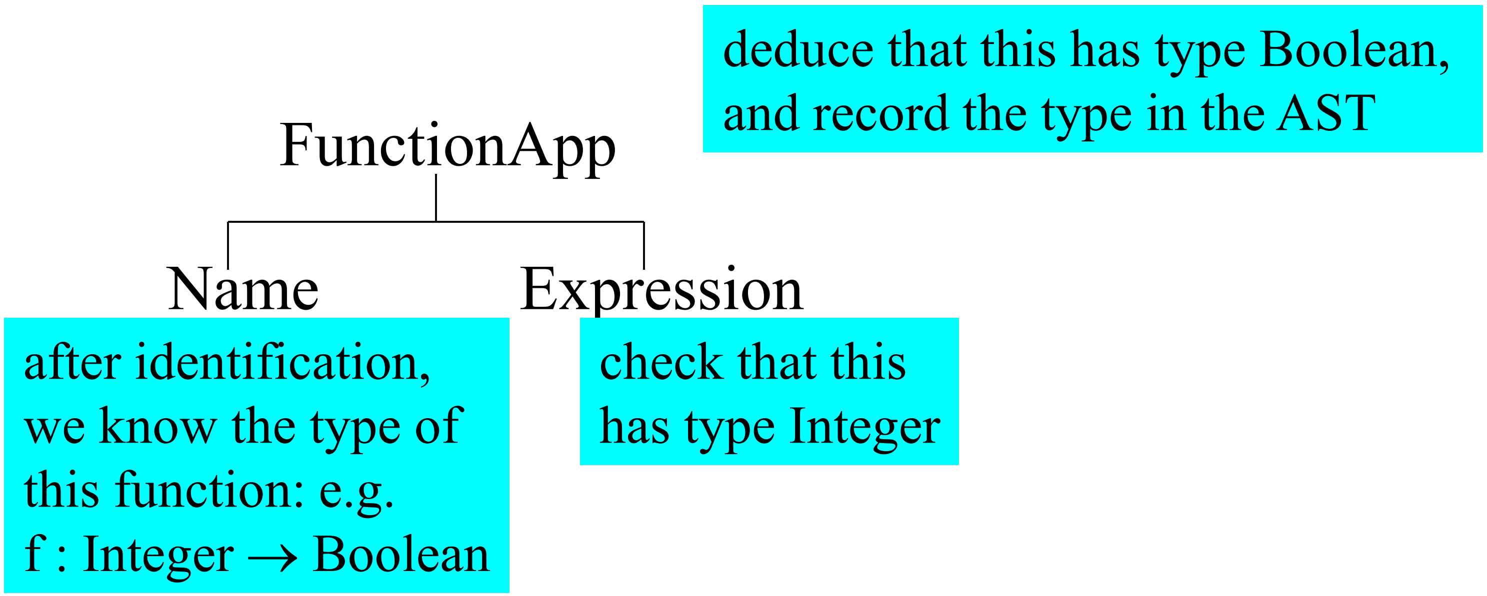


WhileCommand is similar.

$$[\text{IFSTM}] \quad \frac{E \vdash e : \mathsf{Bool} \quad E \vdash S_1 : \mathsf{ok} \quad E \vdash S_2 : \mathsf{ok}}{E \vdash \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else;}\ S_2 : \mathsf{ok}}$$

Type checking

Function applications:



Type checking

Function definitions:

func f(x : ParamType) : ResultType ~ Expression

Typecheck the function body and calculate its type.
Check that the type is ResultType.
Then deduce
f : ParamType → ResultType
e.g.
f : Integer → Boolean

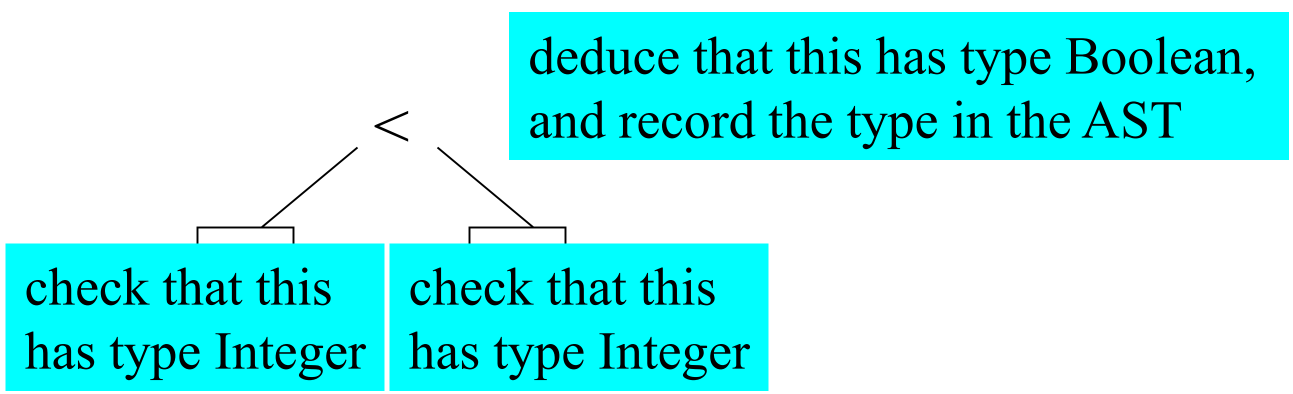
Type checking

Operators in expressions (again):

For each operator we know that the operands must have certain types, and that the result has a certain type. This information can be represented by giving the operators function types:

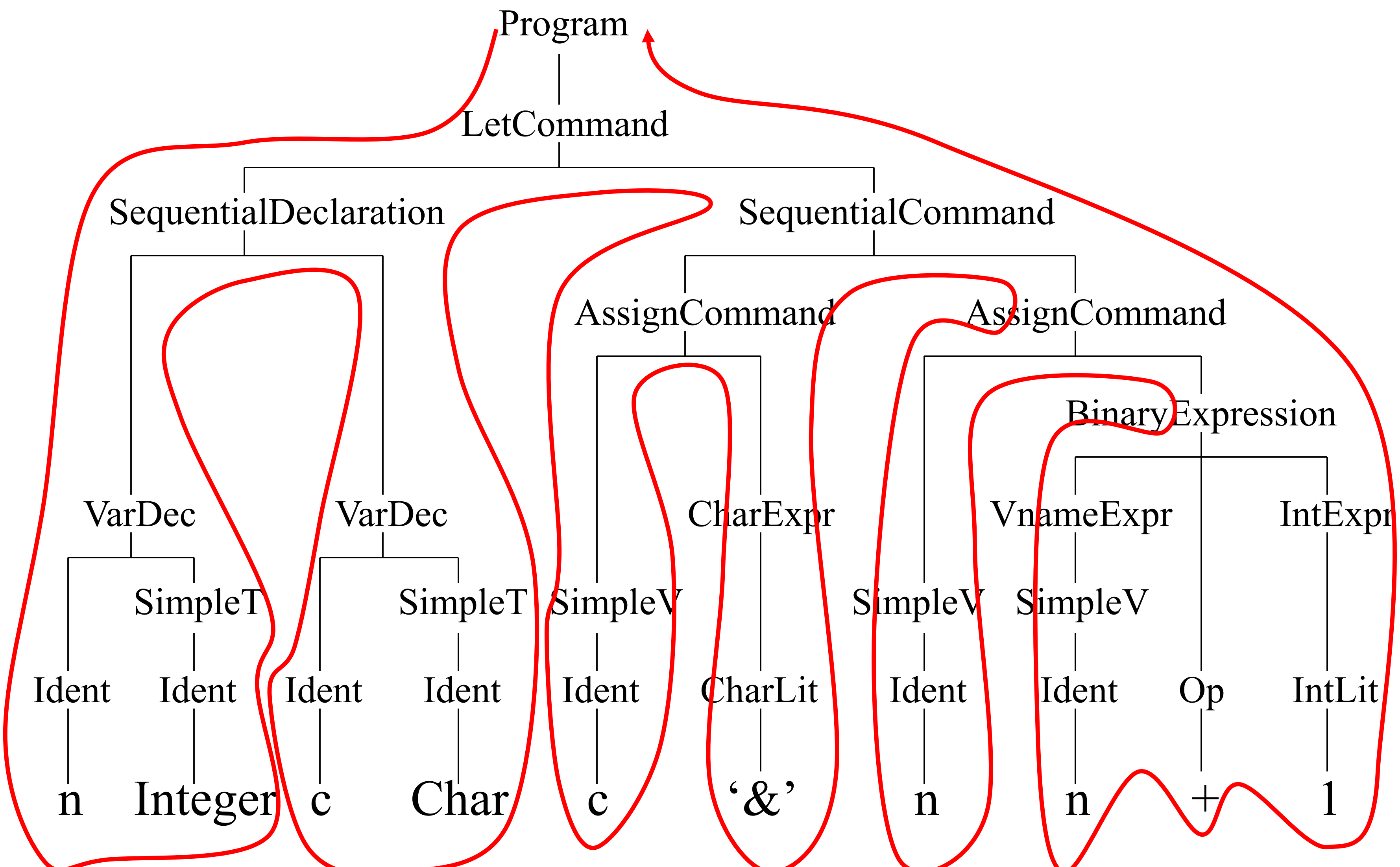
$$+ : \text{Integer} \times \text{Integer} \rightarrow \text{Integer}$$

$$< : \text{Integer} \times \text{Integer} \rightarrow \text{Boolean}$$



Contextual Analysis

Identification and type checking are combined into a depth-first traversal of the AST.



An example using GOF visitor

- Implementation of Mini Triangle Contextual Analyzer
  – Programming Language Processors in Java Compilers and Interpreters

- Full working example in Java
  – http://www.dcs.gla.ac.uk/~daw/books/PLPJ/Triangle-tools-2.1.zip
  – Full working version in C# in General Course Materials on Moodle

Example: Implementation of Mini Triangle Contextual Analyzer

Mini Triangle Abstract Syntax

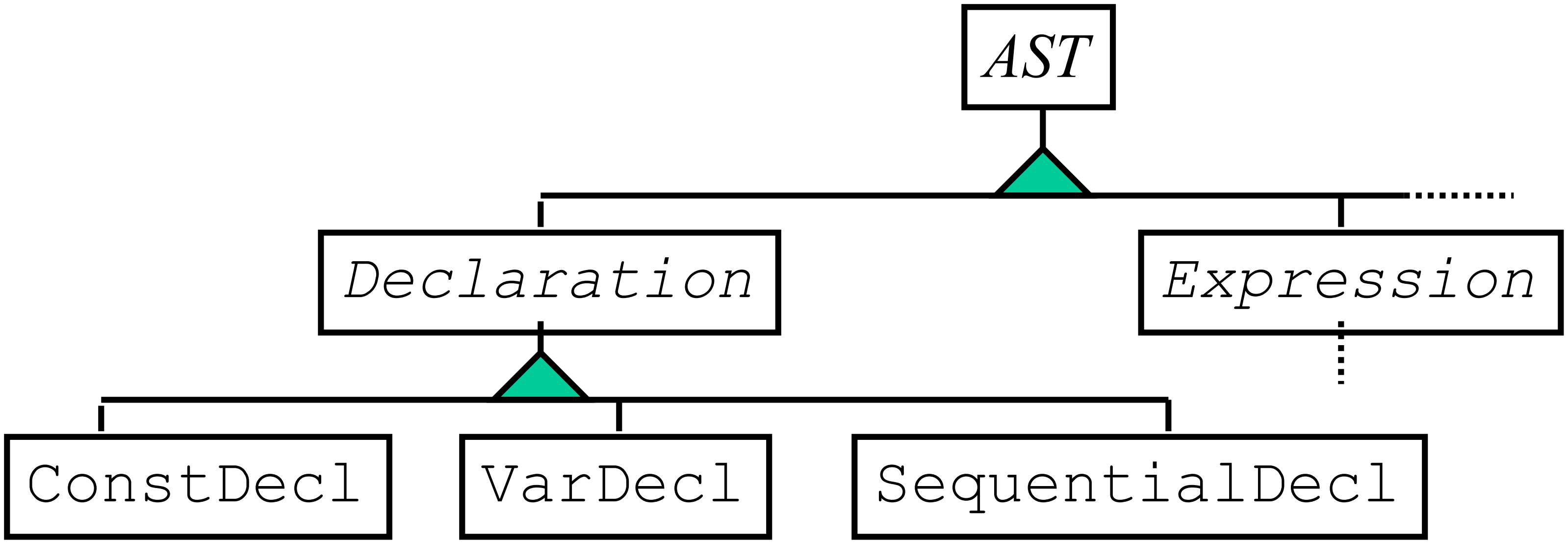
`Program ::= Command`	Program
`Command`	
`  ::= V-name := Expression`	AssignCmd
`    | Identifier ( Expression )`	CallCmd
`    | if Expression then Command`	
`                      else Command`	IfCmd
`    | while Expression do Command`	WhileCmd
`    | let Declaration in Command`	LetCmd
`    | Command ; Command`	SequentialCmd
`V-name ::= Identifier`	SimpleVName
`...`	

RECAP: Mini Triangle Abstract Syntax (ctd)

```
Declaration
   ::= const Identifier ~ Expression   ConstDecl
     | var Identifier : TypeDenoter    VarDecl
     | Declaration ; Declaration       SequentialDecl

TypeDenoter ::= Identifier             SimpleTypeDenoter

Expression
   ::= Integer-Literal                 IntegerExpression
     | V-name                          VnameExpression
     | Operator Expression             UnaryExpression
     | Expression Op Expression        BinaryExpression
```

RECAP: AST representation (ctd)

```
Declaration
  ::= const Identifier ~ Expression         ConstDecl
    | var Identifier : TypeDenoter          VarDecl
    | Declaration ; Declaration             SequentialDecl
```



RECAP: AST representation (ctd)

```
Declaration
   ::= const Identifier ~ Expression          ConstDecl
     | var Identifier : TypeDenoter           VarDecl
     | Declaration ; Declaration              SequentialDecl
```

```
public class ConstDecl extends Declaration {
  public Identifier I;          // constant name
  public Expression E;          // constant value
  ...
}
public class VarDecl extends Declaration {
 ...


...
```

Representing the Decorated AST (in Java)

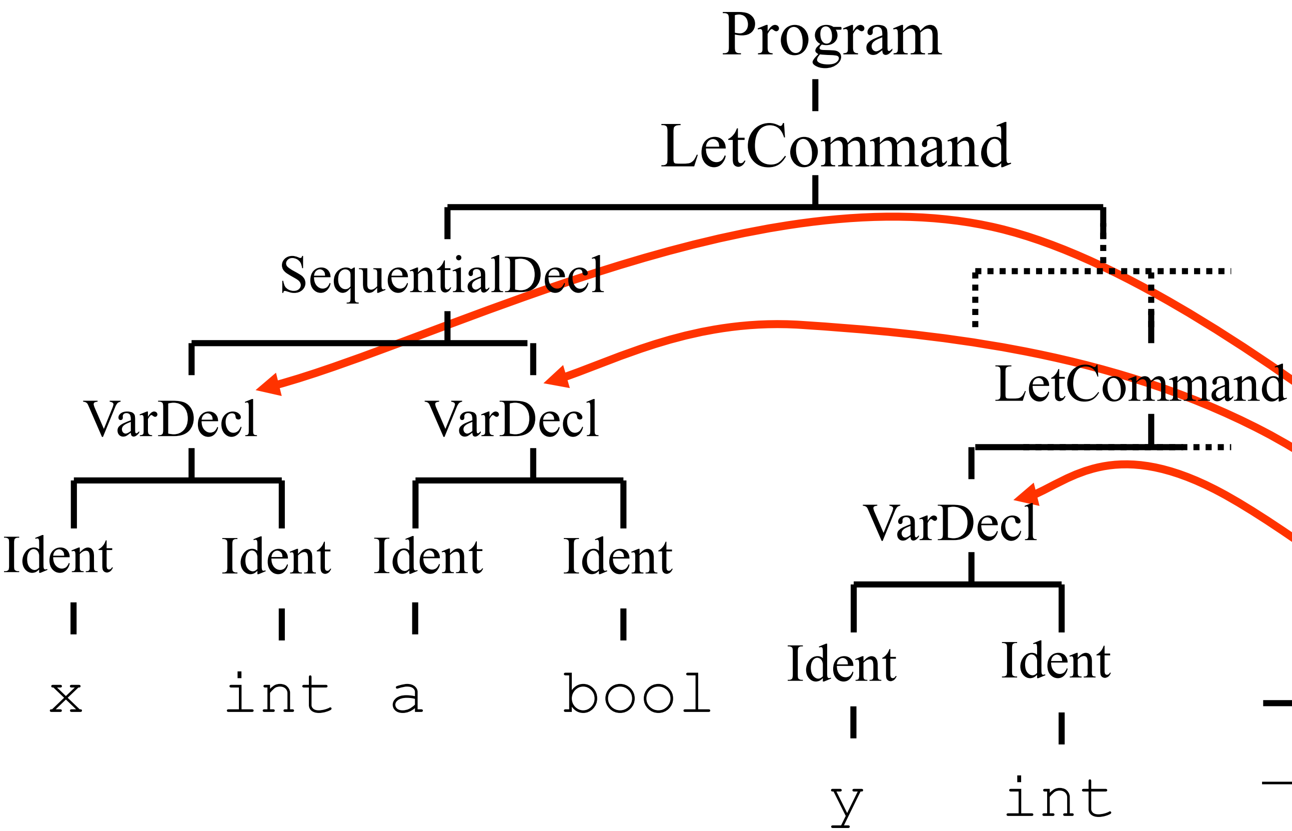
1) We add some instance variables to some of the AST node classes.

```
public abstract class Expression extends AST {
  // Every type-correct expression has a static type
  public Type type;
  ...
}
```

```
public class Identifier extends Token {
  // For applied occurrences only: where was this id declared?
  public Declaration decl;
  ...
}
```

...

Attributes as pointers to Declaration AST's



Id table

Level	Ident	Attr
1	x	•
1	a	•
2	y	•

Representing the Decorated AST (in Java)

…

```
public abstract class VName extends AST {
  // The type of this variable or constant name
  public Type type;
  // Is it a variable? (otherwise it is a constant)
  public boolean variable;
}
```

Traversal over the AST: Visitor Design Pattern

```
public interface Visitor {
  // Programs
  public Object visitProgram(Program p,Object arg);

  // Commands
  public Object visitAssignCommand
        (AssignCommand c,Object arg);
  public Object visitCallCommand
        (CallCommand c,Object arg);
  ...
  // Expressions
  public Object visitVnameExpression
        (VnameExpression e,Object arg);
  public Object visitUnaryExpression
        (UnaryExpression e,Object arg);
  ...
```

Traversal may compute a value

For passing extra arguments/info to a traversal

Traversal over the AST: Visitor Design Pattern

```
public abstract class AST {
  ...
  public abstract Object visit(Visitor v,Object arg);
}
```

In every **concrete** AST class add:

```
public class AssignCommand extends AST {
  ...
  public Object visit(Visitor v,Object arg) {
    return v.visitAssignCommand(this,arg);
  }
}
public class IfCommand extends AST {
  ...
  public Object visit(Visitor v,Object arg) {
    return v.visitIfCommand(this,arg);
  }
}
```

Example: Implementation of Mini Triangle Contextual Analyzer

Mini Triangle Types

```
public class Type {
  private byte kind; // INT, BOOL or ERROR
  public static final byte
    BOOL=0, INT=1, ERROR=-1;

  private Type(byte kind) { ... }

  public boolean equals(Object other) { ... }

  public static Type boolT = new Type(BOOL);
  public static Type intT  = new Type(INT);
  public static Type errorT  = new Type(ERROR);
}
```

Example: Implementation of Mini Triangle Contextual Analyzer

Contextual Analyzer as an AST visitor

```
public class Checker implements Visitor {

  private IdentificationTable idTable;

  public void check(Program prog) {
    idTable = new IdentificationTable();
    // initialize with standard environment
    idTable.enter("false",...);
    ...
    idTable.enter("putint",...);
    prog.visit(this,null);
  }

  ...
```

Checker is a traversal of AST

Start AST traversal with **this** checker

What the Checker Visitor Does

visitProgram	Check whether program is well-formed and return **null.**
visit…Command	Check whether the command is well-formed and return **null**.
visit…Expression	Check expression, decorate it with its type and return the type.
visitSimpleVName	Check whether name is declared. Decorate it with its type and a flag whether it is a variable. Return its type.
visit…Declaration	Check that declaration is well-formed. Enter declared identifier into ID table. Return **null**.
visitSimpleTypeDen	Check that type denoter is well-formed. Decorate with its type. Return the type.
visitIdentifier	Check whether identifier is declared. Decorate with link to its declaration. Return declaration.

Example: Implementation of Mini Triangle Contextual Analyzer

```
public class Checker implements Visitor {
 ...

 //Checking commands

 public Object visitAssignCommand (AssignCommand com,Object arg)
 {
    Type vType = (Type) com.V.visit(this,null);
    Type eType = (Type) com.E.visit(this,null);
    if (! com.V.variable)
       report error: v is not a variable
    if (! eType.equals(vType) )
       report error incompatible types in assignCommand
    return null;
  }

  ...
```

Example: Implementation of Mini Triangle Contextual Analyzer

```
...

public Object visitIfCommand (IfCommand com,Object arg)
{
  Type eType = (Type)com.E.visit(this,null);
  if (! eType.equals(Type.boolT) )
    report error: expression in if not boolean
  com.C1.visit(this,null);
  com.C2.visit(this,null);
  return null;
}

...
```

Example: Implementation of Mini Triangle Contextual Analyzer

```
...
public Object visitSequentialCommand
    (SequentialCommand com,Object arg)
{
  com.C1.visit(this,null);
  com.C2.visit(this,null);
}

public Object visitLetCommand (LetCommand com,Object arg)
{
  idTable.openScope();
  com.D.visit(this,null); // enters declarations into idTable
  com.C.visit(this,null);
  idTable.closeScope();
  return null;
}

...
```

Example: Implementation of Mini Triangle Contextual Analyzer

```
// Expression Checking
public Object visitIntegerExpression
     (IntegerExpression expr,Object arg)
{
  expr.type = Type.intT;  // decoration
  return expr.type;
}

public Object visitVnameExpression
     (VnameExpression expr,Object arg)
{
  Type vType = (Type) expr.V.visit(this,null);
  expr.type = vType; // decoration
  return expr.type;
}
```

Example: Implementation of Mini Triangle Contextual Analyzer

```
public Object visitBinaryExpression
      (BinaryExpression expr,Object arg) {
   Type e1Type = expr.E1.visit(this,null);
   Type e2Type = expr.E2.visit(this,null);
   OperatorDeclaration opdecl =
      (OperatorDeclaration) expr.O.visit(this,null);
   if (opdecl==null) {
       // error: operator not defined
       expr.type = Type.error;
   } else if (opdecl instanceof BinaryOperatorDecl) {
      // check binary operator
   } else {
      // error: operator not binary
      expr.type = Type.errorT;
   }
   return expr.type;
}
```

Example: Implementation of Mini Triangle Contextual Analyzer

```
public Object visitBinaryExpression
      (BinaryExpression expr,Object arg) {
   ...
   } else if (opdecl instanceof BinaryOperatorDecl) {
       BinaryOperatorDecl bopdecl =
          (BinaryOperatorDecl) opdecl;
       if (! e1Type.equals(bopdecl.operand1Type) )
          // error: first argument wrong type
       if (! e2Type.equals(bopdecl.operand2Type) )
          // error: second argument wrong type
       expr.type = bopdecl.resultType;
   } else {
     // error: operator not binary
     ...
   }
   return expr.type;
}
```

Example: Implementation of Mini Triangle Contextual Analyzer

```
// Declaration checking
public Object visitVarDeclaration
      (VarDeclaration decl,Object arg) {
   decl.T.visit(this,null);
   idTable.enter(decl.I.spelling,decl);
   return null;
}

public Object visitConstDeclaration
      (ConstDeclaration decl,Object arg) {
   decl.E.visit(this,null);
   idTable.enter(decl.I.spelling,decl);
   return null;
}

...
```

Implementing type checking from type rules

(conditional)

$$\frac{\Gamma \mid\!- E\colon \text{bool},\ \Gamma \mid\!- C_1\colon T,\ \Gamma \mid\!- C_2\colon T}{\Gamma \mid\!- \text{if } E \text{ then } C_1 \text{ else } C_2\colon T}$$

```
public Object visitIfExpression (IfExpression com,Object arg)
{
  Type eType = (Type)com.E.visit(this,null);
  if (! eType.equals(Type.boolT) )
    report error: expression in if not boolean
  Type c1Type = (Type)com.C1.visit(this,null);
  Type c2Type = (Type)com.C2.visit(this,null);
  if (! c1Type.equals(c2Type) )
    report error: type mismatch in expression branches
  return c1Type;
}
```

Implementing type checking from type rules

(conditional)

$$\frac{\Gamma \mid\!- E: T_E,\ T_E = \text{bool},\ \Gamma \mid\!- C_1: T_1,\ \Gamma \mid\!- C_2: T_2,\ T_1 = T_2}{\Gamma \mid\!- \text{if } E \text{ then } C_1 \text{ else } C_2: T_1}$$

```
public Object visitIfExpression (IfExpression com,Object arg)
{
  Type eType = (Type)com.E.visit(this,null);
  if (! eType.equals(Type.boolT) )
    report error: expression in if not boolean
  Type c1Type = (Type)com.C1.visit(this,null);
  Type c2Type = (Type)com.C2.visit(this,null);
  if (! c1Type.equals(c2Type) )
    report error: type mismatch in expression branches
  return c1Type;
}
```

Pause

Implementing Tree Traversal

- "Traditional" OO approach
- Visitor approach
  - GOF
  - Using static overloading
  - Reflective
  - (dynamic)
  - (SableCC style)
- "Functional" approach
- Active patterns in Scala (or F#)
- (Aspect oriented approach)

```
1  Start  → Stmt $
2  Stmt   → id assign E
3         | if lparen E rparen Stmt else Stmt fi
4         | if lparen E rparen Stmt fi
5         | while lparen E rparen do Stmt od
6         | begin Stmts end
7  Stmts  → Stmts semi Stmt
8         | Stmt
9  E      → E plus T
10        | T
11 T      → id
12        | num
```

Figure 7.14: Grammar for a simple language.

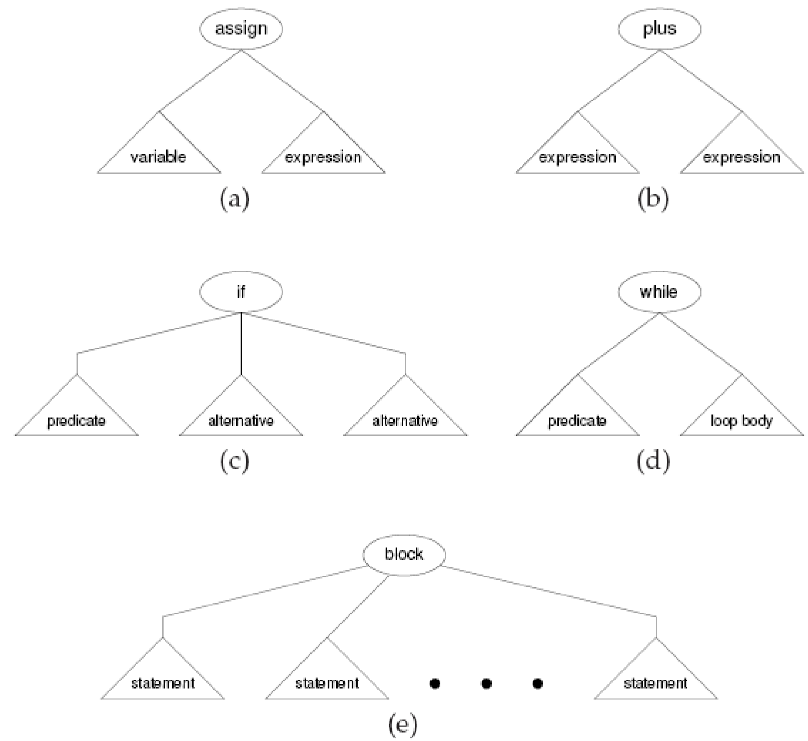


Figure 7.15: AST structures: A specific node is designated by an ellipse. Tree structure of arbitrary complexity is designated by a triangle.

```
class IfNode extends AbstractNode
    procedure typeCheck( )
        /*    Type-checking code for an if                    */
    end
    procedure codeGen( )
        /*    Generate code for an if                         */
    end
    ...
end

class PlusNode extends AbstractNode
    procedure typeCheck( )
        /*    Type-checking code for a plus                   */
    end
    procedure codeGen( )
        /*    Generate code for a plus                        */
    end
    ...
end
...
```

Figure 7.25: Inferior design: phase code distributed among node types.

```
foreach AbstractNode n ∈ AST do
    switch (n.getType( ))
        case IfNode
            call f.visit(⟨IfNode ⇓ n⟩)
        case PlusNode
            call f.visit(⟨PlusNode ⇓ n⟩)
        case MinusNode
            call f.visit(⟨MinusNode ⇓ n⟩)
```

Figure 7.26: An alternative for achieving double dispatch.

```
class Visitor
    /*    Generic visit                                  */
    procedure VISIT( AbstractNode n )                    (28)
        n.ACCEPT( this )                                 (29)
    end
end
class TypeChecking extends Visitor                       (30)
    procedure VISIT( IfNode i )
    end
    procedure VISIT( PlusNode p )
    end
    procedure VISIT( MinusNode m )
    end
end

class IfNode extends AbstractNode
    procedure ACCEPT( Visitor v )                        (31)
        v.VISIT( this )
    end
    ...
end
class PlusNode extends AbstractNode
    procedure ACCEPT( Visitor v )                        (32)
        v.VISIT( this )                                  (33)
    end
    ...
end
class MinusNode extends AbstractNode
    procedure ACCEPT( Visitor v )                        (34)
        v.VISIT( this )
    end
    ...
end
```

Figure 7.23: Visitor pattern

```
class ReflectiveVisitor
    /*   Generic visit                                              */
    procedure VISIT( AbstractNode n )                                   (35)
        this.DISPATCH( n )                                              (36)
    end
    procedure DISPATCH( Object o )
        /*   Find and invoke the VISIT( n ) method                  */
        /*   whose declared parameter n is the closest match        */
        /*   for the actual type of o.                              */
    end
    procedure DEFAULTVISIT( AbstractNode n )                            (37)
        foreach AbstractNode c ∈ Children(n) do this.VISIT( c )
    end
end
class IfNode                                                            (38)
    extends AbstractNode
    implements { NeedsBooleanPredicate }
end
class WhileNode                                                         (39)
    extends AbstractNode
    implements { NeedsBooleanPredicate }
end
class PlusNode
    extends AbstractNode
    implements { NeedsCompatibleTypes }
end
class TypeChecking extends ReflectiveVisitor                            (40)
    procedure VISIT( NeedsBooleanPredicate nbp )                        (41)
        /*   Check the type of nbp.GETPREDICATE( )                  */
    end
    procedure VISIT( NeedCompatibleTypes nct )                          (42)
    end
    procedure VISIT( NeedsLeftChildType nlct )                          (43)
    end
end
```

Figure 7.24: Reflective Visitor

Consequences of using Visitor

- Addition of new operations is easy
  - New operations can be created by simply adding a new visitor
- Gathers related operations together
  - All operation related code is in the visitor
  - Code for different operations are in different sub-classes of visitor
  - Unrelated operations are not mixed together in the object classes
- Adding a new concrete type in the object structure is hard
  - Each Visitor has to be recompiled with an appropriate method for the new type

Flavours of the Visitor Pattern

- Traditional OO style
  - actASTtraditionalOO
- GOF style
  - acASTGOFVisitor
- Exploiting static overloading
  - acASTVisitor
- Reflective Visitor
  - acASTreflective

Full working versions in
General Course Materials
On Moodle

Type Checking Using Reflective Visitor

- Using the visitor pattern (in Chap. 7)
  - SemanticsVisitor: a subclass of Visitor
    - The top-level visitor for processing declarations and doing semantic checking on the AST nodes
  - TopDeclVisitor
    - A specialized visitor invoked by SemanticsVisitor for processing declarations
  - TypeVisitor
    - A specialized visitor used to handle an identifier that represents a type or a syntactic form that defines a type (such as an array)

An abstract java like OO language

Program -> ClassDeclaration *

ClassDeclaration -> **class** Modifiers Name **extends** Parent **{** Fields* Constructor* Method* **}**

Fields -> Type Name*

Constructor -> ..

Method -> Modifiers Type Name **(** Parameter* **){** Statement* **}**

Statement -> Assignment
| ..
| IfTesting
| WhileLooping
| DoWhileLooping
| ForLooping
| Continuing | Breaking | Returning | Switching | Label Statement

IfTesting -> **if** Exp **then** Statement **else** Statement

```
class NodeVisitor
    procedure visitChildren(n)
        foreach c ∈ n.getChildren() do
            call c.accept(this)                                          (11)
    end
end

class SemanticsVisitor extends NodeVisitor
    /*   visit methods for other node types are defined in Section 8.8   */
end

class TopDeclVisitor extends SemanticsVisitor
    procedure visit(VariableListDeclaring vld)                           (12)
        /*   Section 8.6.1 on page 303                                    */
    end
    procedure visit(TypeDeclaring td)                                    (13)
        /*   Section 8.6.3 on page 305                                    */
    end
    procedure visit(ClassDeclaring cd)                                   (14)
        /*   Section 8.7.1 on page 317                                    */
    end
    procedure visit(MethodDeclaring md)                                  (15)
        /*   Section 8.7.2 on page 321                                    */
    end
end

class TypeVisitor extends TopDeclVisitor
    procedure visit(Identifier id)                                       (16)
        /*   Section 8.6.2 on page 304                                    */
    end
    procedure visit(ArrayDefining arraydef)                              (17)
        /*   Section 8.6.5 on page 311                                    */
    end
    procedure visit(StructDefining structdef)                            (18)
        /*   Section 8.6.6 on page 312                                    */
    end
    procedure visit(EnumDefining enumdef)                                (19)
        /*   Section 8.6.7 on page 313                                    */
    end
end
```

Figure 8.11: Structure of the declarations visitors, with references to sections addressing specific constructs.

Variable and Type Declarations

- Simple variable declarations
  - A type name and a list of identifiers
    - Visitor actions: (Fig. 8.13)

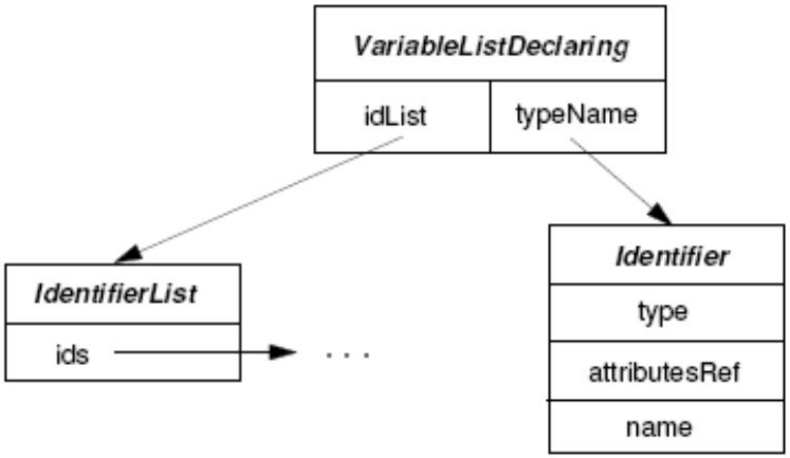


Figure 8.12: Abstract Syntax Tree for Variable Declarations

```
/*   Visitor code for Marker (12) on page 302                          */
procedure visit( VariableListDeclaring vld )
    typeVisitor ← new TypeVisitor( )                                  (20)
    call vld.typeName.accept( typeVisitor )                           (21)
    foreach id ∈ vld.idList do                                        (22)
        if currentSymbolTable.declaredLocally( id.name )              (23)
        then
            call error( "This variable is already declared : ", id.name )
            id.type ← errorType
            id.attributesRef ← null
        else
            id.type ← vld.typeName.type                               (24)
            attr.kind ← variableAttributes
            attr.variableType ← id.type
            id.attributesRef ← attr
            call currentSymbolTable.enterSymbol( id.name, attr )
end
```

Figure 8.13: visit method in TopDeclVisitor for VariableListDeclaring.

Handling Type Names

```
/*   Visitor code for Marker (16) on page 302                          */
procedure VISIT( Identifier id )
    attr ← currentSymbolTable.RETRIEVESYMBOL( id.name )                 (25)
    if attr ≠ null and attr.kind = typeAttributes
    then
        id.type ← attr.thisType                                         (26)
        id.attributesRef ← attr
    else
        call ERROR("This identifier is not a type name : ", id.name)    (27)
        id.type ← errorType
        id.attributesRef ← null
end
```

Figure 8.14: VISIT method in TypeVisitor for Identifier.

Type Declarations

- A name and a description of the type to be associated with it
  - Visit method: (Fig. 8.16)

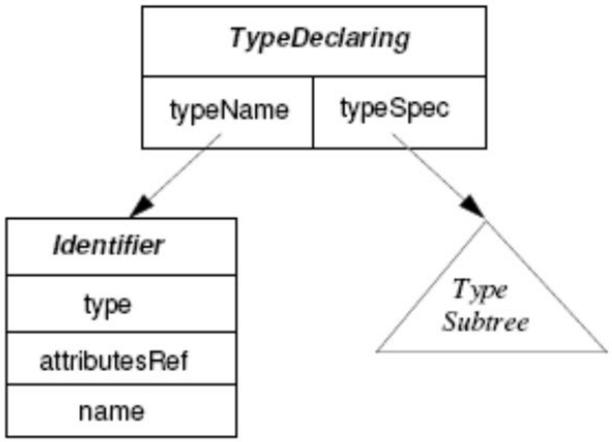


Figure 8.15: Abstract Syntax Tree for Type Declarations

```
/*    Visitor code for Marker (13) on page 302                          */
procedure VISIT( TypeDeclaring td )
    typeVisitor ← new TypeVisitor( )                                    (28)
    call td.typeSpec.ACCEPT(typeVisitor)                                (29)
    name ← td.typeName.name                                             (30)
    if currentSymbolTable.DECLAREDLOCALLY(name)                         (31)
    then
        call ERROR("This identifier is already declared : ", name)
        td.typeName.type ← errorType
        td.typeName.attributesRef ← null
    else
        attr ← new Attributes(typeAttributes)                           (32)
        attr.thisType ← td.typeSpec.type
        call currentSymbolTable.ENTERSYMBOL(name, attr)
        td.typeName.type ← td.typeSpec.type
        td.typeName.attributesRef ← attr
end
```

Figure 8.16: VISIT method in TopDeclVisitor for TypeDeclaring.

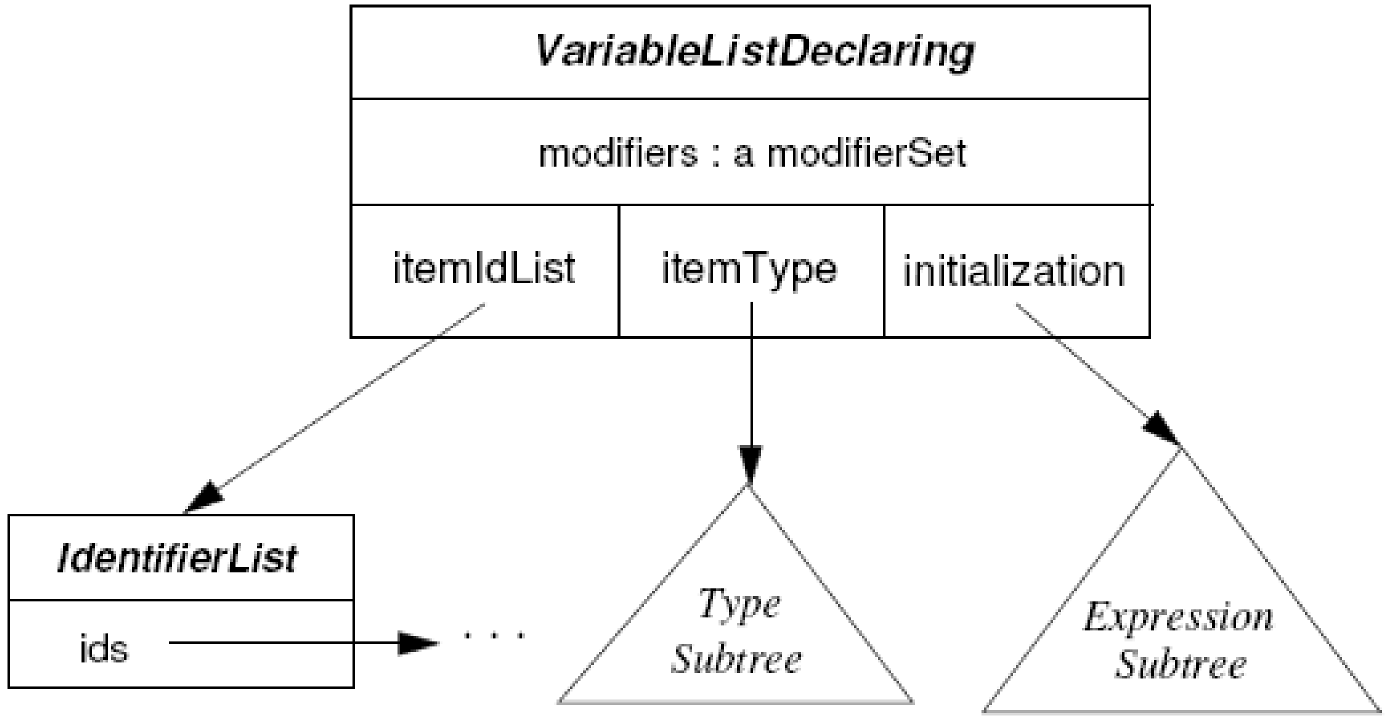


Figure 8.17: AST for Generalized Variable Declarations

```
/*  Generalized visitor code for Marker (12)                                   */
procedure VISIT( VariableListDeclaring vld )
    typeVisitor ← new TypeVisitor( )                                          (33)
    call vld.itemType.ACCEPT( typeVisitor )
    declType ← vld.itemType.type
    if vld.initialization ≠ null                                              (34)
    then
        checkingVisitor ← new SemanticsVisitor( )
        call vld.initialization.ACCEPT( checkingVisitor )
        if not ASSIGNABLE( vld.initialization.type, declType )                (35)
        then
            call ERROR( "Initialization expression not assignable to variable type at", vld )
    else
        if const ∈ vld.modifiers                                              (36)
        then
            call ERROR( "Initialization expression missing in constant declaration at", vld )
    foreach id ∈ vld.itemIdList do
        if currentSymbolTable.DECLAREDLOCALLY( id.name )
        then
            call ERROR( "Variable name cannot be redeclared : ", id.name )
            id.type ← errorType
            id.attributesRef ← null
        else
            attr.kind ← variableAttributes
            attr.variableType ← declType
            attr.modifiers ← declType                                         (37)
            call currentSymbolTable.ENTERSYMBOL( id.name, attr )
            id.type ← declType
            id.attributesRef ← attr
end
```

Figure 8.18: Code for TopDeclVisitor's VariableListDeclaring

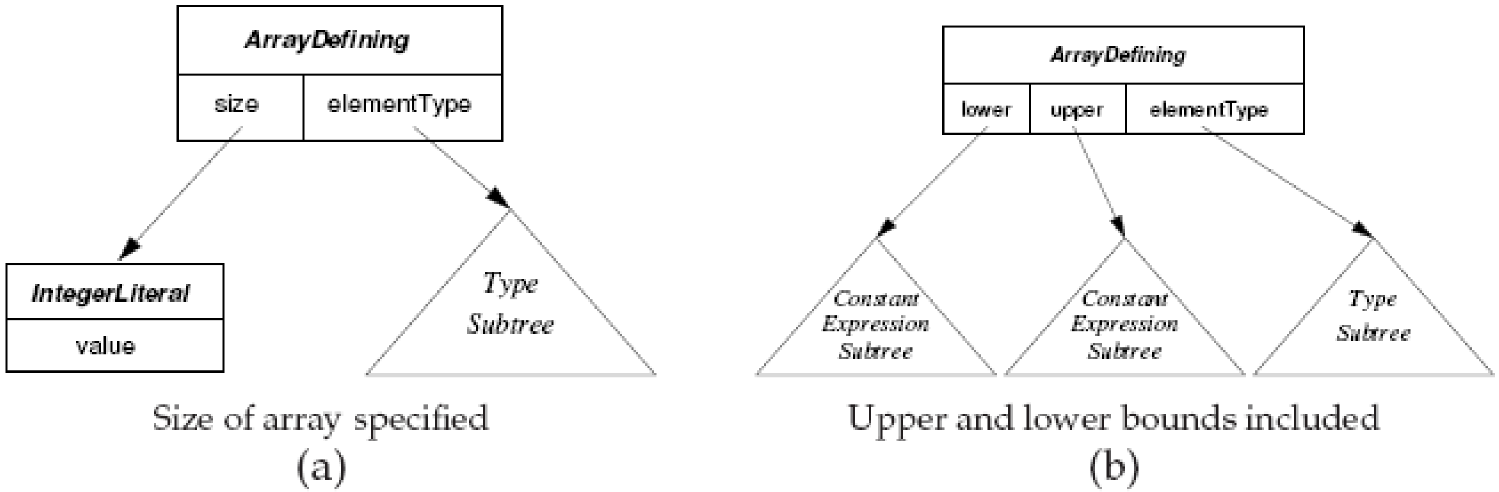


Figure 8.19: Abstract Syntax Trees for Array Definitions

/⋆ Visitor code for Marker (17) on page 302 ⋆/

procedure VISIT(*ArrayDefining* *arraydef*)
  call VISITCHILDREN(*arraydef*)
  arraydef.type ← **new** *TypeDescriptor*(*arrayType*) (38)
  arraydef.type.elementType ← *arraydef.elementType.type*
  arraydef.type.arraysize ← *arraydef.size.value*
end

Figure 8.20: VISIT method in TypeVisitor for ArrayDefining.

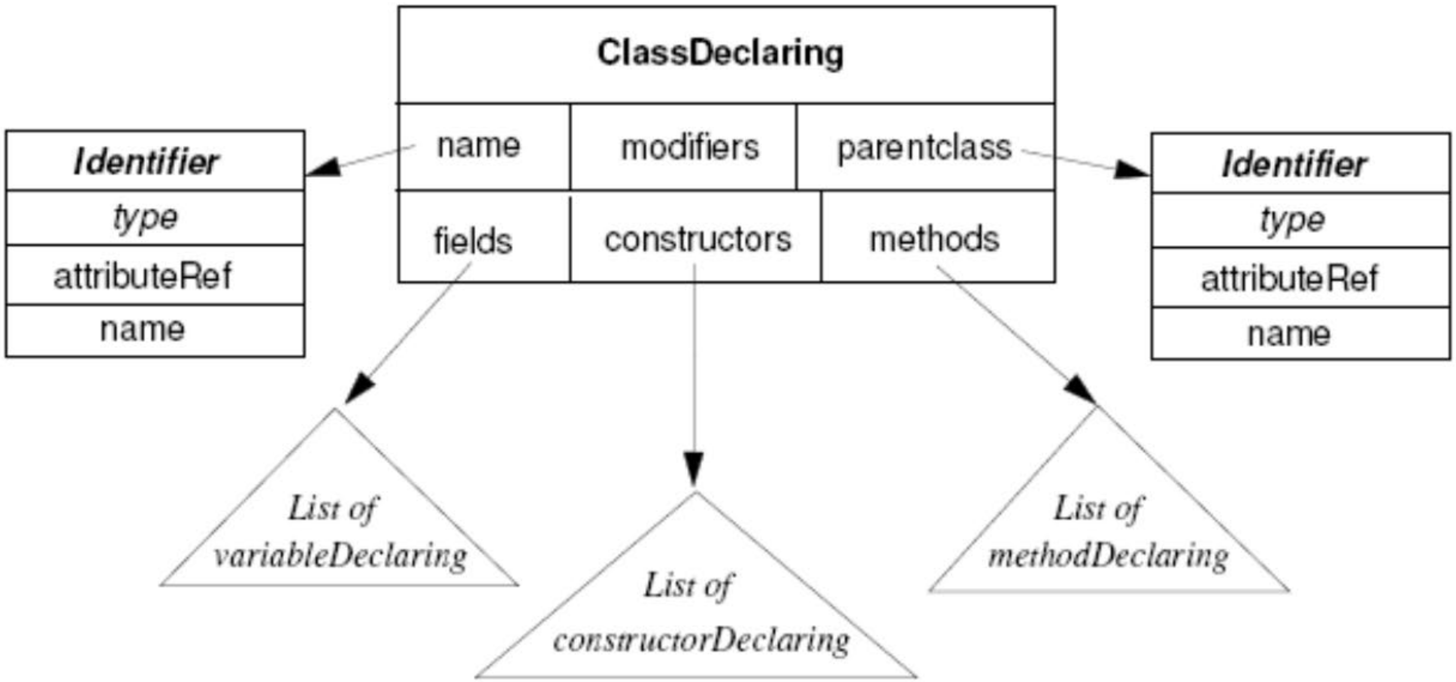


Figure 8.27: Abstract Syntax Tree for a Class Declaration

```
/*   Visitor code for Marker (14) on page 302                              */
procedure VISIT( ClassDeclaring cd )
    typeRef ← new TypeDescriptor( ClassType )                              (51)
    typeRef.names ← new SymbolTable( )
    attr ← new Attributes( ClassAttributes )
    attr.classType ← typeRef
    call currentSymbolTable.ENTERSYMBOL( name.name, attr )
    call SETCURRENTCLASS( attr )
    if cd.parentclass = null                                               (52)
    then  cd.parentclass ← GETREFTOOBJECT( )
    else
        typeVisitor ← new TypeVisitor( )
        call cd.parentclass.ACCEPT( typeVisitor )
    if cd.parentclass.type = errorType
    then  attr.classtype ← errorType
    else
        if cd.parentclass.type.kind ≠ classType
        then
            attr.classtype ← errorType
            call ERROR( parentClass.name, "does not name a class" )
        else
            typeRef.parent ← cd.parentClass.attributeRef                   (53)
            typeRef.isFinal ← MEMBEROF( cd.modifiers, final )
            typeRef.isAbstractl ← MEMBEROF( cd.modifiers, abstract )
            call typeRef.names.INCORPORATE( cd.parentclass.type.names )    (54)
            call OPENSCOPE( typeRef.names )
            call cd.fields.ACCEPT( this )                                  (55)
            call cd.constructors.ACCEPT( this )
            call cd.methods.ACCEPT( this )
            call CLOSESCOPE( )
    call SETCURRENTCLASS( null )
end
```

Figure 8.29: VISIT method in TopDeclVisitor forClassDeclaring

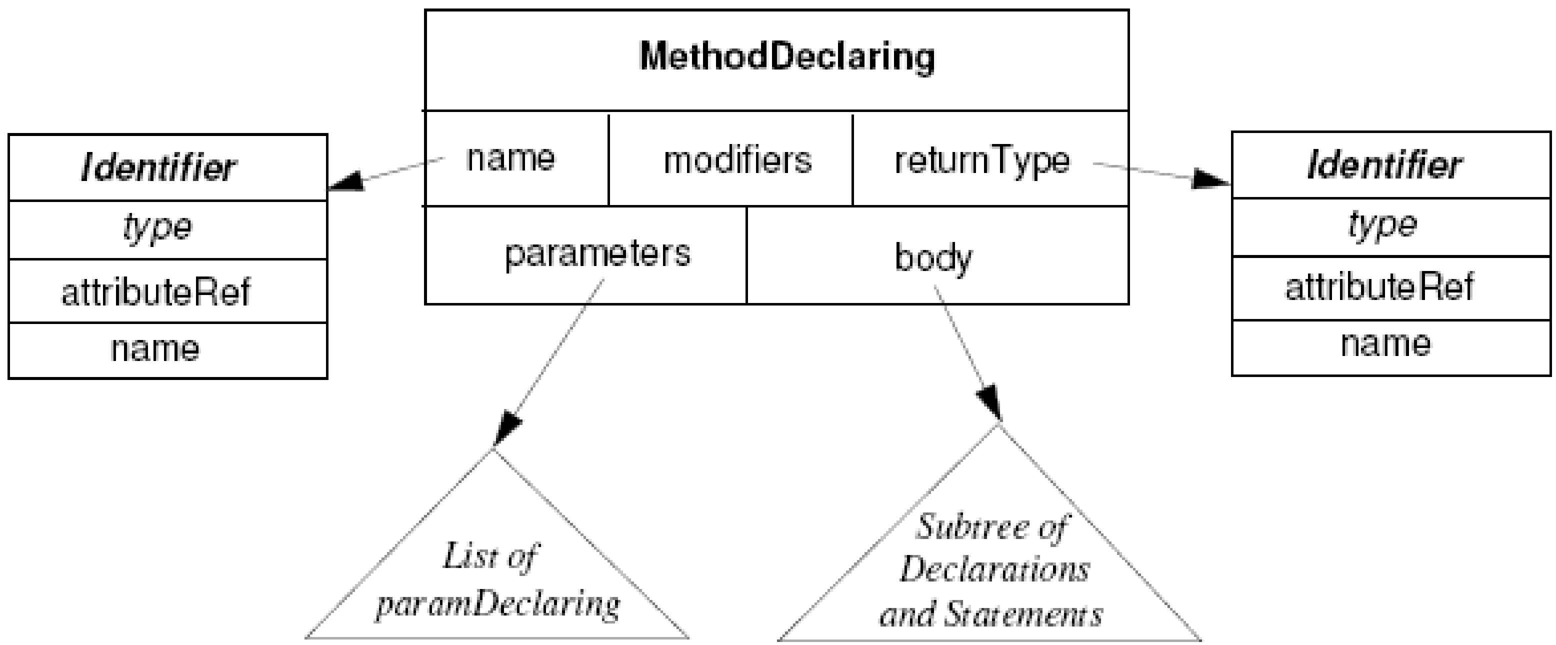


Figure 8.30: Abstract Syntax Tree for a Method Declaration

```
/*   Visitor code for Marker (15) on page 302                          */
procedure VISIT( MethodDeclaring md )
    typeVisitor ← new TypeVisitor( )                                      (56)
    call md.returnType.ACCEPT( typeVisitor )
    attr ← new Attributes( MethodAttributes )
    attr.returnType ← md.returnType.type
    attr.modifiers ← md.modifiers
    attr.isDefinedIn ← GETCURRENTCLASS( )
    attr.locals ← new SymbolTable( )
    call currentSymbolTable.ENTERSYMBOL( name.name, attr )
    md.name.attributeRef ← attr
    call OPENSCOPE( attr.locals )
    oldCurrentMethod ← GETCURRENTMETHOD( )
    call SETCURRENTMETHOD( attr )
    call md.parameters.ACCEPT( this )                                     (57)
    attr.signature ← parameters.signature.ADDRETURN( attr.returntype )
    call md.body.ACCEPT( this )                                           (58)
    call SETCURRENTMETHOD( oldCurrentMethod )
    call CLOSESCOPE( )
end
```

Figure 8.31: VISIT method in TopDeclVisitor for MethodDeclaring.

```
class NodeVisitor
    procedure visitChildren(n)
        foreach c ∈ n.getChildren() do call c.accept(this)
    end
end

class SemanticsVisitor extends NodeVisitor
    procedure checkBoolean(c)                                        (1)
        if c.type ≠ Boolean and c.type ≠ errorType
        then call error("Require Boolean type at", c)
    end

    procedure visit(IfTesting ifn)                                   (2)
        call visitChildren(ifn)
        call checkBoolean(ifn.condition)
    end

    procedure visit(WhileLooping wn)                                 (3)
        call visitChildren(wn)
        call checkBoolean(wn.condition)
    end

    procedure visit(DoWhileLooping dwn)                              (4)
        call visitChildren(dwn)
        call checkBoolean(dwn.condition)
    end

    procedure visit(ForLooping fn)                                   (5)
        call openScope()
        call visitChildren(fn)
        if fn.condition ≠ null
        then call checkBoolean(fn.condition)
        call closeScope()
    end

    procedure visit(LabeledStmt ls)
        /★  Figure 9.11 on page 357                              ★/
    end
    procedure visit(Continuing cn)
        /★  Figure 9.12 on page 358                              ★/
    end
    procedure visit(Breaking bn)
        /★  Figure 9.15 on page 360                              ★/
    end
    procedure visit(Returning rn)
        /★  Figure 9.18 on page 362                              ★/
    end
end
```

Figure 9.1: Semantic Analysis Visitors (Part 1)

Other Semantic Analysis

- Reachability
  - …; return; a = a+1; ..
  - Adds a *isReachable* instance variable to AST
  - Warning issued if set to false
  - Also adds *terminatesNormally*
- Throws analysis
  - In Java exceptions are part of the type system
    - Checked/unchecked exceptions
      - modifiers return-type method-name (param-list) throws-clause

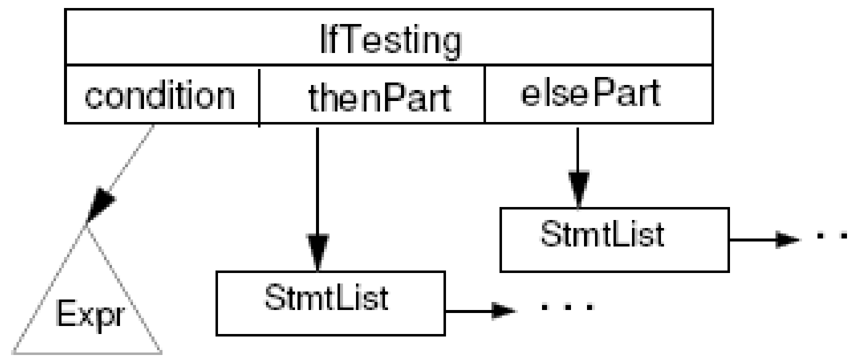


Figure 9.2: Abstract Syntax Tree for an If Statement

```
class ReachabilityVisitor extends NodeVisitor
    procedure visit( IfTesting ifn )                                    (6)
        ifn.thenPart.isReachable ← true
        ifn.elsePart.isReachable ← true
        call visitChildren( ifn )
        thenNormal ← ifn.thenPart.terminatesNormally
        elseNormal ← ifn.elsePart.terminatesNormally
        ifn.terminatesNormally ← thenNormal or elseNormal
    end

    procedure visit( WhileLooping wn )
        /★    Figure 9.6 on page 352                                 ★/
    end
    procedure visit( DoWhileLooping dwn )
        /★    Figure 9.7 on page 354                                 ★/
    end
    procedure visit( ForLooping fn )
        /★    Figure 9.8 on page 354                                 ★/
    end
    procedure visit( LabeledStmt ls )                                   (7)
        ls.stmt.isReachable ← ls.isReachable
        call visitChildren( ls )
        ls.terminatesNormally ← ls.stmt.terminatesNormally
    end

    procedure visit( Continuing cn )                                    (8)
        cn.terminatesNormally ← false
    end

    procedure visit( Breaking fn )
        /★    Figure 9.16 on page 360                                ★/
    end
    procedure visit( Returning rn )
        rn.terminatesNormally ← false
    end
end
```

Figure 9.3: Reachability Analysis Visitors (Part 1)

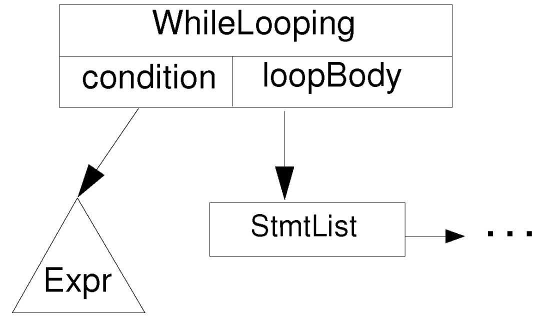


Figure 9.5: Abstract Syntax Tree for a While Statement

```
procedure VISIT( WhileLooping wn )
    wn.terminatesNormally ← true                              (21)
    wn.loopBody.isReachable ← true
    constExprVisitor ← new ConstExprVisitor( )
    call wn.condition.ACCEPT( constExprVisitor )
    conditionValue ← wn.condition.exprValue
    if conditionValue = true
    then
        wn.terminatesNormally ← false                         (22)
    else
        if conditionValue = false
        then
            wn.loopBody.isReachable ← false                   (23)
    call wn.loopBody.ACCEPT( this )                           (24)
end
```

Figure 9.6: Reachability Analysis for a While Statement

Semantic Checking Summary

- This phase of the compiler implements algorithms for checking the language scope and type rules
  - Define your scope and type rules
- If compiler is implemented in an OO language and use an AST choose between:
  - Traditional OO
  - (Traditional) Visitor
  - Reflective Visitor

What can you do in your project now?

- Start defining the type system for your language
  – Informal now
  – Formalize later
- Start implementing the type checker for your language

- Recommendation:
  – Start with simple types
  – Add composit and complex types later

Languages and Compilers (SProg og Oversættere)

Lecture 12
Types

Bent Thomsen
Department of Computer Science
Aalborg University

With acknowledgement to Simon Gay, Elsa Gunter and Elizabeth White whose slides this lecture is based on.

Learning goals

- Understand primitive and composit types
  - How implementations may affect types in languages
  - Pointer and references
  - Constructed datatypes:
    - Arrays
    - Records/structs
    - Unions or variant records
  - Structural and Name Equivalence
  - Recursive Types
  - E.g.: $\text{List} = \text{Unit} + (\text{Int} \times \text{List})$
  - Implicit versus explicit type conversions

- Understand some of the principles behind more advanced type systems
  - Polymorphism
  - Subtyping

Types revisited

- Fisher et al. and Sebesta, to some extent, may leave you with the impression that types in languages are simple and type checking is a minor part of the compiler
- However, type system design and type checking and/or inferencing algorithms is one of the hottest topics in programming language research at present!
- Types:
  - Have to be an integral part of the language design
    - Syntax
    - Contextual constraints (static type checking)
    - Code generation (space allocation and dynamic type checking)
  - Provides a precise criterion for safety and sanity of a design.
    - Language level
    - Program level
  - Close connections with logics and semantics.
    - The Curry–Howard correspondence

Typechecking

- Static typechecking
  - All type errors are detected at compile-time
  - Mini Triangle is *statically typed*
  - Most modern languages have a large emphasis on static typechecking
- Dynamic typechecking
  - Scripting languages such as JavaScript, PhP, Perl and Python do run-time typechecking
- Mix of Static and Dynamic
  - object-oriented programming requires some runtime typechecking: e.g. Java has a lot of compile-time typechecking but it is still necessary for some potential runtime type errors to be detected by the runtime system

- Static typechecking involves calculating or *inferring* the types of expressions (by using information about the types of their components) and checking that these types are what they should be (e.g. the condition in an *if* statement must have type *Boolean*).

Static Typechecking

- Static (compile-time) or dynamic (run-time)
  - *static is often desirable: finds errors sooner, doesn't degrade performance*
- Verifies that the programmer's intentions (expressed by declarations) are observed by the program
- A program which typechecks is guaranteed to behave well at run-time
  - *at least: never apply an operation to the wrong type of value more: eg. security properties*
- A program which typechecks respects the high-level abstractions
  - *eg: public/protected/private access in Java*

Why are Type declarations important?

- Organize data into high-level structures
  essential for high-level programming
- Document the program
  basic information about the meaning of variables and functions, procedures or methods
- Inform the compiler
  example: how much storage each value needs
- Specify simple aspects of the behaviour of functions
  "types as specifications" is an important idea

Why type systems are important

- Economy of execution
  - E.g. no null pointer checking is needed in SML
- Economy of small-scale development
  - A well-engineered type system can capture a large number of trivial programming errors thus eliminating a lot of debugging
- Economy of compiling
  - Type information can be organised into interfaces for program modules which therefore can be compiled separately
- Economy of large-scale development
  - Interfaces and modules have methodological advantages allowing separate teams to work on different parts of a large application without fear of code interference
- Economy of development and maintenance in security areas
  - If there is any way to cast an integer into a pointer type (or object type) the whole runtime system is compromised – most vira and worms use this method of attack
- Economy of language features
  - Typed constructs are naturally composed in an orthogonal way, thus type systems promote orthogonal programming language design and eliminate artificial restrictions

Why study type systems and programming languages?

The type system of a language has a strong effect on the “feel” of programming.

Examples:

- In original Pascal, the result type of a function cannot be an array type. In Java, an array is just an object and arrays can be used anywhere.
- In SML, programming with lists is very easy; in Java it is much less natural.

To understand a language fully, we need to understand its type system. The underlying typing concepts appearing in different languages in different ways, help us to compare and understand language features.

Java Example

Type definitions and declarations are essential aspects of high-level programming languages.

```
class Example {
      int a;
      void set(int x) {a=x;}
      int  get() {return a;}
}

Example e = new Example();
```

Where are the type definitions and declarations in the above code?

SML example

Type definitions and declarations are essential aspects of high-level programming languages.

```
datatype 'a tree =
   INTERNAL of {left:'a tree,right:'a tree}
 | LEAF of {contents:'a}

fun sum(tree: int tree) =
  case tree of
    INTERNAL{left,right} => sum(left) + sum(right)
   | LEAF{contents} => contents
```

Where are the type definitions and declarations in the above code?

Types

- Types are either primitive or constructed.
- Primitive types are atomic with no internal structure as far as the program is concerned
  - Integers, float, char, …
- Arrays, unions, structures, functions, … can be treated as constructor types
- Pointers (or references) and String are treated as basic types in some languages and as constructed types in other languages

Specification of Primitive Data Types

- Basic attributes of a primitive type usually used by the compiler and then discarded
- Some partial type information may occur in data object
- Values usually match with hardware types:
  - 8 bits, 16 bits, 32 bits, 64 bits
- Operations: primitive operations with hardware support, and user-defined/library operations built from primitive ones
- But there are design choices to be made!

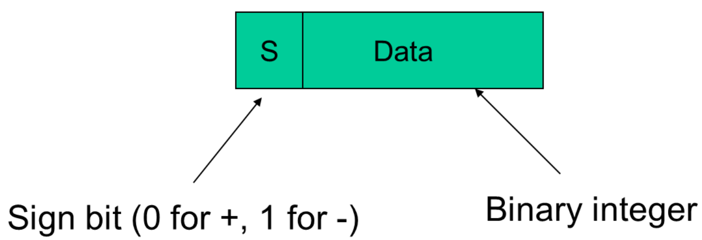
Integers – Specification

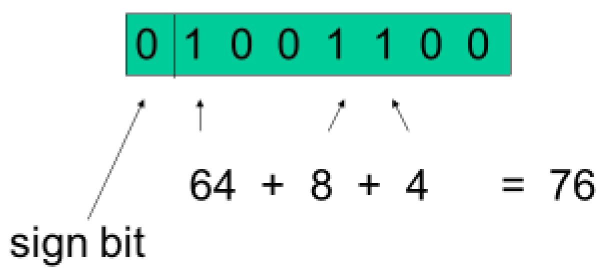
- The set of values of type *Integer* is a finite set
  - {-*maxint* … *maxint* }
  - typically -2^31 through 2^31 – 1
  - –2^30 through 2^30 - 1
  - not the mathematical set of integers (as operations may overflow).
- Standard collection of operators:
  - +, -, *, /, mod, ~ (negation)
- Standard relational operators:
  - =, <, >, <=, >=, =/=
- The language designer has to decide
  - which representation to use
  - The collection of operators and relations

Integers - Implementation

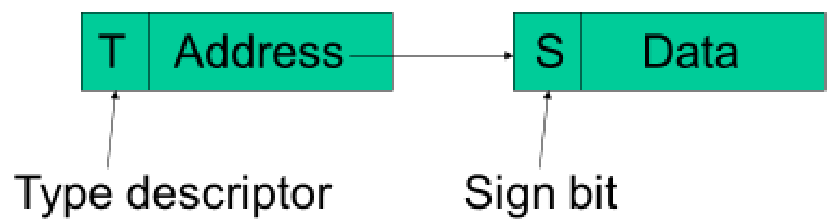
- Implementation:
  - Binary representation in 2's complement arithmetic
  - Three different standard representations:

- First kind:

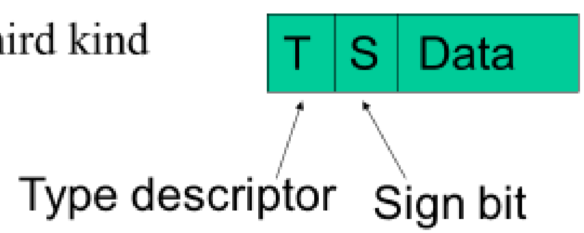




- Second kind



- Third kind



Floating Points

- IEEE standard 754 specifies both a 32- and 64-bit standard
- At least one supported by most hardware
- Some hardware also has proprietary representations
- Numbers consist of three fields:
  - S (sign), E (exponent), M (mantissa)

S	E	M

- Every non-zero number may be uniquely written as

$$(-1)^S * 2^E * M$$

where $1 \leq M < 2$ and S is either 0 or 1

Language design issue

- Should my language support floating points?
- Should it support IEEE standard 754
  - 32 bit, 64 bits or both
- Should my language support native floating points?
- Should floating points be the only number representation in my language?

Other Primitive Data

- Short integers (C) - 16 bit, 8 bit
- Long integers (C) - 64 bit
- Boolean or logical - 1 bit with value true or false (often stored as bytes)
- Byte - 8 bits
- Java has
  - byte, short, int, long, float, double, char, boolean
- C# also has
  - sbyte, ushort, uint, ulong

Characters

- Character - Single 8-bit byte - 256 characters
- ASCII is a 7 bit 128 character code
- Unicode is a 16-bit character code (Java)
- In C, a char variable is simply 8-bit integer numeric data

Enumerations

- Motivation: Type for case analysis over a small number of symbolic values
- Example: (Ada)

  Type DAYS **is** {Mon, Tues, Wed, Thu, Fri, Sat, Sun}
- Implementation: Mon → 0; … Sun → 6
- Treated as ordered type (Mon < Wed)
- In C, always implicitly coerced to integers
- Java didn't have enum until Java 1.5

Java Type-safe enum

Remember

```
public class Token {
   byte kind; String spelling;
   final static byte
      IDENTIFIER = 0; INTLITERAL = 1; OPERATOR   = 2;
      BEGIN      = 3; CONST      = 4; ...
      ...
...
}
```

```
private void parseSingleCommand() {
   switch (currentToken.kind) {
     case Token.IDENTIFIER : ...
     case Token.IF : ...
     ... more cases ...
     default: report a syntax error
   }
}
```

Java Type-safe enum

Can now be written as

```
public class Token {
   String spelling;
   enum kind {IDENTIFIER, INTLITERAL, OPERATOR,
      BEGIN, CONST, ... }
      ...
...
}
```

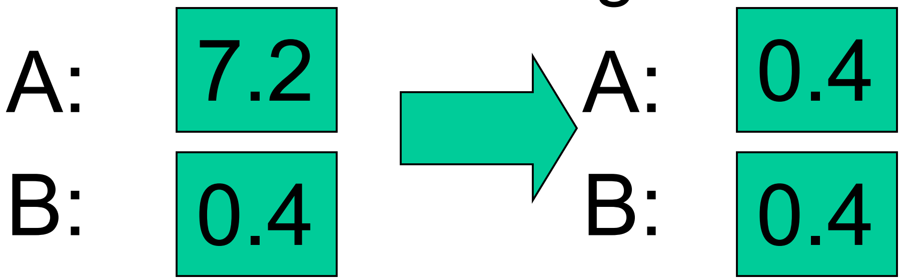
```
private void parseSingleCommand() {
   switch (currentToken.kind) {
     case IDENTIFIER : ...
     case IF : ...
     ... more cases ...
     default: report a syntax error
   }
}
```

Pointers

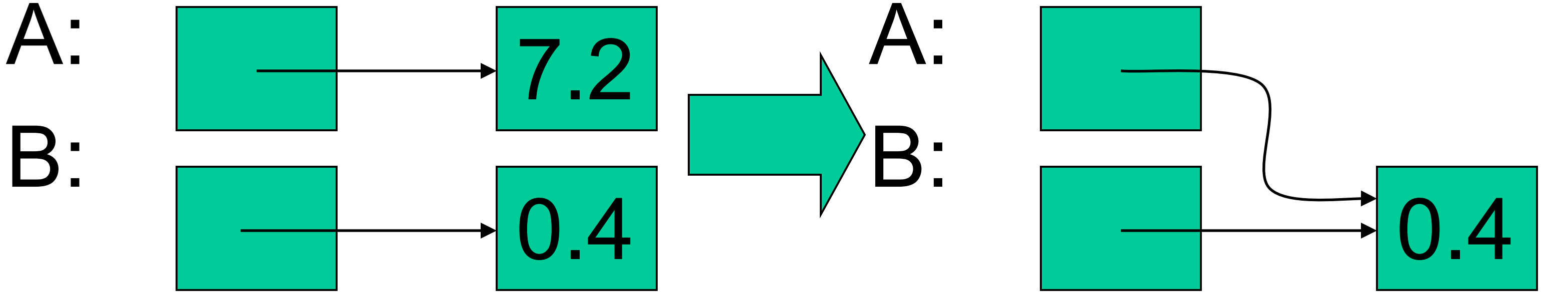
- A *pointer type* is a type in which the range of values consists of memory addresses and a special value, nil (or null)
- Each pointer can point to an object of another data structure
  - Its l-value is its address; its r-value is the address of another object
- Accessing r-value of r-value of pointer called *dereferencing*
- Use of pointers to create arbitrary data structures

Pointer Aliasing

- A:= B
  - Numeric assignment

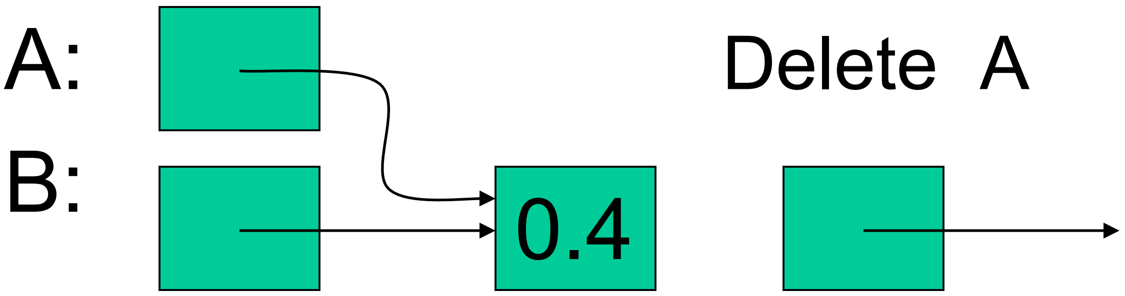


  - Pointer assignment

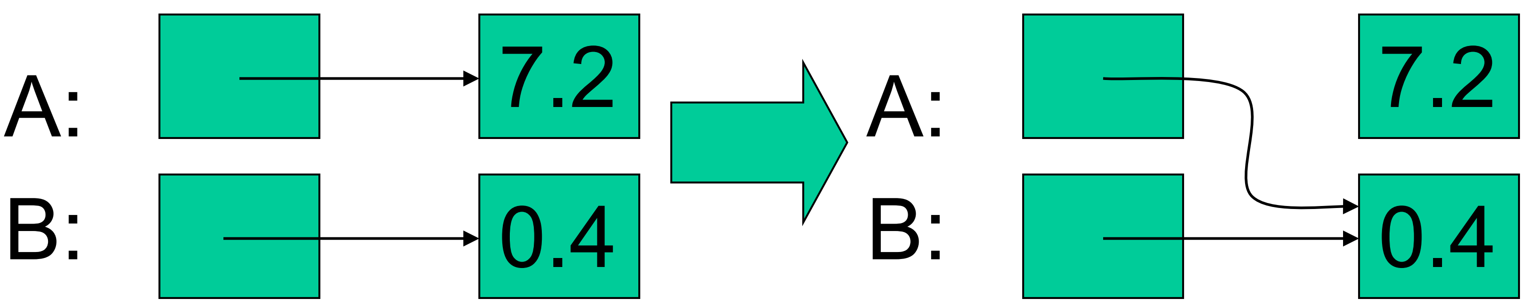


Problems with Pointers

- Dangling Pointer



- Garbage (lost heap-dynamic variables)



SML references

- An alternative to allowing pointers directly
- References in SML can be typed
- … but they introduce some abnormalities

- SML reference cells
  - Different types for location and contents

    x : int   non-assignable integer value
    y : int ref   location whose contents must be integer
    !y   the contents of location y
    ref x   expression creating new cell initialized to x
  - SML assignment

    operator := applied to memory cell and new contents
  - Examples

    y := x+3   place value of x+3 in cell y; requires x:int
    y := !y + 3 add 3 to contents of y and store in location y
-

References in Java and C#

- Similar to SML both Java and C# use references to heap allocated objects

```
class Point {
  int x,y;
  public Point(int x, int y) {
     this.x=x; this.y=y;
  }

  public void move(int dx, int dy) {
     x=x+dx; y=y+dy;
  }
}
…
Point p = new Point(2,3);
p.move(5,6);
Point q = new Point(0,0);
p = q;
p.move(3,7);
q = null;
```

Nullable Types in C#

- T? same as System.Nullable<T>

```
int? x = 123;
double? y = 1.25;
```

- null literal conversions

```
int? x = null;
double? y = null;
```

- Nullable conversions

```
int i = 123;
int? x = i;             // int --> int?
double? y = x;          // int? --> double?
int? z = (int?)y;       // double? --> int?
int j = (int)z;         // int? --> int
```

Strings

- Can be implemented as
  - a primitive type as in SML
  - an object as in Java
  - an array of characters (as in C and C++)
- If primitive, operations are built in
- If object or array of characters, string operations provided through a library

- String implementations:
  - Fixed declared length
  - Variable length with declared maximum
  - Unbounded length
    - Linked list of fixed length strings
    - null terminated contiguous array

Arrays

An array is a collection of values, all of the same type, indexed by a range of integers (or sometimes a range within an enumerated type).

In Ada: a : array (1..50) of Float; (static arrays)
In Java: float[] a; (dynamic arrays)

Most languages check at runtime that array indices are within the bounds of the array: a(51) is an error. (In C you get the contents of the memory location just after the end of the array!)

If the bounds of an array are viewed as part of its type, then array bounds checking can be viewed as typechecking, but in general it is impossible to do it statically: consider a(f(1)) for an arbitrary function f.

Static typechecking is a compromise between *expressiveness* and *computational feasibility*. More about this later

Array Layout and Component Access

- Component access through subscripting, both for lookup (r-value) and for update (l-value)

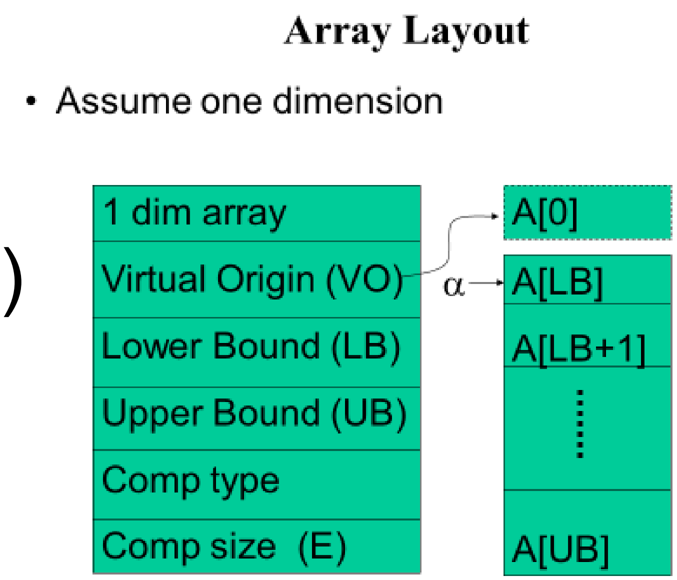
- Component access should take constant time (ie. looking up the 5th element takes same time as looking up 100th element)

- L-value of A[i] = VO + (E * i)

  $= \alpha + (E * (i – LB))$

- Computed at compile time
- VO = α - (E * LB)

- More complicated for multiple dimensions



Pause

Composite Data Types

- Composite data types are sets of data objects built from data objects of other types
- Data type constructors are arrays, structures, unions, lists, …
- It is useful to consider the structure of types and type constructors independently of the form which they take in particular languages.

Products and Records

If T and U are types, then $T \times U$ (written (T * U) in SML) is the type whose values are pairs (t,u) where t has type T and u has type U.

Mathematically this corresponds to the *cartesian product* of sets. More generally we have *tuple* types with any number of components. The components can be extracted by means of *projection functions*.

Product types more often appear as *record types*, which attach a label or *field name* to each component. Example in Ada and C:

```
type T is
record
   x : Integer;
   y : Float
end record
```

```
struct T {
   int x;
   float y;
}
```

Products and Records

If *v* is a value of type *T* then *v* contains an Integer and a Float. Writing *v.x* and *v.y* can be more readable than *fst(v)* and *snd(v)*.

```
type T is
record
   x : Integer;
   y : Float
end record
```

Record types are mathematically equivalent to products.

An object can be thought of as a record in which some fields are functions, and a class definition as a record type definition in which some fields have function types. Object-oriented languages also provide *inheritance*, leading to *subtyping* relationships between object types.

Variant Records

In Pascal, the value of one field of a record can determine the presence or absence of other fields. Example:

```
type T = record
            x : integer;
            case b : boolean of
              false : (y : integer);
              true : (z : boolean)
         end
```

It is not possible for static type checking to eliminate all type errors from programs which use variant records in Pascal: the compiler cannot check consistency between the *tag field* and the data which is stored in the record. The following code passes the type checker in Pascal:

```
var r : T, a : integer;
begin
  r.x := 1; r.b := true; r.z := false;
  a := r.y * 5
end
```

Variant Records in Ada

Ada handles variant records safely. Instead of a tag field, the type definition has a parameter, which is set when a particular record is created and then cannot be changed.

```
type T(b : Boolean) is record
   x : Integer;
   case b is
      when False => y : Integer;
      when True  => z : Boolean
   end case
end record;

declare r : T(True), a : Integer;
begin
   r.x := 1; r.z := False;
   a := r.y * 5;
end;
```

r does not have field y, and never will

this type error can be detected statically

Disjoint Unions

The mathematical concept underlying variant record types is the *disjoint union*. A value of type $T+U$ is either a value of type T or a value of type U, tagged to indicate which type it belongs to:

$$T+U = \{ \, left(x) \mid x \in T \, \} \cup \{ \, right(x) \mid x \in U \, \}$$

SML and other functional languages support disjoint unions by means of *algebraic datatypes*, e.g.

```
datatype X = Alpha String | Numeric Int
```

The *constructors* Alpha and Numeric can be used as functions to build values of type X, and pattern-matching can be used on a value of type X to extract a String or an Int as appropriate.

An enumerated type is a disjoint union of copies of the *unit* type (which has just one value). Algebraic datatypes unify enumerations and disjoint unions (and recursive types) into a convenient programming feature.

Variant Records and Disjoint Unions

The Ada type:

```
type T(b : Boolean) is record
   x : Integer;
   case b is
      when False => y : Integer;
      when True  => z : Boolean
   end case
end record;
```

can be interpreted as

$$(Integer \times Integer) + (Integer \times Boolean)$$

where the Boolean parameter b plays the role of the *left* or *right* tag.

Note C also has union types
but they are unsafe as no check is performed on field selection

Functions

In a language which allows functions to be treated as values, we need to be able to describe the type of a function, independently of its definition.

In Ada, defining function f(x : Float) return Integer is …

produces a function f whose type is

function (x : Float) return Integer

the name of the parameter is insignificant (it is a *bound name*) so this is the same type as function (y : Float) return Integer

In SML this type is written Float → Int

In Scala this type is written Float => Int

Functions and Procedures

A function with several parameters can be viewed as a function with one parameter which has a product type:

function (x : Float, y : Integer) return Integer

$$\text{Float} \times \text{Int} \rightarrow \text{Int}$$

In Ada, procedure types are different from function types:

procedure (x : Float, y : Integer)

whereas in Java a procedure is simply a function whose result type is *void*. In SML, a function with no interesting result could be given a type such as $\text{Int} \rightarrow (\,)$ where ( ) is the empty product type (also known as the *unit* type) although in a purely functional language there is no point in defining such a function.

Structural and Name Equivalence

At various points during type checking, it is necessary to check that two types are the same. What does this mean?

structural equivalence: two types are the same if they have the same structure: e.g. arrays of the same size and type, records with the same fields.

name equivalence: two types are the same if they have the same name.

Example: if we define

```
type A = array 1..10 of Integer;
type B = array 1..10 of Integer;
function f(x : A) return Integer is …
var b : B;
```

then f(b) is correct in a language which uses structural equivalence, but incorrect in a language which uses name equivalence.

Structural and Name Equivalence

Different languages take different approaches, and some use both kinds.

Ada uses name equivalence.
Triangle uses structural equivalence.
Haskell uses structural equivalence for types defined by *type* (these are viewed as new names for existing types) and name equivalence for types defined by *data* (these are algebraic datatypes; they are genuinely new types).

Structural equivalence is sometimes convenient for programming, but does not protect the programmer against incorrect use of values whose types accidentally have the same structure but are logically distinct.

Name equivalence is easier to implement in general, especially in a language with recursive types.

Recursive Types

Example: a list is either empty, or consists of a value (the *head*) and a list (the *tail*)

SML:

```
datatype List = Nil
              | Cons (Int * List)
```

```
Cons 2 (Cons 3 (Cons 4 Nil))
```

represents [2,3,4]

Abstractly: $List = Unit + (Int \times List)$

In SML, the implementation uses pointers, but the programmer does not have to think in terms of pointers.

Recursive Types

Java:

```
class List {
   int head;
   List tail;
}
```

The Java definition does not mention pointers,
but we use the explicit null pointer **null** to represent the empty list.

Equivalence of Recursive Types

In the presence of recursive types, defining structural equivalence is more difficult.

We expect $List = Unit + (Int \times List)$

and $NewList = Unit + (Int \times NewList)$

to be equivalent, but complications arise from the (reasonable) requirement that $List = Unit + (Int \times List)$

and $NewList = Unit + (Int \times (Unit + (Int \times NewList)))$

should be equivalent.

It is usual for languages to avoid this issue by using name equivalence for recursive types, but recent research on co-inductive types show it is Possible and (sometimes) useful to have structural equivalence on recursive types

Other Practical Type System Issues

- Implicit versus explicit type conversions
  - Explicit ➔ user indicates (Ada, SML)
  - Implicit ➔ built-in (C int/char) -- coercions
- Overloading – meaning based on context
  - Built-in
  - Extracting meaning – parameters/context
- Polymorphism
- Subtyping

Coercions Versus Conversions

- When A has type **real** and B has type **int**, many languages allow coercion implicit in

  A := B

- In the other direction, often no coercion allowed; must use explicit conversion:
  - B := round(A); Go to integer nearest B
  - B := trunc(A); Delete fractional part of B

Explicit vs. Implicit conversion Autoboxing/Unboxing

- In Java 1.4 you had to write:

```
Integer x = Integer.valueOf(6);
Integer y = Integer.valueOf(2 * x.IntValue);
```

- In Java 1.5 you can write:

```
Integer x = 6;              //6 is boxed
Integer y = 2*x + 3;        //x is unboxed, 15 is boxed
```

  - Autoboxing wrap ints into Integers
  - Unboxing extract ints from Integers

Explicit vs. Implicit conversion Autoboxing/Unboxing

- Extending a language can imply difficult design compromises. In Java 1.5 we can write:

- Integer x = 3; (an integer object)
- int y = 3; (an integer)
- Integer z = 3; (an integer)
- .. x==y .. (true due to auto unboxing)
- .. y == z .. (true due to auto unboxing)
- .. x == z .. (false due to object comparisson)

- I.e. the convenience of autoboxing/unboxing leads to the == operator no longer being transitive
- Note: Not a problem in C# as autoboxing/unboxing is handled by the run-time system.

Polymorphism

Polymorphism describes the situation in which a particular operator or function can be applied to values of several different types. There is a fundamental distinction between:

- *ad hoc polymorphism*, usually called *overloading*, in which a single name refers to a number of unrelated operations.
    - Examples: + and static overloading of methods
- *bounded or Subtype polymorphism (inheritance polymorphism)*
- *parametric polymorphism (generics)*, in which the same computation can be applied to a range of different types which have structural similarities.

Most languages have some support for overloading.

Parametric polymorphism is familiar from functional programming, but less common (or less well developed) in imperative languages. Generics (or Parametric Polymorphism) has recently had a lot of attention in OO languages.

Parametric polymorphism (generics)

```
datatype ’a tree =
   INTERNAL of {left:’a tree,right:’a tree}
 | LEAF of {contents:’a}

fun tw(tree: ‘a tree, comb: ‘a*‘a->’a) =
  case tree of
    INTERNAL{left,right} => comb(tw(left),tw(right))
  | LEAF{contents} => contents
```

Parametric polymorphism (generics)

```
public class List<ItemType>
{
   private ItemType[] elements;
   private int count;

   public void Add(ItemType element) {
      if (count == elements.Length) Resize(count * 2);
      elements[count++] = element;
   }

   public ItemType this[int index] {
      get { return elements[index]; }
      set { elements[index] = value; }
   }

   publ
      g
   }
}
```

```
List<int> intList = new List<int>();

intList.Add(1);              // No boxing
intList.Add(2);              // No boxing
intList.Add("Three");        // Compile-time error

int i = intList[0];          // No cast required
```

Implementing generic types

- Type erasure, e.g:
  - <T extends Addable> T add(T a, T b) { … }
  - can be compiled, type-checked, and called the same way as:
  - Addable add(Addable a, Addable b) { … }

- Template:
- Apply the template to the provided template arguments. E.g calling template
  - <class T> T add(T a, T b) { … }
  - as add<int>(1, 2)
  - actual function int __add__T_int(int a, int b)

The Hindley-Milner Type inference Algorithm

Algorithm $\mathscr{W}$

$$\mathscr{W}(\bar{p}, f) = (T, \bar{f}), \text{ where}$$

(i) If f is x, then:

if λx_σ or *fix* x_σ is active in $\bar{p}$ then
$T = I, \bar{f} = x_\sigma$;
if *let* x_σ is active in $\bar{p}$ then
$T = I, \bar{f} = x_\tau$
where $\tau = [\beta_i/\alpha_i]\sigma$, α_i are the generic variables of σ,
and β_i are new variables.

(ii) If f is (de), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$;
let $U = \mathscr{U}(S\rho, \sigma \to \beta)$, β new;
then $T = USR$, and $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

(iii) If f is (*if d then e else e'*), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$ and $U_0 = \mathscr{U}(\rho, \iota_0)$;
let $(S, \bar{e}_\sigma) = \mathscr{W}(U_0 R\bar{p}, e)$, and $(S', \bar{e}'_{\sigma'}) = \mathscr{W}(SU_0R\bar{p}, e')$;
let $U = \mathscr{U}(S'\sigma, \sigma')$;
then $T = US'SU_0R$, and
$\bar{f} = U((\textit{if } S'SU_0\bar{d} \textit{ then } S'\bar{e} \textit{ else } \bar{e}')_\sigma)$.

(iv) If f is $(\lambda x \cdot d)$, then:

let $(R, \bar{d}) = \mathscr{W}(\bar{p} \cdot \lambda x_\beta, d)$, where β is new;
then $T = R$, and $\bar{f} = (\lambda x_{R\beta} \cdot \bar{d}_\rho)_{R\beta \to \rho}$.

(v) If f is (*fix* $x \cdot d$), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p} \cdot \textit{fix } x_\beta, d)$, β new;
let $U = \mathscr{U}(R\beta, \rho)$;
then $T = UR$, and $\bar{f} = (\textit{fix } x_{UR\beta} \cdot U\bar{d})_{UR\beta}$.

(vi) If f is (*let* $x = d$ *in* e), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$;
let $(S, e_\sigma) = \mathscr{W}(R\bar{p} \cdot \textit{let } x_\rho, e)$;
then $T = SR$, and $\bar{f} = (\textit{let } x_{S\rho} = S\bar{d} \textit{ in } \bar{e})_\sigma$. ∎

- First used in SML
- A Theory of Type Polymorphism in Programming
  - Robin Milner (1977)
- Algoritmn basically builds and solves equations over type expressions
- Now in use in:
  - Haskell, C#, F#, Visual Basic .Net 9.0

Subtyping

The interpretation of a type as a set of values, and the fact that one set may be a subset of another set, make it natural to think about when a value of one type may be considered to be a value of another type.

Example: the set of integers is a subset of the set of real numbers. Correspondingly, we might like to consider the type Integer to be a *subtype* of the type Float. This is often written Integer <: Float.

The subtype relation enjoys the following properties:
X <: X (indempotent)
X<:Y and Y<:Z then X<:Z (transitivity)

Different languages provide subtyping in different ways, including (in some cases) not at all. In object-oriented languages, subtyping arises from inheritance between classes.

Subtyping and Polymorphism

```
abstract class Shape {
  abstract float area( ); }
```

the idea is to define several classes of Shape, all of which define the area function

```
class Square extends Shape {
  float side;
  float area( ) {return (side * side); } }
```

Square <: Shape

```
class Circle extends Shape {
  float radius;
  float area( ) {return ( PI * radius * radius); } }
```

Circle <: Shape

Objects can be thought of as (extendible) records of fields and methods.
That is why Square <: Shape and Circle <: Shape

Subtyping and Polymorphism

```
float totalarea(Shape[] s)  {
  float t = 0.0;
  for (int i = 0; i < s.length; i++) {
    t = t + s[i].area( ); };
  return t;
}
```

totalarea can be applied to any array whose elements are subtypes of Shape. (This is why we want Square[] <: Shape[] etc.)

This is an example of a concept called *bounded polymorphism*.

Subtyping for Product Types

The rule is:

if $A <: T$ and $B <: U$ then $A \times B <: T \times U$

This rule, and corresponding rules for other structured types, can be worked out by following the principle:

$T <: U$ means that whenever a value of type U is expected, it is safe to use a value of type T instead.

What can we do with a value *v* of type $T \times U$?

• use *fst(v)* , which is a value of type T

• use *snd(v)* , which is a value of type U

If *w* is a value of type $A \times B$ then *fst(w)* has type A and can be used instead of *fst(v)*. Similarly *snd(w)* can be used instead of *snd(v)*. Therefore *w* can be used where *v* is expected.

Subtyping for Function Types

Suppose we have $f : A \to B$ and $g : T \to U$ and we want to use f in place of g.

It must be possible for the result of f to be used in place of the result of g, so we must have B <: U.

It must be possible for a value which could be a parameter of g to be given as a parameter to f, so we must have T <: A.

Therefore: if T <: A and B <: U then A → B <: T → U

Compare this with the rule for product types, and notice the *contravariance*: the condition on subtyping between A and T is the other way around.

Correctness of Type Systems

How does a language designer (or a programmer) know that correctly-typed programs really have the desired run-time properties?

To answer this question we need to see how to specify type systems, and how to prove that a type system is *sound*.

To do this we can use techniques similar to those from SOS

To prove soundness we also need to specify the *semantics* (meaning) of programs - what happens when they are run.

So studying types will lead us to a deeper understanding of the meaning of programs.

Connection with Semantics

- Type system is part of the static semantics
  - Static semantics: the well-formed programs
  - Dynamic semantics: the execution model
- Safety theorem: types predict behaviour.
  - Types describe the states of an abstract machine model.
  - Execution behaviour must cohere with these descriptions.
  - **Theorem:** If Γ |- E:τ and E$\rightarrow$ E’ then Γ |- E’:τ
  - See Theorem 13.9 p. 196 in Transitions and Trees
- Thus a type is a specification and a type checker is a theorem prover.
- Type checking is the most successful formal method!
  - In principle there are limits.
  - In practice there is no end in sight.
- Examples:
  - Using types for low-level languages, say inside a compiler.
  - Extending the expressiveness of type systems for high-level languages.

Summary

- Static typing is important
- Type system has to be an integral part of the language design
- There are a lot of nitty-gritty decisions about primitive data types
- Composite types are best understood independently of language manifestation to ensure correctness of implementation
- Type systems can (and should) be formalised

Languages and Compilers (SProg og Oversættere)

Lecture 13 Programming Language Design Expressions and Statements

Bent Thomsen

Department of Computer Science

Aalborg University

With acknowledgement to Simon Gay, John Mitchell and Elsa Gunter who’s slides this lecture is based on.

Learning goals

- Overview of common language constructs and design questions
- Understand
  - Explicit sequence control vs. Implicit sequence control
    - Evaluation of expressions
    - Statements
  - Structured sequence control vs. unstructured sequence control
    - Conditional Selection
    - Loop constructs
    - Jumps

Syntactic Elements

- Declarations and Definitions
  - Scopes and visibility
  - always before use or not, initialization or not,
- **Expressions**
- **Statements**
- Subprograms

- Separate subprogram definitions (Module system)
- Separate data definitions
- Nested subprogram definitions
- Separate interface definitions

Sequence control

- Implicit and explicit sequence control
  - Expressions
    - Precedence rules
    - Associativity
  - Statements
    - Sequence
    - Conditionals
    - Loop constructs
    - unstructured vs. structured sequence control

Expression Evaluation

- Determined by
  - operator evaluation order
  - operand evaluation order
- Operators:
  - Most operators are either infix or prefix (some languages have postfix)
  - Order of evaluation determined by operator precedence and associativity

Example

- What is the result of:

  `3 + 4 * 5 + 6`

- Possible answers:
  - `41 = ((3 + 4) * 5) + 6`
  - `47 = 3 + (4 * (5 + 6))`
  - `29 = (3 + (4 * 5)) + 6 = 3 + ((4 * 5) + 6)`
  - `77 = (3 + 4) * (5 + 6)`

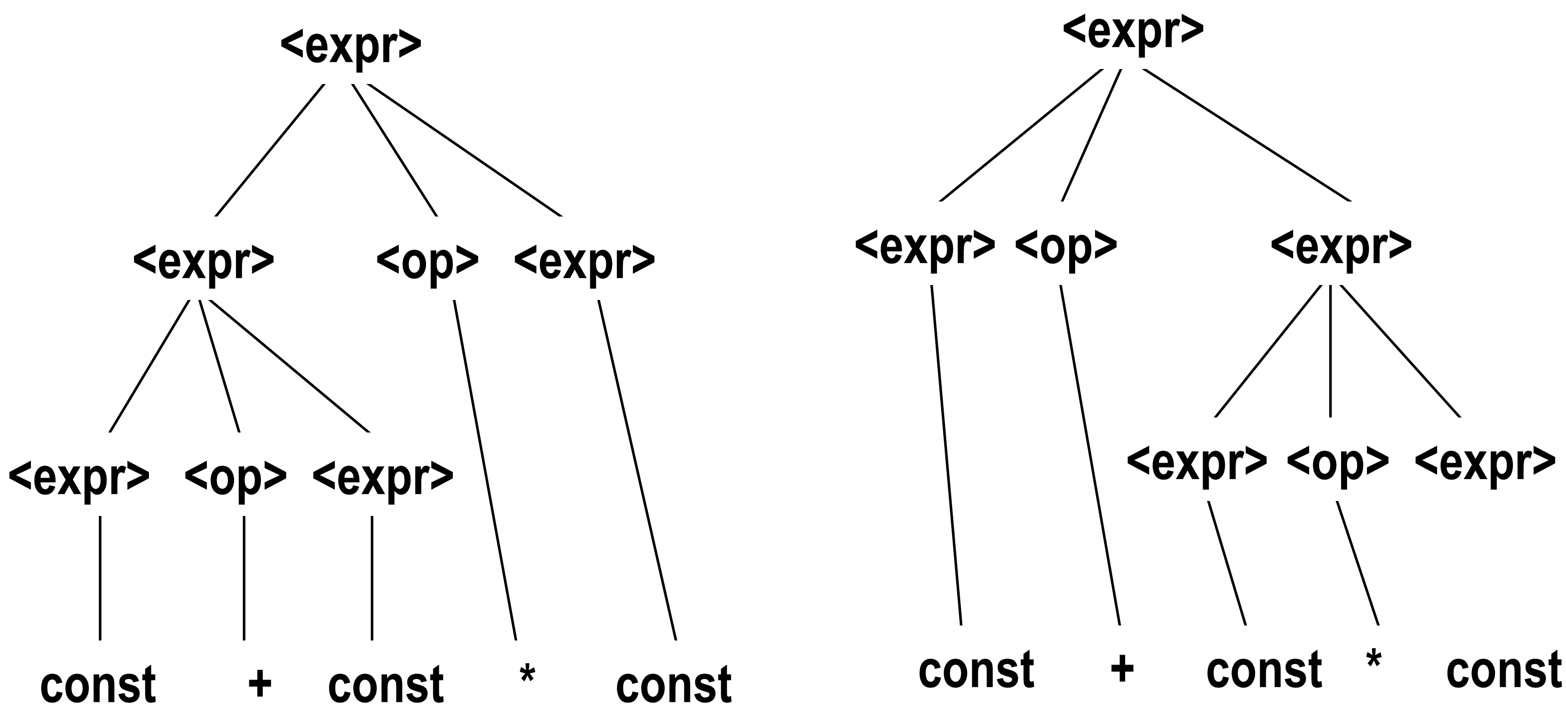
- In most languages, **`3 + 4 * 5 + 6 = 29`**
- … but it depends on the precedence of operators

An Ambiguous Expression Grammar

How to parse 3+4*5?

<expr> → <expr> <op> <expr> | const

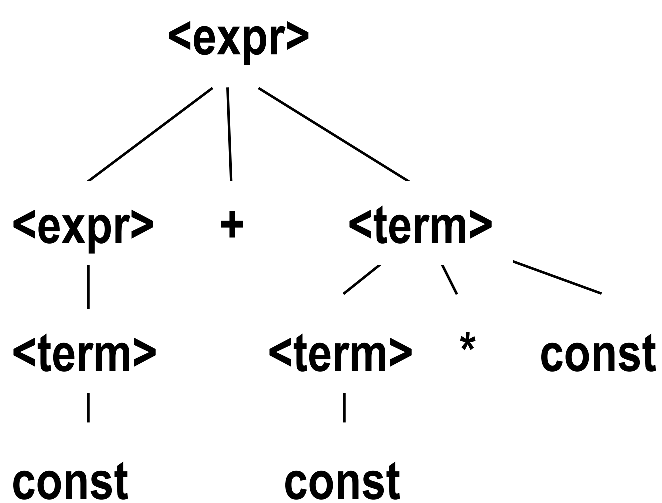
<op> → + | *



Expressing Precedence in grammar

- We can use the parse tree to indicate precedence levels of the operators

<expr> → <expr> + <term> | <term>
<term> → <term> * `const` | `const`



In LALR parsers we can specify
Precedence which translates into
Solving shift-reduce conflicts

Note in LL(1) parsers we have to use
Left recursion elimination

Expr → Term Expr1 .
Expr1 →+ Term Expr1
| .
Term → const Term1 .
Term1 →* const Term1
| .

Operator Precedence

- Operators of highest precedence evaluated first (bind more tightly).

- Precedence for operators usually given in a table, e.g.:

- In APL, all infix operators have same precedence

Level	Operator	Operation
Highest	** abs not	Exp, abs, negation
	* / mod rem	
	+ -	Unary
	+ - &	Binary
	= <= < > =>	Relations
Lowest	And or xor	Boolean

Precedence table for ADA

C precedence levels

Precedence	Operators	Operator names
17	tokens, a[k], f()	Literals, subscripting, function call
	.,->	Selection
16	++, --	Postfix increment/decrement
15*	++, --	Prefix inc/dec
	~, -, sizeof	Unary operators, storage
	!,&,*	Logical negation, indirection
14	typename	Casts
13	*, /, %	Multiplicative operators
12	+,-	Additive operators
11	<<, >>	Shift
10	<,>,<=, >=	Relational
9	==, !=	Equality
8	&	Bitwise and
7	^	Bitwise xor
6	|	Bitwise or
5	&&	Logical and
4	||	Logical or
3	?:	Conditional
2	=, +=, -=, *=,	Assignment
	/=, %=, <<=, >>=,	
	&=, ^=, |=	
1	,	Sequential evaluation

Associativity

- When we have sorted precedence we need to sort associativity!
- What is the value of:

```
7 - 5 - 2
```

- Possible answers:
  - In Pascal, C++, SML associate to the left

```
7 - 5 - 2 = (7 - 5) - 2 = 0
```

  - In APL, associate to the right

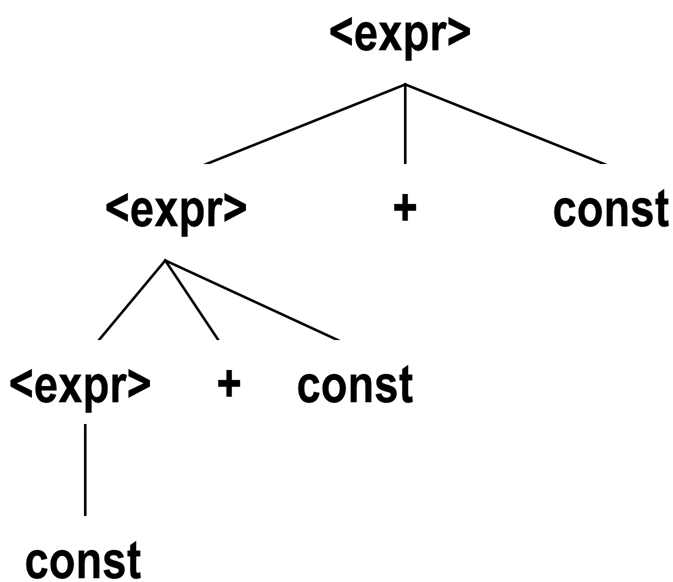
```
7 - 5 - 2 = 7 - (5 - 2) = 4
```

Again we can use syntax

- Operator associativity can also be indicated by a grammar

<expr> -> <expr> + <expr> | const (ambiguous)

<expr> -> <expr> + const | const (unambiguous)



In LALR parsers we can specify
Associativity which translates into
Solving shift-reduce conflicts

Operand Evaluation Order

- Example:

```
A := 5;
f(x) = {A := x+x; return x};
B := A + f(A);
```

- What is the value of B?
- 10 or 15?

Example

- If assignment returns the assigned value, what is the result of

```
x = 5;
y = (x = 3) + x;
```

- Possible answers: 6 or 8
- Depends on language, and sometimes compiler
  - C allows compiler to decide
  - SML forces left-to-right evaluation
    - Note assignment in SML returns a unit value
    - .. but we could define a derived assignment operator in SML as fn (x,v)=>(x:=v;v)

Solution to Operand Evaluation Order

- Disallow all side-effects
  - "Purely" functional languages try to do this – Miranda, Haskell
  - It works!
  - Consequence
    - No two-way parameters in functions
    - No non-local references in functions
  - Problem:
    - I/O, error conditions such as overflow are inherently side-effecting
    - Programmers want the flexibility of two-way parameters (what about C?) and non-local references

Solution to Operand Evaluation Order

- Disallow all side-effects in expressions but allow in statements
  - Problem: not applicable in languages with nesting of expressions and statements

Solution to Operand Evaluation Order

- Fix order of evaluation
  - SML does this – left to right
  - Problem: makes some compiler optimizations hard or impossible
- Leave it to the programmer to be sure the order doesn't matter
  - Problem: Usually requires lots of brackets
  - Problem: error prone

  - Fortress: Parallel evaluation unless specified to be sequential

Short-circuit Evaluation

- Boolean expressions:
- Example: **`x <> 0 andalso y/x > 1`**
- Problem: if **`andalso`** is ordinary operator and both arguments must be evaluated, then **`y/x`** will raise an error when **`x = 0`**

- Similar problem for conditional expressions
- Example **`(x == 0)?0:sum/x`**

Boolean Expressions

- Most languages allow (some version of) **`if`**...**`then`**...**`else, andalso, orelse`** not to evaluate all the arguments
- **`if true then A else B`**
  - doesn't evaluate B
- **`if false then A else B`**
  - doesn't evaluate A
- **`if b_exp then A else B`**
  - Evaluates b_exp, then applies previous rules

Boolen Expressions

- **`Bexp1 andalso Bexp2`**
  - If Bexp1 evaluates to false, doesn’t evaluate Bexp2
- **`Bexp1 orelse Bexp2`**
  - If Bexp1 evaluates to true, doesn’t evaluate Bexp2

Short-circuit Evaluation – Other Expressions

- Example: `0 * A = 0`
- Do we need to evaluate A?

- In general, in **f(x,y,…,z)** are the arguments to **f** evaluated before **f** is called and the values are passed? Or are the unevaluated expressions passed as arguments to **f** allowing **f** to decide which arguments to evaluate and in which order?

Eager Evaluation

- If a language requires all arguments to be evaluated before a function is called, the language does ***eager evaluation*** and the arguments are passed using pass by value (also called ***call by value***) or pass by reference

Lazy Evaluation

- If a language allows a function to determine which arguments to evaluate and in which order, the language does ***lazy evaluation*** and the arguments are passed using pass by name (also called ***call by name***)

Lazy Evaluation

- Lazy evaluation is mainly done in purely functional languages
- Some languages support a mix
- The effect of lazy evaluation can be implemented in functional languages with eager evaluation
  - Use thunking **`fn()=>exp`** and pass function instead of exp
- C# 2.0 has a Lazy evaluation construct:
  - **`yield return`** which can be used with Iterators

Call by name

- In call-by-name evaluation, the arguments to a function are not evaluated before the function is called — rather, they are substituted directly into the function body (using capture-avoiding substitution) and then left to be evaluated whenever they appear in the function.
- If an argument is not used in the function body, the argument is never evaluated
- If it is used several times, it is re-evaluated each time it appears
  - (in Pure lazy functional languages memorization can be used – why?)
- Algol 60 introduced call-by-name.
- Long consider too expensive and weird
  - but now in Scala
  - Can be simulated in C# using Expression<T> parameters
- The classical use case for call-by-name is Jensens device

Arithmetic Expressions

- Design issues for arithmetic expressions:
  1. What are the operator precedence rules?
  2. What are the operator associativity rules?
  3. What is the order of operand evaluation?
  4. Are there restrictions on operand evaluation side effects?
  5. Does the language allow user-defined operator overloading?
     - C++, Ada, C# allow user defined overloading
     - Can lead to readability problems
  6. What mode mixing is allowed in expressions?
     - Are operators of different types, e.g. int and float allowed
     - How is type conversion done

Pause

Syntactic Elements

- Definitions
- Declarations
- Expressions
- **Statements**
- Subprograms

- Separate subprogram definitions (Module system)
- Separate data definitions
- Nested subprogram definitions
- Separate interface definitions

Control of Statement Execution

- Sequential
- Conditional Selection
- Looping Construct
- Must have all three to provide full power of a Computing Machine

Basic sequential operations

- Skip (in C it is just a blanck statement with ;)
- Assignments
  - Most languages treat assignment as a basic operation
  - Some languages have derived assignment operators such as:
    - **`+=`** and **`*=`** in C
- I/O
  - Some languages treat I/O as basic operations
  - Others like, C, SML, Java treat I/O as functions/methods
- Sequencing
  - **`C;C`**
- Blocks
  - **`begin …end`**
  - **`{…}`**
  - **`let .. in .. end`**

Assignment Statements

- Simple assignments:
  - **`A = 10`** or **`A := 10`** or **`A is 10`** or **`=(A,10)`**
  - In SML assignment is just another (infix) function
    - **`:= : ''a ref * ''a -> unit`**
- More complicated assignments:
  1. Multiple targets (PL/I)
     `A, B = 10`
  2. Conditional targets (C, C++, and Java)
     `(first==true)? total : subtotal = 0`
  3. Compound assignment operators (C, C++, and Java)
     `sum += next;`

Assignment Statements

- More complicated assignments (continued):

  4. Unary assignment operators (C, C++, and Java)

     a++; (increment a with one but return a)

     ++a; (increment a with one but return a+1)

     What does ++a-- evaluate to?

  C, C++, and Java treat = as an arithmetic binary operator

  e.g.

  a = b * (c = d * 2 + 1) + 1

  This is inherited from ALGOL 68

  – **=** Can be bad if it is overloaded for the relational operator for equality e.g. (PL/I) **A = B = C;**

  – Note difference from C

Assignment Statements

- Assignment as an Expression
  - In C, C++, and Java, the assignment statement produces a result
  - So, they can be used as operands in expressions e.g.

```
while ((ch = getchar())!=EOF){…}
```

  - Disadvantage
    - Another kind of expression side effect

Conditional Selection

- Design Considerations:
  - What controls the selection
  - What can be selected:
    - FORTRAN IF: **IF** (boolean_expr) statement

```
      IF (.NOT. condition) GOTO 20
           ...
           ...
    20 CONTINUE
```

    - Modern languages allow any kind of program block
  - What is the meaning of nested selectors

Conditional Selection

- Single-way
  - **`IF`** … **`THEN`** …
  - Controlled by boolean expression
- Two-way
  - **`IF`** … **`THEN`** … **`ELSE`**
  - Controlled by boolean expression
  - **`IF`** … **`THEN`** … usually treated as degenerate form of

    `IF` … **`THEN`** … **`ELSE`**
  - **`IF`**…**`THEN`** together with **`IF..THEN`**…**`ELSE`** require disambiguating associativity

Two-Way Selection Statements

- Nested Selectors
- e.g. (Java) **`if ...`**

```
        if ...
          ...
      else ...
```

- Which **`if`** gets the **`else`**?
- Java's static semantics rule: **`else`** goes with the nearest **`if`**

Two-Way Selection Statements

- ALGOL 60's solution - disallow direct nesting

```
if ... then
   begin
   if ...
      then ...
      else ...
   end
```

```
if ... then
   begin
   if ... then ...
   end
 else ...
```

Two-Way Selection Statements

- FORTRAN 90 and Ada solution – closing special words
  - e.g. (Ada)

```
if ... then
   if ... then
      ...
   else
      ...
   end if
 end if
```

  - Advantage: readability

```
if ... then
   if ... then
      ...
   end if
 else
   ...
 end if
```

- **`ELSEIF`**
  - Equivalent to nested **`if`**...**`then`**...**`else`**...

Multi-Way Conditional Selection

- **`SWITCH`**
  - Typically controlled by scalar type
  - Each selection has own block of statements it executes
  - What if no selection is given?
    - Language gives default behavior
    - Language forces total coverage, typically with programmer-defined default case
  - One block of code for whole switch
  - Selection specifies program point in block
  - **`break`** used for early exit from block

Switch on String in C#

```
Color ColorFromFruit(string s) {
   switch(s.ToLower()) {
      case "apple":
         return Color.Red;
      case "banana":
         return Color.Yellow;
      case "carrot":
         return Color.Orange;
      default:
         throw new InvalidArgumentException();

   }
}
```

Switch on Type in F#

```
type 'a Visitor =
  class
    abstract member visitPlusExp: 'a * 'a -> 'a
    abstract member visitMinusExp: 'a * 'a -> 'a
    abstract member visitTimesExp: 'a * 'a -> 'a
    abstract member visitDivideExp: 'a * 'a -> 'a
    abstract member visitIdentifier: string -> 'a
    abstract member visitIntegerLiteral: string -> 'a
    new() = {}
```

```
let rec TreeWalker (c:'a Visitor) (ee:Exp) =
  match ee with
  | :? PlusExp as e -> (c.visitPlusExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? MinusExp as e -> (c.visitMinusExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? TimesExp as e -> (c.visitTimesExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? DivideExp as e -> (c.visitDivideExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? Identifier as e -> (c.visitIdentifier e.f1)
  | :? IntegerLiteral as e -> (c.visitIntegerLiteral e.f1);;
```

```
type Interpreter =
  class
    inherit int Visitor
    override x.visitPlusExp (x,y) = x + y
    override x.visitMinusExp (x,y) = x - y
    override x.visitTimesExp (x,y) = x * y
    override x.visitDivideExp (x,y) = x / y
    override x.visitIdentifier s = Lookup s
    override x.visitIntegerLiteral s = System.Int32.Parse s
    new() = {}
end;;
```

Pattern matching in C# 7.0

The following is an example of pattern matching:

```
class Geometry();
class Triangle(int Width, int Height, int Base) : Geometry;
class Rectangle(int Width, int Height) : Geometry;
class Square(int width) : Geometry;

Geometry g = new Square(5);
switch (g)
{
    case Triangle(int Width, int Height, int Base):
        WriteLine($"{Width} {Height} {Base}");
        break;
    case Rectangle(int Width, int Height):
        WriteLine($"{Width} {Height}");
        break;
    case Square(int Width):
        WriteLine($"{Width}");
        break;
    default:
        WriteLine("<other>");
        break;
}
```

In the sample above you can see how we match on the data type and immediately destructure it into its components.

Loops

- Main types:
- Counter-controlled iterators (For-loops)
- Logical-test iterators
- Iterations based on data structures
- Recursion

For-loops

- Controlled by loop variable of scalar type with bounds and increment size
- Scope of loop variable?
  - Extends beyond loop?
  - Within loop?
- When are loop parameters calculated?
  - Once at start
  - At beginning of each pass

Iterative Statements

ALGOL 60 Design choices:

1. Control expression can be `int` or `real`; its scope is whatever it is declared to be
2. Control variable has its last assigned value after loop termination
3. The loop variable cannot be changed in the loop, but the parameters can, and when they are, it affects loop control
4. Parameters are evaluated with every iteration, making it very complex and difficult to read

Iterative Statements

Pascal:

- Syntax:

  `for` variable := initial (**`to`** | **`downto`**) final **`do`** statement

- Design Choices:

  1. Loop variable must be an ordinal type of usual scope
  2. After normal termination, loop variable is undefined
  3. The loop variable cannot be changed in the loop; the loop parameters can be changed, but they are evaluated just once, so it does not affect loop control
  4. Just once

Iterative Statements

Ada:

- Syntax:

  `for` var **`in`** [reverse] discrete_range **`loop`** ...

  `end loop`
- Design choices:
  1. Type of the loop variable is that of the discrete range; its scope is the loop body (it is implicitly declared)
  2. The loop variable does not exist outside the loop
  3. The loop variable cannot be changed in the loop, but the discrete range can; it does not affect loop control
  4. The discrete range is evaluated just once

Iterative Statements

C:

- Syntax:

  for ([expr_1] ; [expr_2] ; [expr_3]) statement

  – The expressions can be whole statements, or even statement sequences, with the statements separated by commas

  – The value of a multiple-statement expression is the value of the last statement in the expression

  e.g.,

```
for (i = 0, j = 10; j == i;  i++) …
```

  – If the second expression is absent, it is an infinite loop

Iterative Statements

- C Design Choices:
  1. There is no explicit loop variable
  2. Loop variable, if there is one, has whatever was assigned last
  3. Everything can be changed in the loop
  4. The first expression is evaluated once, but the other two are evaluated with each iteration
- This loop statement is the most flexible
- But also rather difficult to analyze ..

Iterative Statements

C++:

- Differs from C in two ways:
  1. The control expression can also be Boolean
  2. The initial expression can include variable definitions (scope is from the definition to the end of the loop body)

Java:

- Differs from C++ in that the control expression must be Boolean

Logic-Test Iterators

- While-loops
  - Test performed before entry to loop
- **`repeat`**...**`until`** and **`do`**...**`while`**
  - Test performed at end of loop
  - Loop always executed at least once
- Design Issues:
  1. Pretest or posttest?
  2. Should this be a special case of the counting loop statement (or a separate statement)?

Iterative Statements

C , C++, and Java – **`break`**:

- Unconditional; for any loop or **`switch`**; one level only (except Java's can have a label)
- There is also a **`continue`** statement for loops; it skips the remainder of this iteration, but does not exit the loop

Counter-Controlled Loops: Examples

- Python

  for loop_variable **in** object:

  – loop body

  [**else**:

  – else clause]

  – The object is often a range, which is either a list of values in brackets (`[2, 4, 6]`), or a call to the range function (**range**`(5)`, which returns `0, 1, 2, 3, 4`

  – The loop variable takes on the values specified in the given range, one for each iteration

  – The else clause, which is optional, is executed if the loop terminates normally

Iterative Statements

- Iteration Based on Data Structures
  - Concept: use order and number of elements of some data structure to control iteration
  - Control mechanism is a call to a function that returns the next element in some chosen order, if there is one; else exit loop
  - C's **`for`** can be used to build a user-defined iterator
  - e.g.

```
for (p=hdr; p; p=next(p))
{ ... }
```

  - Perl has a built-in iterator for arrays and hashes

    e.g.,

```
foreach $name (@names)
{ print $name }
```

C# Foreach Loops

```
foreach (T x in C) S
```

<u>is implemented as</u>

```
IEnumerable<T> c = C;
IEnumerator<T> e = c.GetEnumerator();
while (e.MoveNext())
{ T x = e.Current; S }
```

Recursion

- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code
  - i.e. Recursion is a technique that solves a problem by solving a smaller problem of the same type
  - How do I write recursive functions?
    - Determine the base case(s)
      - the one for which you know the answer
    - Determine the general case(s)
      - the one where the problem is expressed as a smaller version of itself
- Iteration can be used in place of recursion and visa versa
  - An iterative algorithm uses a looping construct
  - A recursive algorithm uses a branching structure

Recursion vs. iteration

- Recursive implementation

```
int Factorial(int n)
{
 if (n==0)
   return 1;
 else
   return n * Factorial(n-1);
}
```

- Iterative implementation

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2;
      count <= n;
      count++)
   fact = fact * count;

 return fact;
}
```

Counter-Controlled Loops: Examples

- F#
  - Because counters require variables, and functional languages do not have variables, counter-controlled loops must be simulated with recursive functions

```
let rec forLoop loopBody reps =
  if reps <= 0 then ()
  else
    loopBody()
    forLoop loopBody, (reps - 1)
```

  - This defines the recursive function `forLoop` with the parameters `loopBody` (a function that defines the loop's body) and the number of repetitions
  - `()` means do nothing and return nothing

Recursion vs. iteration

- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code
- Recursive solutions are often less efficient, in terms of both time and space, than iterative solutions
  - Well this is what the literature says …
  - This is usually true for languages such as C, Java and C# as method calls can be expensive and deep recursions can take up a lot of stack space
  - However, on modern hardware, functions calls call, especially tail recursive calls can be cheap. Thus modern functional languages like Haskell, SML, Scala and F# encourage recursion

Gotos

- Requires notion of program point
- Transfers execution to given program point
- Basic construct in machine language
- Implements loops
- Makes programs hard to read and reason about
- Hard to know how a program got to a given point
- Generally thought to be a bad idea in a high level language

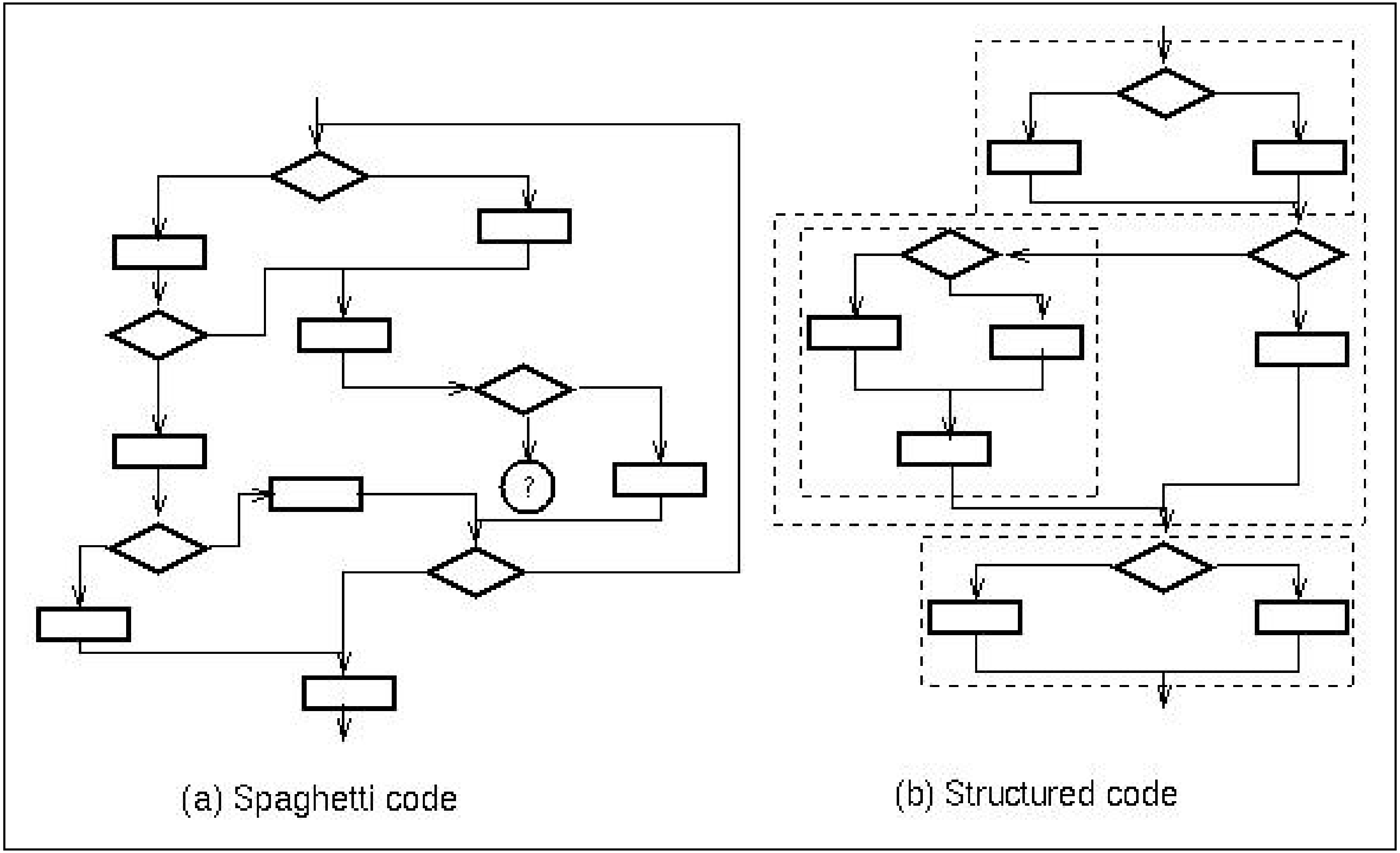
Fortran Control Structure

```
10 IF (X .GT. 0.000001) GO TO 20
11 X = -X
    IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
     X = X-Y-Y
30  X = X+Y
     ...
50 CONTINUE
     X = A
     Y = B-A
     GO TO 11
     …
```

Historical Debate

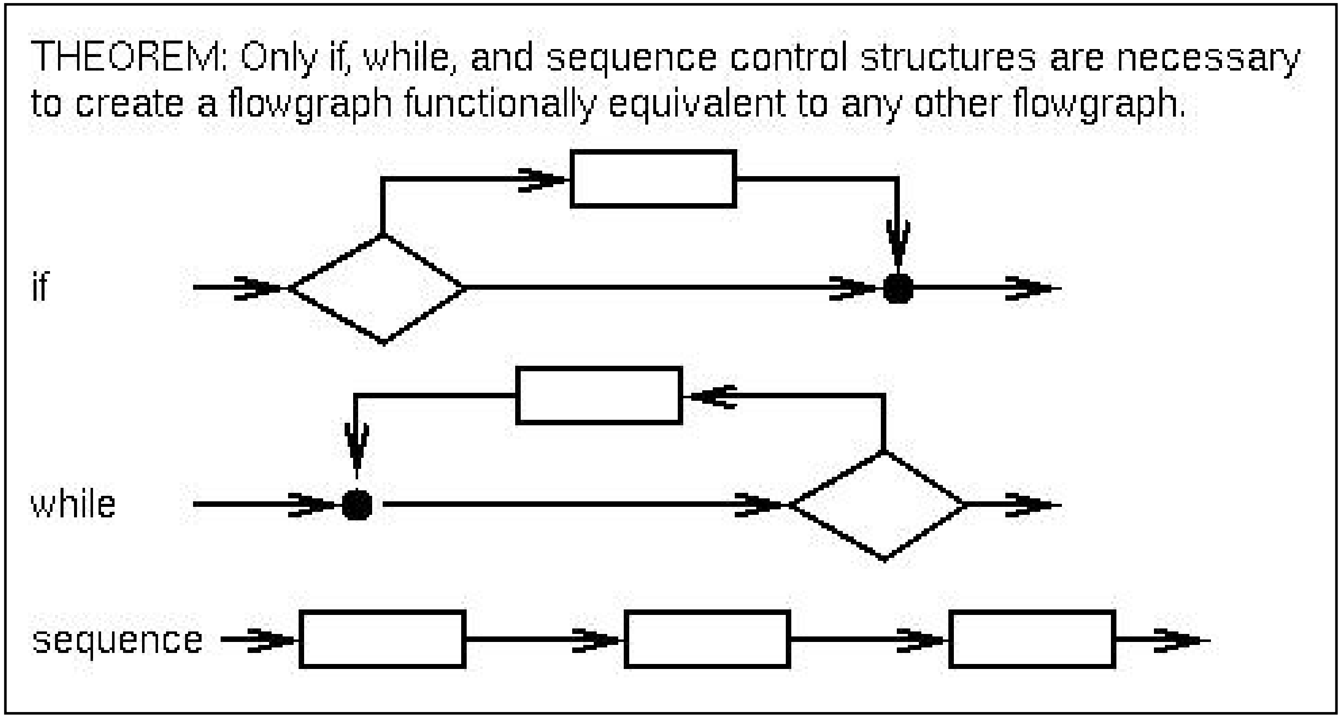
- Dijkstra, Go To Statement Considered Harmful
  - Letter to Editor, *C ACM*, March 1968
  - Now on web: http://www.acm.org/classics/oct95/
- Knuth, Structured Prog. with go to Statements
  - You can use goto, but do so in structured way …
- Continued discussion
  - Welch, GOTO (Considered Harmful)^n, n is Odd
- General questions
  - Do syntactic rules force good programming style?
  - Can they help?

Spaghetti code



Structured programming

- Issue in 1970s: Does this limit what programs can be written?
- Resolved by Structure Theorem of Böhm-Jacobini.
- Here is a graph version of theorem originally developed by Harlan Mills:



Advance in Computer Science

- Standard constructs that structure jumps

  `if … then … else … end`

  `while … do … end`

  `for … { … }`

  `case …`

- Modern style
  - Group code in logical blocks
  - Avoid explicit jumps except for function return
  - Cannot jump *into* middle of block or function body
- But there may be situations when "jumping" is the right thing to do!

Exceptions: Structured Exit

- Terminate part of computation
  - Jump out of construct
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
  - Unnecessary activation records may be deallocated
    - May need to free heap space, other resources
- Two main language constructs
  - Declaration to establish exception *handler*
  - Statement or expression to *raise* or *throw* exception

Often used for unusual or exceptional condition, but not necessarily.

Summary of Control of Statement Execution

- Sequential
- Conditional Selection
- Looping Construct
- Must have all three to provide full power of a Computing Machine
- Sometimes jumps are needed!

What can you do in your projects now?

- Revisit your token grammer and CFG
- Test front end implementation techniques:
  - Recursive decent by hand
  - JavaCC, ANTLR, Jflex/CUP, SableCC
  - Use a toy language or a subset of your own language
- Generate AST
- Make a pretty printing tree walker
  - Composit, Visitor (GOF, static overloading, reflexsive)
  - Test that programs you input come out roughly the same!
- Make a scope and type checking tree walker

Languages and Compilers (SProg og Oversættere)

Lecture 14-1

Programming Language Design – Subprograms

Bent Thomsen

Department of Computer Science

Aalborg University

Learning Goals

- Gain insigt into abstractions in programming languages
  - The principle of Abstraction

- Programming language design evaluation methods

Syntactic Elements

- Declarations and Definitions
  - Scopes and visibility
  - always before use or not, initialization or not,
- Expressions
- Statements
- **Subprograms**

- Separate subprogram definitions (Module system)
- Separate data definitions
- Nested subprogram definitions
- Separate interface definitions

Subprograms

1. A subprogram has a single entry point

2. The caller is suspended during execution of the called subprogram

3. Control always returns to the caller when the called subprogram's execution terminates

Functions or Procedures?

- Procedures provide user-defined statements
    - Abstractions over statements
- Functions provide user-defined operators
    - Abstractions over expressions
- Methods used for both functions and procedures

Subprograms

- Specification: name, signature, actions
  - C/C++: `typ0 f(typ1 para1, typ2 para2, ...) { ... }`
  - SML: `fun f para1 para2 = ...`
  - Pascal:
    ```
    function f(para1 : typ1, para2 : typ2, ...) : retval;
      var retval : typ0;
      begin  ...  end
    ```
- Signature: number and types of input arguments, number and types of output results
  - Sometimes this is called the subprogram protocol
- Actions: direct function relating input values to output values; side effects on global state and subprogram internal state
- May depend on implicit arguments in form of non-local variables

Local Referencing Environments

- Local variables can be stack-dynamic
  - Advantages
    - Support for recursion
    - Storage for locals is shared among some subprograms
  - Disadvantages
    - Allocation/de-allocation, initialization time
    - Indirect addressing
    - Subprograms cannot be history sensitive
- Local variables can be static
  - Advantages and disadvantages are the opposite of those for stack-dynamic local variables

Subprogram As Abstraction

- Subprograms encapsulate local variables and specifics of algorithm applied
  - Once compiled, programmer cannot access these details in other programs
  - In most languages subprogram definitions are not executables, but e.g. in Python a function definition is executed to bind the function name in the current local namespace to a function object
- Application of subprogram does not require user to know details of input data layout (just its type)
  - Form of information hiding

Basic Definitions

- Function declarations in C and C++ are often called *prototypes*
- A *subprogram declaration* provides the protocol, but not necessarily the body, of the subprogram
- A *formal parameter* is a (dummy) variable listed in the subprogram header and used in the subprogram
- An *actual parameter* represents a value or address used in the subprogram call statement
- A *subprogram definition* provides the body, of the subprogram and may provide the protocol

Actual/Formal Parameter Correspondence

- Positional
  - The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
  - Safe and effective
  - E.g. in C# PrintOrderDetails("Gift Shop", 31, "Red Mug");
- Keyword
  - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
  - *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors
  - *Disadvantage*: User must know the formal parameter's names
  - E.g. in C# PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");

Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)
  - In C++, default parameters must appear last because parameters are positionally associated

- Variable numbers of parameters
  - C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by **`params`**
  - In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.
  - In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk
  - In Lua, a variable number of parameters is represented as a formal parameter with three periods; they are accessed with a `for` statement or with a multiple assignment from the three periods

Subprogram Parameters

- Formal parameters: names (and types) of arguments to the subprogram used in defining the subprogram body
- Actual parameters: arguments supplied for formal parameters when subprogram is called
- *Actual/Formal Parameter Correspondence:*
  - attributes of variables are used to exchange information
    - Name – Call-by-name
    - Memory Location – Call-by reference
    - Value
      - Call-by-value (one way from actual to formal parameter)
      - Call-by-value-result (two ways between actual and formal parameter)
      - Call-by-result (one way from formal to actual parameter)

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)
  - *Disadvantages* (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

Pass-by-Reference (Inout Mode)

- Pass an access path
- Also called pass-by-sharing
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases (access broadened)

Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller, by physical move
  - Require extra storage location and copy operation
- Potential problem: `sub(p1, p1);` whichever formal parameter is copied back will represent the current value of `p1`

Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

Pass-by-Name (Inout Mode)

- By textual substitution
  - (or thunking – i.e. passing a function)
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding

Design Considerations for Parameter Passing

1. Efficiency
2. One-way or two-way

- These two are in conflict with one another!
  - Good programming → limited access to variables, which means one-way whenever possible
  - Efficiency → pass by reference is fastest way to pass structures of significant size
  - Also, functions should not allow reference parameters

Parameters that are Subprograms

- It is sometimes convenient to pass subprogram names or even subprograms as parameters
- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter?

- Note this is first class functions or lambdas which is now becoming part of mainstream languages!!

Parameters that are Subprogram Names: Parameter Type Checking

- C and C++: functions cannot be passed as parameters but pointers to functions can be passed and their types include the types of the parameters, so parameters can be type checked
- FORTRAN 95 type checks
- Ada does not allow subprogram parameters; an alternative is provided via Ada's generic facility
- Java until Java 8 did not allow method names to be passed as parameters
- C# supports functions a parameters through delegates
  - Delegates can now be anonymous or lambda expression
  - We talk about first class functions
- Functional languages supports functions as first class functions

Criteria in a good language design

- The criterias from Sebesta's book are well established ”rules of thumb”
- But until recently they had litlle or no research backing.
- Since 2009 a new directions in programming language design research has emerged
  - could be called Evidence based Programming Language Design
  - Use of social science methods

Table 1.1 Language evaluation criteria and the characteristics that affect them

Characteristic	CRITERIA		
	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

What is orthognality?

- "The number of independent primitive concepts has been minimized in order that the language be easy to describe, to learn, and to implement. On the other hand, these concepts have been applied "orthogonally" in order to maximize the expressive power of the language while trying to avoid deleterious superfluities"
  – Adriaan van Wijngaarden et al., Revised Report on the Algorithmic Language ALGOL 68, section 0.1.2, Orthogonal design

What is orthogonality?

- "A precise definition is difficult to produce, but languages that are called orthogonal tend to have a small number of core concepts and a set of ways of uniformly combining these concepts. The semantics of the combinations are uniform; no special restrictions exist for specific instances of combinations." – David Schmidt
  - Ex:
    - A[4+(F(X)-1)] OK in Algol but not in Fortran IV
    - Pascal, only values from the scalar types can be results from function procedures. In contrast, ML allows a function to return a value from any legal type whatsoever.

What is lack of orthogonality?

- The C language is somewhat inconsistent in its treatment of concepts and thus not as orthogonal as it could be
- Examples of exceptions follow:
  - Structures (but not arrays) may be returned from a function.
  - An array can be returned if it is inside a structure.
  - A member of a structure can be any data type
    - (except void, or the structure of the same type).
  - An array element can be any data type (except void).
  - Everything is passed by value (except arrays).
  - Void can be used as a type in a structure, but a variable of this type cannot be declared in a function.

Tennent's Language Design principles

- The Principle of Abstraction
  - All major syntactic categories should have abstractions defined over them. For example, functions are abstractions over expressions
- The Principle of Correspondence
  - Declarations ≈ Parameters
- The Principle of Data Type Completeness
  - All data types should be first class without arbitrary restriction on their use

–Originally defined by R.D.Tennent

Principle of correspondence

- Example in Pascal:

```
var i : integer;
begin
  i := -j;
  write(i)
end
```

and

```
procedure p(i : integer);
begin
  write(i)
end;
begin p(-j) end
```

- Are equivalent

Example of missing correspondence

In Pascal:

```
procedure inc(var i : integer);
  begin
    i := i + 1
  end;

var x : integer;
begin
  x := 1;
  inc(x);
  writeln(x);
end
```

No corresponding declaration

However C has correspondence

```
void inc(int *i) {
  *i = *i + 1;
}

int x = 1;
inc(&x);
printf("%d", x);

int x = 1;
{
  int *i = &x;
  *i = *i + 1;
}
printf("%d", x);
```

The Concept of Abstraction

- The concept of abstraction is fundamental in programming (and computer science)
- Tennents principle of abstraction
  - is based on identifying all of the semantically-meaningful syntactic categories of the language and then designing a coherent set of abstraction facilities for each of these.
- Nearly all programming languages support process (or command) abstraction with subprograms (procedures)
- Many programming languages support expression abstraction with functions
- Nearly all programming languages designed since 1980 have supported data abstraction:
  - Abstract data types
  - Objects
  - Modules

Cognitive Dimensions

- Developed by Thomas Green, Univ. of Leeds
- Used to analyze the *usability of information artifacts*
- Applied to discover useful things about usability problems that are not easily analyzed using conventional techniques
- Framework (as opposed to model or theory)

Cognitive Dimensions (2)

- Focused on ***notations***, such as
  - Music, Dance
  - Programming languages
- And on ***information handling devices***, such as
  - Spreadsheets
  - Database query systems
  - IDEs
- Gives descriptions of aspects, attributes, or ways that a user thinks about a system, called dimensions
- The 14 dimensions (and more have been added)

Dimensions

- Abstraction
  - types and availability of abstraction mechanisms
- Hidden dependencies
  - important links between entities are not visible
- Premature commitment
  - constraints on the order of doing things
- Secondary notation
  - extra information in means other than formal syntax
- Viscosity
  - resistance to change
- Visibility
  - ability to view components easily

Abstractions

- ***Types and availability of abstraction mechanisms***

- ***An abstraction is a class of entities or grouping of elements to be treated as one entity*** (thereby lowering viscosity).
- Abstraction barrier:
  - minimum number of new abstractions that must be mastered before using the system (e.g. *Z*)
- Abstraction hunger:
  - require user to create abstractions

Abstraction features

- Abstraction-tolerant systems:
  - permit but do not require user abstractions (e.g. word processor styles)
- Abstraction-hating systems:
  - do not allow definition of new abstractions (e.g. spreadsheets)
- Abstraction *changes the notation*.

Abstraction implications

- Abstractions are hard to create and use
- Abstractions must be maintained
  - useful for modification and transcription
  - increasingly used for personalisation
- Involve the introduction of an *abstraction manager* sub-device
  - including its own viscosity, hidden dependencies, juxtaposability etc.

Hidden Dependencies

- Important links between entities are not visible
- *Examples:*
  - class hierarchies
  - HTML links
  - spreadsheet cells

Secondary Notation

- Extra information in means other than formal syntax
- *Examples:*
  - Comments in programming languages
  - Pop-up boxes for icons
  - Well-designed icons

Viscosity

- Resistance to change
  - Fixed mental model
  - Hard-coded structure
- *Examples:*
  - Technical literature, with cross-references and section headings (because introducing a new section requires many changes to cross-references)

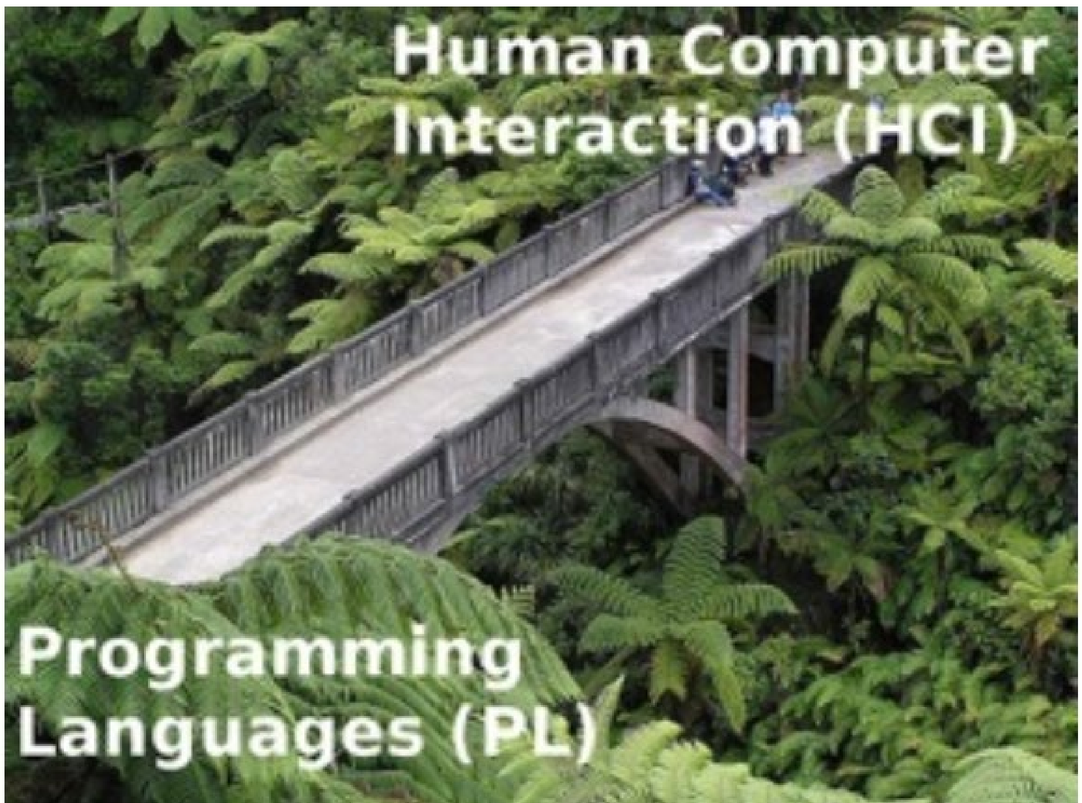
Further Dimmensions

- Closeness of mapping
  - closeness of representation to domain
- Consistency
  - similar semantics expressed in similar forms
- Diffuseness
  - verbosity of language
- Error-proneness
  - notation invites mistakes
- Hard mental operations
  - high demand on cognitive resources
- Progressive evaluation
  - work-to-date checkable any time
- Provisionality
  - degree of commitment to actions or marks
- Role-expressiveness
  - component purpose is readily inferred
- And more …
  - several new dimensions still under discussion

Supplementary Material

- Cognitive Dimensions of Notations website
  www.cl.cam.ac.uk/~afb21/CognitiveDimensions
- 10th Anniversary CD of Notations Workshop
  www.cl.cam.ac.uk/~afb21/CognitiveDimensions/workshop2005/index.html

PLATEAU - ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools



Programming Language design

- Designing a new programming language or extending an existing programming language usually follows an iterative approach:

1. Create ideas for the programming language or extensions
2. Describe/define the programming language or extensions
3. Implement the programming language or extensions
4. Evaluate the programming language or extensions
5. If not satisfied, goto 1

Discount Method for Evaluating Programming Languages

1. Create tasks specific to the language being tested - tasks that the participants of the experiment should solve. Estimate the time needed for each task (max 1 hour)
2. Create a short sample sheet of code examples in the language being tested, which the participants can use as a guideline for solving the tasks.
3. Prepare setup (e.g. use of NotePad++ and recorder) and do a sample test with 1 person.
   - Adjust tasks if needed
4. Perform the test on each participant, i.e. make them solve the tasks defined in step 1. (Use approx. 5 test persons)
5. Each participant should be interviewed briefly after the test, where the language and the tasks can be discussed.
6. Analyze the resulting data to produce a list of problems
   - Cosmetic problems, Serious problems, Critical problems

Discount Method for Evaluating Programming Languages

- Method inspired by the Discount Usability Evaluation (DUE) method and Instant Data Analysis (IDA) method
- Reference:
  - Svetomir Kurtev, Tommy Aagaard Christensen, and Bent Thomsen.
  - Discount method for programming language evaluation.
  - In Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2016). ACM, New York, NY, USA, 1-8. DOI: https://doi.org/10.1145/3001878.3001879

What can you do in your project now?

- Design abstractions
  - Functions and/or Procedures or ..

- Evaluate your language design
  - Revisit Sebesta’s design criteria
  - Tennent’s principles
  - Cognitive dimmensions
  - Discount Method for Evaluating Programming Languages

Languages and Compilers
(SProg og Oversættere)

Lecture 14 – 2
Interpreters

Bent Thomsen

Department of Computer Science

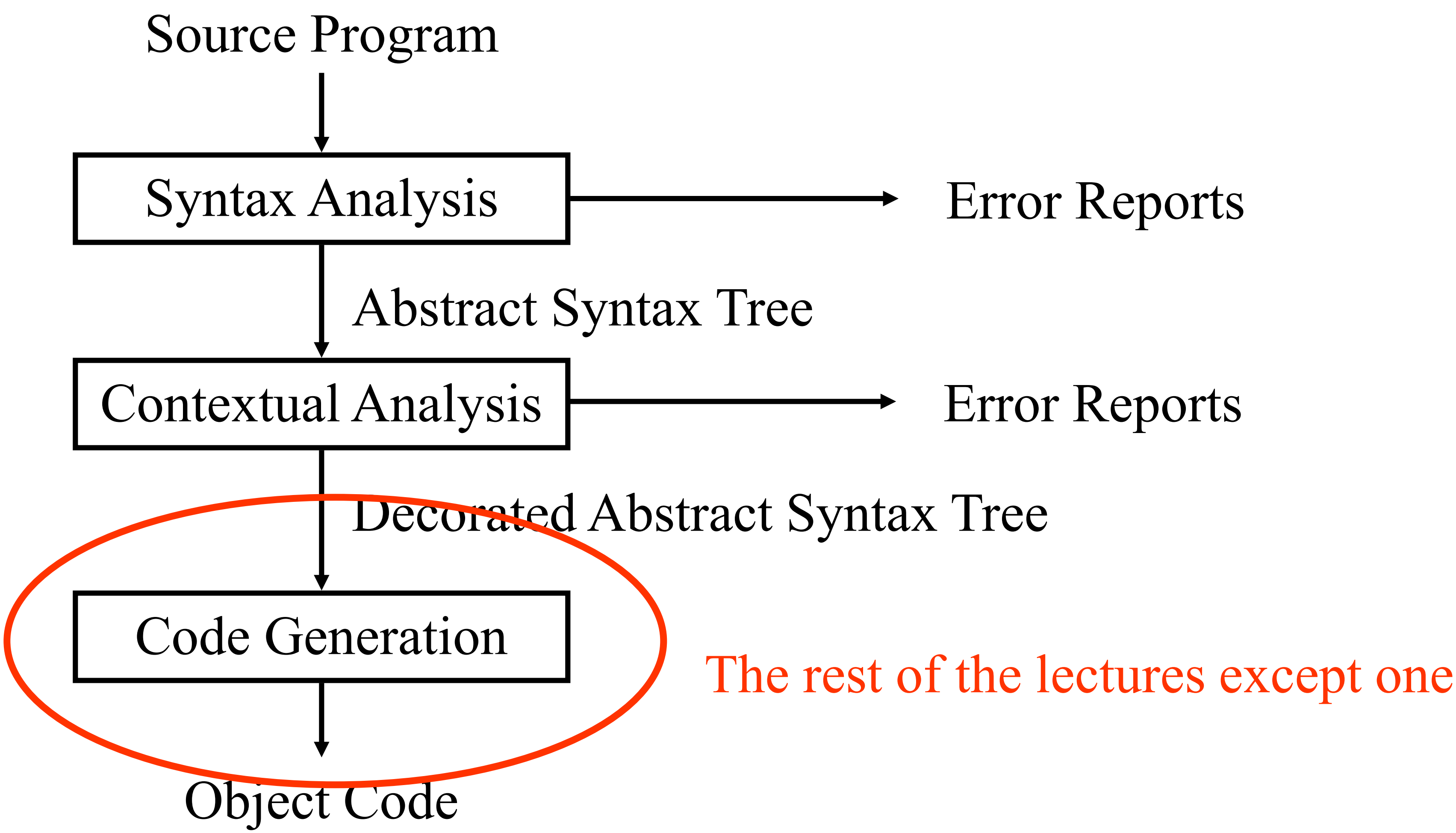
Aalborg University

With acknowledgement to Norm Hutchinson whose slides this lecture is based on.

Learning goals

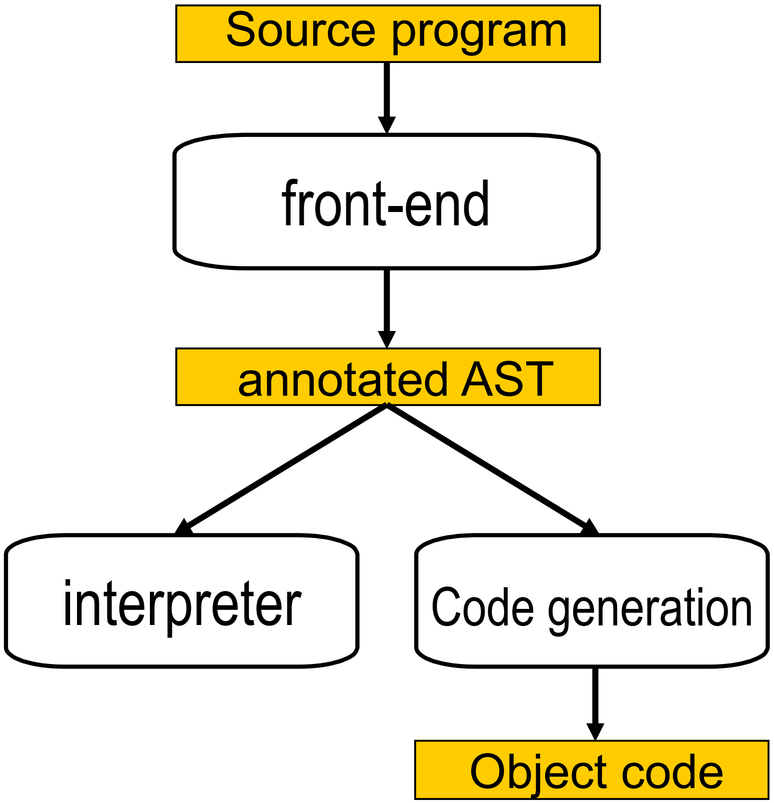
- To get an undertanding of interpretation
  - Recursive interpretation
  - Iterative interpretation

The “Phases” of a Compiler



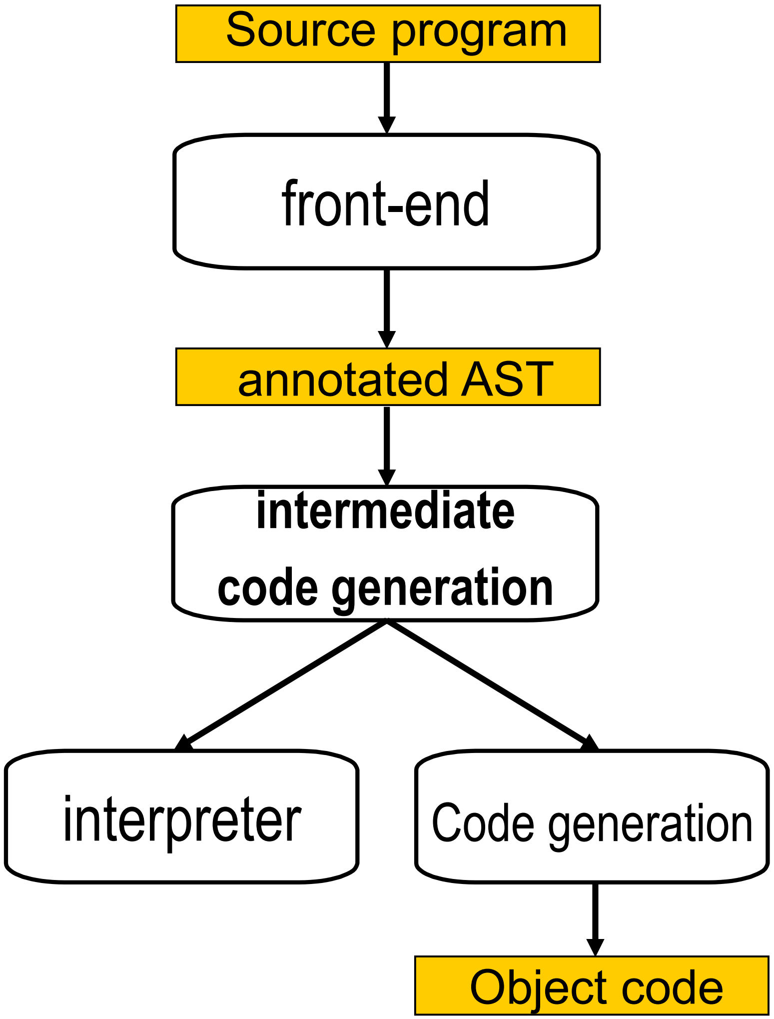
What’s next?

- interpretation
- code generation
  - code selection
  - register allocation
  - instruction ordering



What's next?

- intermediate code
- interpretation
- code generation
  - code selection
  - register allocation
  - instruction ordering



Intermediate code

- language independent
  - no (or few) structured types,
    only basic types (char, int, float)
  - no structured control flow,
    only (un)conditional jumps

- linear format
  - Java byte code

The usefulness of Interpreters

- Quick implementation of new language
  - Remember bootstrapping
- Testing and debugging
- Portability via Abstract Machine
- Hardware emulation

Interpretation

- recursive interpretation
  - operates directly on the AST [attribute grammar]
  - simple to write
  - thorough error checks
  - very slow: speed of compiled code 100 times faster
- iterative interpretation
  - operates on intermediate code
  - good error checking
  - slow: 10x

Recursive interpretation

- Two phased strategy
  - Fetch and analyze program
    - Recursively analyzing the phrase structure of source
    - Generating AST
    - Performing semantic analysis
      - Recursively via visitor
  - Execute program
    - Recursively by walking the decorated AST

Change the calc.cup

```
terminal           PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
terminal Integer  NUMBER;
non terminal Integer expr;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
expr  ::= expr:e1 PLUS expr:e2
       {: RESULT = new Integer(e1.intValue()+ e2.intValue()); :}
      | expr:e1 MINUS expr:e2
       {: RESULT = new Integer(e1.intValue()- e2.intValue());  :}
      | expr:e1 TIMES expr:e2
       {: RESULT = new Integer(e1.intValue()* e2.intValue());  :}
      | expr:e1 DIVIDE expr:e2
       {: RESULT = new Integer(e1.intValue()/ e2.intValue());  :}
      | LPAREN expr:e RPAREN {: RESULT = e;    :}
      | NUMBER:e {: RESULT= e; :}
```

Recursive Interpreter for Mini Triangle

Representing Mini Triangle values in Java:

```
public abstract class Value { }

public class IntValue extends Value {
        public short i;
}

public class BoolValue extends Value {
        public boolean b;
}

public class UndefinedValue extends Value { }
```

Recursive Interpreter for Mini Triangle

A Java class to represent the state of the interpreter:

```
public class MiniTriangleState {
       public static final short DATASIZE = …;

       //Code Store
       Program program; //decorated AST
       //Data store
       Value[] data = new Value[DATASIZE];
       //Register …
       byte status;
       public static final byte  //status value
              RUNNING = 0, HALTED = 1, FAILED = 2;
}
```

Recursive Interpreter for Mini Triangle

```
public class MiniTriangleProcesser
        extends MiniTriangleState implements Visitor {

        public void fetchAnalyze () {
                //load the program into the code store after
                //performing syntactic and contextual analysis
        }
        public void run () {
                … // run the program
        public Object visit…Command
                        (…Command com, Object arg) {
                //execute com, returning null (ignoring arg)
        }
        public Object visit…Expression
                        (…Expression expr, Object arg) {
                //Evaluate expr, returning its result
        }
        public Object visit…
}
```

Recursive Interpreter for Mini Triangle

```
public Object visitAssignCommand
                    (AssignCommand com, Object arg) {
      Value val = (Value) com.E.visit(this, null);
      assign(com.V, val);
      return null;
}

public Objects visitCallCommand
                    (CallCommand com, Object arg) {
      Value val = (Value) com.E.visit(this, null);
      CallStandardProc(com.I, val);
      return null;
}

public Object visitSequentialCommand
                    (SequentialCommand com, Object arg) {
      com.C1.visit(this, null);
      com.C2.visit(this, null);
      return null;
}
```

Recursive Interpreter for Mini Triangle

```
public Object visitIfCommand
                (IfCommand com, Object arg) {
        BoolValue val = (BoolValue) com.E.visit(this, null);
        if (val.b) com.C1.visit(this, null);
        else       com.C2.visit(this, null);
        return null;
}

public Object visitWhileCommand
                (WhileCommand com, Object arg) {
        for (;;) {
                BoolValue val = (BoolValue) com.E.visit(this, null)
                if (! Val.b) break;
                com.C.visit(this, null);
        }
        return null;
}
```

Recursive Interpreter for Mini Triangle

```
public Object visitIntegerExpression
                    (IntegerExpression expr, Object arg){
      return new IntValue(Valuation(expr.IL));
}
public Object visitVnameExpression
                    (VnameExpression expr, Object arg) {
      return fetch(expr.V);
}
…
public Object visitBinaryExpression
                    (BinaryExpression expr, Object arg){
      Value val1 = (Value) expr.E1.visit(this, null);
      Value val2 = (Value) expr.E2.visit(this, null);
      return applyBinary(expr.O, val1, val2);
}
```

Recursive Interpreter for Mini Triangle

```
public Object visitConstDeclaration
                    (ConstDeclaration decl, Object arg){
      KnownAddress entity = (KnownAddress) decl.entity;
      Value val = (Value) decl.E.visit(this, null);
      data[entity.address] = val;
      return null;
}
public Object visitVarDeclaration
                    (VarDeclaration decl, Object arg){
      KnownAddress entity = (KnownAddress) decl.entity;
      data[entity.address] = new UndefinedValue();
      return null;
}
public Object visitSequentialDeclaration
                    (SequentialDeclaration decl, Object arg){
      decl.D1.visit(this, null);
      decl.D2.visit(this, null);
      return null;
}
```

Recursive Interpreter for Mini Triangle

```
Public Value fetch (Vname vname) {
      KnownAddress entity =
            (KnownAddress) vname.visit(this, null);
      return data[entity.address];
}
Public void assign (Vname vname, Value val) {
      KnownAddress entity =
            (KnownAddress) vname.visit(this, null);
      data[entity.address] = val;
}
Public void fetchAnalyze () {
      Parser parse = new Parse(…);
      Checker checker = new Checker(…);
      StorageAllocator allocator = new StorageAllocator();
      program = parser.parse();
      checker.check(program);
      allocator.allocateAddresses(program);
}
Public void run () {
      program.C.visit(this, null);
}
```

Recursive Interpreter and Semantics

- Code for Recursive Interpreter is very close to a denotational semantics
- (see chapter 14 p. 211-221 in Transitions and Trees)

$$\mathcal{S}_{\text{ds}}[\![x := a]\!]s = s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$\mathcal{S}_{\text{ds}}[\![\texttt{skip}]\!] = \text{id}$$

$$\mathcal{S}_{\text{ds}}[\![S_1 \texttt{ ; } S_2]\!] = \mathcal{S}_{\text{ds}}[\![S_2]\!] \circ \mathcal{S}_{\text{ds}}[\![S_1]\!]$$

$$\mathcal{S}_{\text{ds}}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] = \text{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\text{ds}}[\![S_1]\!], \mathcal{S}_{\text{ds}}[\![S_2]\!])$$

$$\mathcal{S}_{\text{ds}}[\![\texttt{while } b \texttt{ do } S]\!] = \text{FIX } F$$

$$\text{where } F\ g = \text{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{\text{ds}}[\![S]\!], \text{id})$$

Recursive Interpreter and Semantics

- Code for Recursive Interpreter can be derived from big step semantics

$$[\text{plus}_{\text{bss}}] \qquad \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 + a_2 \rightarrow_a v} \qquad \text{hvor } v = v_1 + v_2$$

```
public Object visitBinaryExpression
                     (BinaryExpression expr, Object arg){
       Value val1 = (Value) expr.E1.visit(this, null);
       Value val2 = (Value) expr.E2.visit(this, null);
       return applyBinary(expr.O, val1, val2);
}
```

Recursive Interpreter and Semantics

- Code for Recursive Interpreter can be derived from big step semantics

$$[\text{ass}_{\text{bss}}] \qquad \langle x := a, s\rangle \to s[x \mapsto v] \qquad \text{hvor } s \vdash a \to_a v$$

```
public Object visitAssignCommand
                    (AssignCommand com, Object arg) {
      Value val = (Value) com.E.visit(this, null);
      assign(com.V, val);
      return null;
}

Public void assign (Vname vname, Value val) {
      KnownAddress entity =
            (KnownAddress) vname.visit(this, null);
      data[entity.address] = val;
}
```

Recursive Interpreters

- Usage
  - Quick implementation of high-level language
    - LISP, SML, Prolog, … , all started out as interpreted languages
  - Scripting languages
    - If the language is more complex than a simple command structure we need to do all the front-end and static semantics work anyway.
    - Web languages
      - JavaScript, PhP, ASP where scripts are mixed with HTML or XML tags

Iterative interpretation

- Follows a very simple scheme:

```
Initialize
Do {
        fetch next instruction
        analyze instruction
        execute instruction
} while (still running)
```

- Typical source language will have several instructions
- Execution then is just a big case statement
  - one for each instruction

Iterative Interpreters

- Command languages
- Query languages
  - SQL
- Simple programming languages
  - Basic
- Virtual Machines

Mini-Shell

Script	::=	Command*
Command	::=	Command-Name Argument* end-of-line
Argument	::=	Filename
	|	Literal
Command-Name	::=	**create**
	|	**delete**
	|	**edit**
	|	**list**
	|	**print**
	|	**quit**
	|	Filename

Mini-Shell Interpreter

```
Public class MiniShellCommand {
      public String    name;
      public String[]  args;
}

Public class MiniShellState {
      //File store…
      public …

      //Registers
      public byte status; //Running or Halted or Failed

      public static final byte // status values
            RUNNING = 0, HALTED = 1, FAILED = 2;
}
```

Mini-Shell Interpreter

```
Public class MiniShell extends MiniShellState {
        public void Interpret () {
                … // Execute the commands entered by the user
                  // terminating with a quit command
        }
        public MiniShellCommand readAnalyze () {
                … //Read, analysze, and return
                  //the next command entered by the user
        }
        public void create (String fname) {
                … // Create empty file wit the given name
        }
        public void delete (String[] fnames) {
                … // Delete all the named files
        }
        …
        public void exec (String fname, String[] args) {
                … //Run the executable program contained in the
                … //named files, with the given arguments
        }
}
```

Mini-Shell Interpreter

```
Public void interpret () {
      //Initialize
      status = RUNNING;
      do {
            //Fetch and analyse the next instruction
            MiniShellCommand com = readAnalyze();

            // Execute this instruction
            if (com.name.equals("create"))
                  create(com.args[0]);
            else if (com.name.equals("delete"))
                  delete(com.args)
            else if …

            else if (com.name.equals("quit"))
                  status = HALTED;
            else status = FAILED;
      } while (status == RUNNING);
}
```

Hypo: a Hypothetic Abstract Machine

- 4096 word code store
- 4096 word data store
- PC: program counter, starts at 0
- ACC: general purpose register
- 4-bit op-code
- 12-bit operand
- Instruction set:

Op-code	Instruction	Meaning
0	STORE d	word at address d $\leftarrow$ ACC
1	LOAD d	ACC $\leftarrow$ word at address d
2	LOADL d	ACC $\leftarrow$ d
3	ADD d	ACC $\leftarrow$ ACC $+$ word at address d
4	SUB d	ACC $\leftarrow$ ACC $-$ word at address d
5	JUMP d	PC $\leftarrow$ d
6	JUMPZ d	PC $\leftarrow$ d, if ACC $= 0$
7	HALT	stop execution

Hypo Interpreter Implementation (1)

```
public class HypoInstruction {
  public byte op;         // op-code field
  public short d;         // operand field

  public static final byte
    STOREop = 0,
     ...
}

public class HypoState {
  public static final short CODESIZE = 4096;
  public static final short DATASIZE = 4096;

  public HypoInstruction [] code = new HypoInstruction[CODESIZE];

  public short [] data = new short[DATASIZE];

  public short PC;
  public short ACC;
  public byte status;

  public static final byte
    RUNNING = 0, HALTED = 1, FAILED = 2;
}
```

Hypo Interpreter Implementation (2)

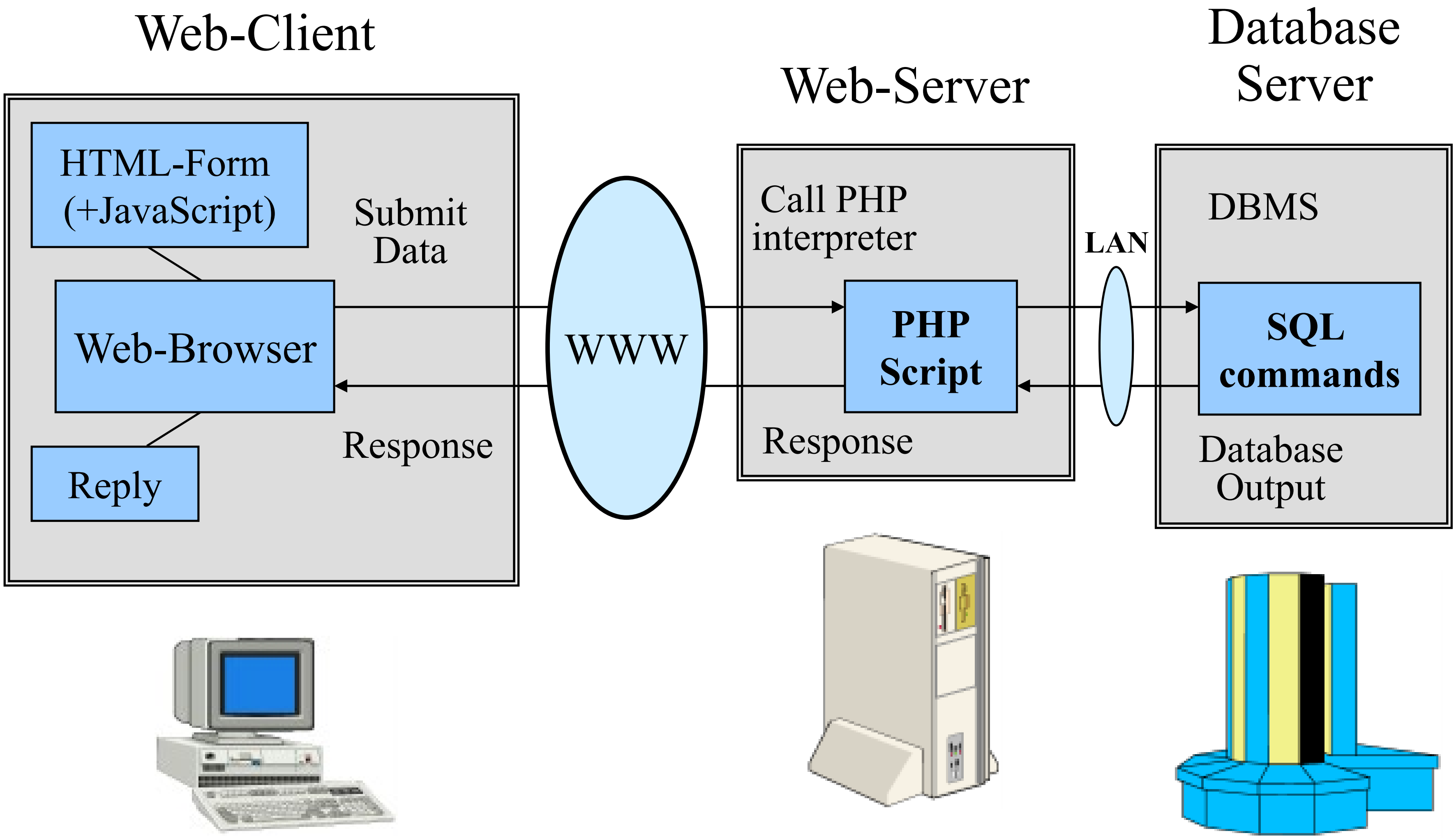
```
public class HypoInterpreter extends HypoState {
  public void load () { ... }
  public void emulate() {
    PC = 0; ACC = 0; status = RUNNING;
    do {
      // fetch:
      HypoInstruction instr = code[PC++];

      // analyse:
      byte op = instr.op;
      byte d  = instr.d;

      // execute:
      switch (op) {
        case STOREop: data[d] = ACC; break;
        case LOADop:  ACC = data[d]; break;
        ...
      }
    } while (status == RUNNING);
}
```

Other iterative interpreters

- Java Virtual Machine (JVM)
- .Net CLR
- Dalvik VM

- Note: LLVM is not a traditional virtual machine !
  - However LLVM provides an IR that can be used for further compilation

Interpreters are everywhere on the web



Interpreters versus Compilers

Q: What are the tradeoffs between compilation and interpretation?

Compilers typically offer more advantages when

- programs are deployed in a production setting
- programs are "repetitive"
- the instructions of the programming language are complex

Interpreters typically are a better choice when

- we are in a development/testing/debugging stage
- programs are run once and then discarded
- the instructions of the language are simple
- the execution speed is overshadowed by other factors
  - e.g. on a web server where communications costs are much higher than execution speed

What can you do in your project now

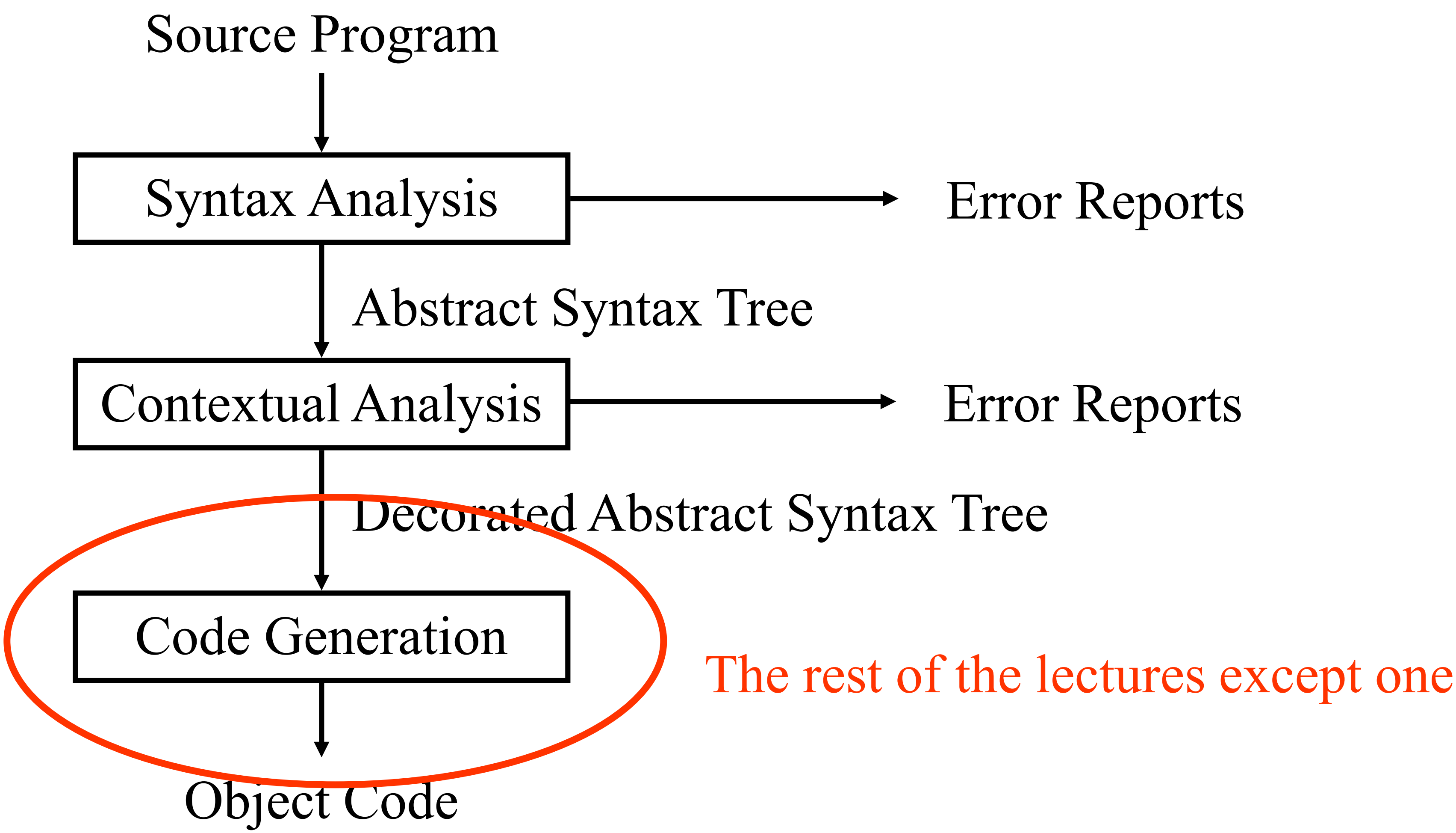
- Build a recursive interpreter!

Languages and Compilers (SProg og Oversættere)

Lecture 15
Intermediate Representations

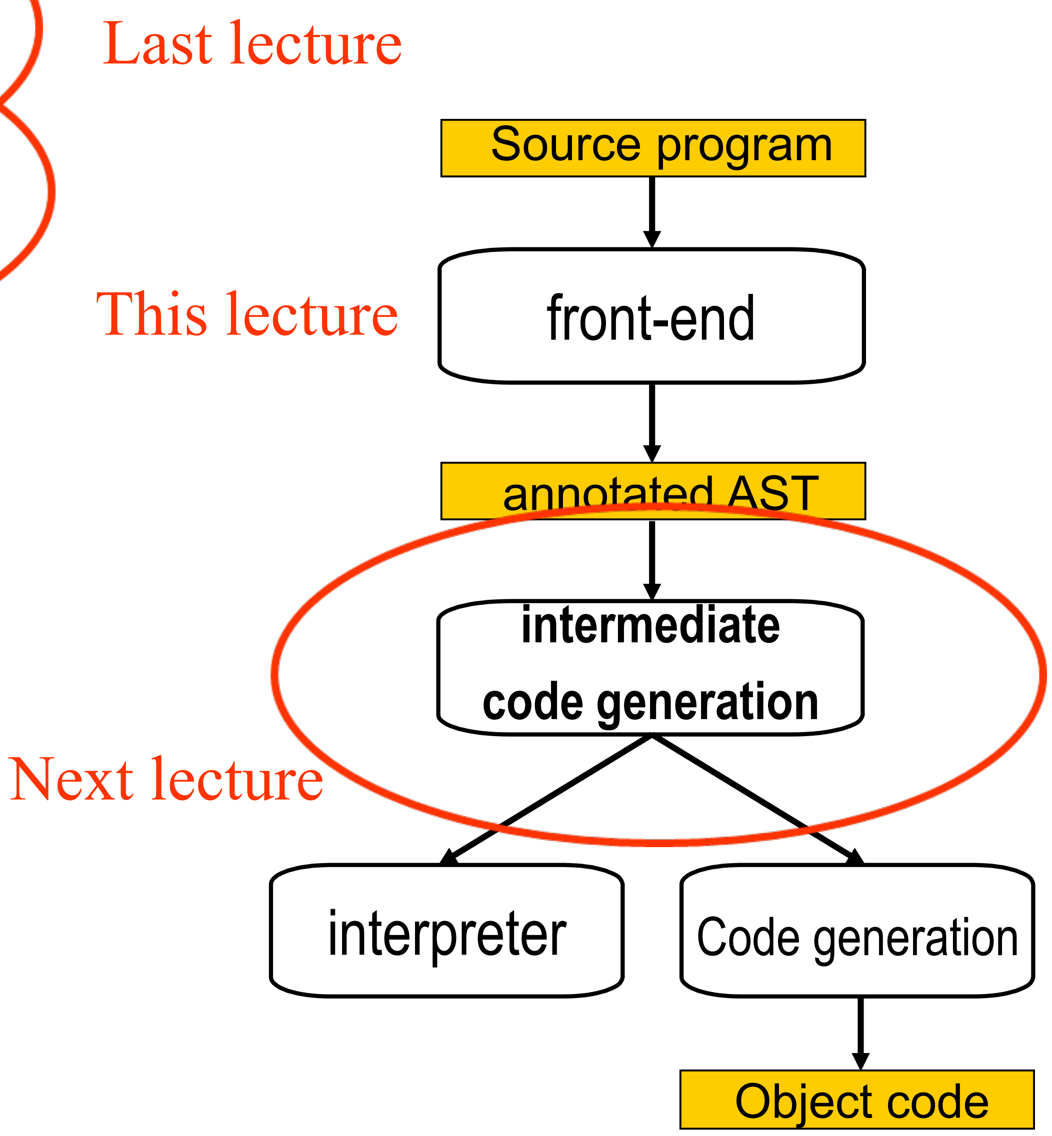
Bent Thomsen
Department of Computer Science
Aalborg University

The “Phases” of a Compiler

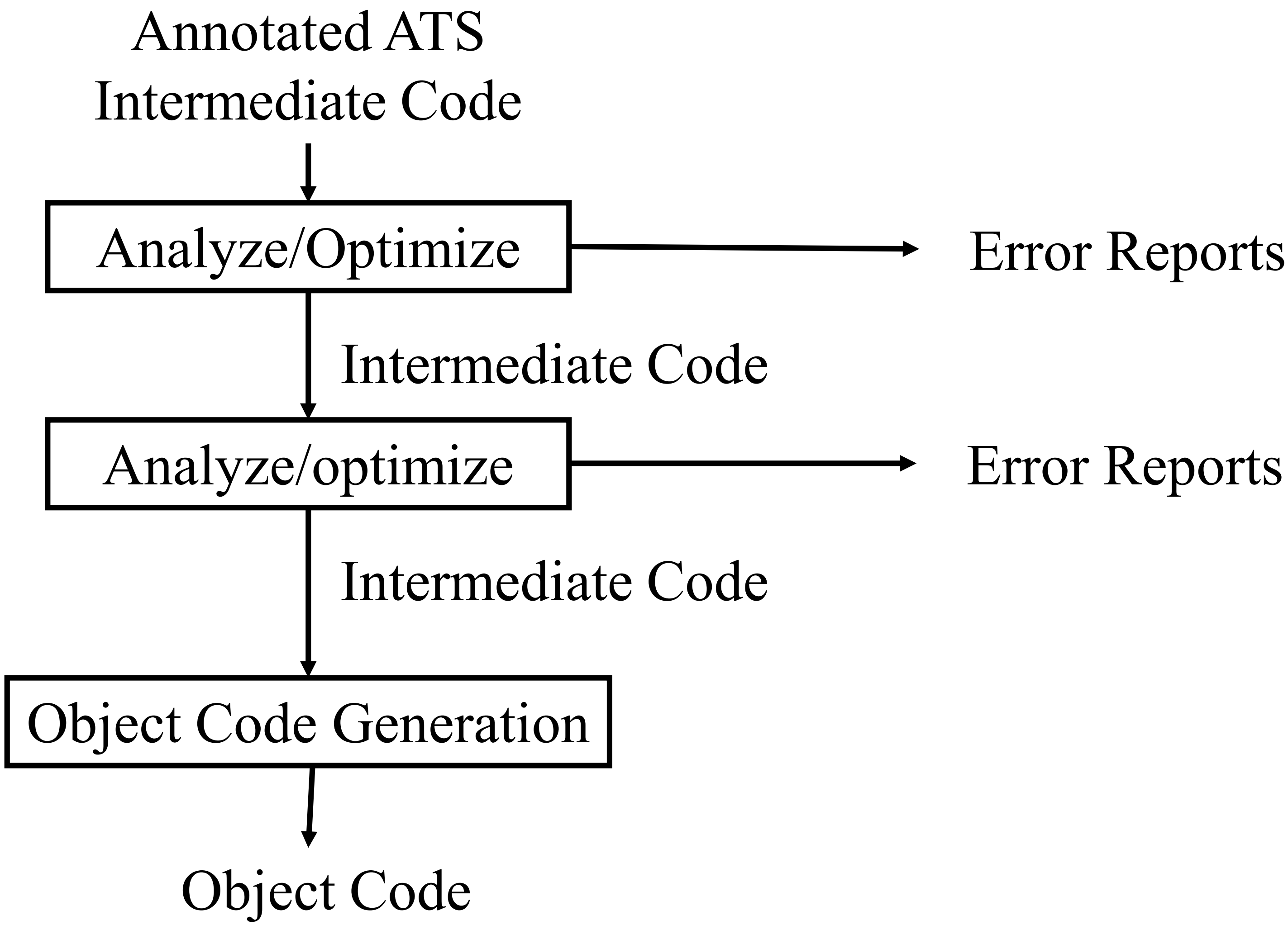


What's next?

- interpretation
- intermediate code
- code generation
  - code selection
  - register allocation
  - instruction ordering



The Code generation "Phases" of a Compiler



Intermediate Representations

- Abstract Syntax Tree
  - Convenient for semantic analysis phases
  - Convenient for recursive interpretation
  - We can generate code directly from the AST, but...
  - What about multiple target architectures?
- Intermediate Representation
  - "Neutral" architecture
  - Easy to translate to native code
  - Can abstracts away complicated runtime issues
    - Stack Frame Management
    - Memory Management
    - Register Allocation

Overview

- Semantic gap between high-level source languages and target machine language
- Examples
  - Early C++ compilers
    - cpp: preprocessor
    - cfront: translate C++ into C
    - C compiler

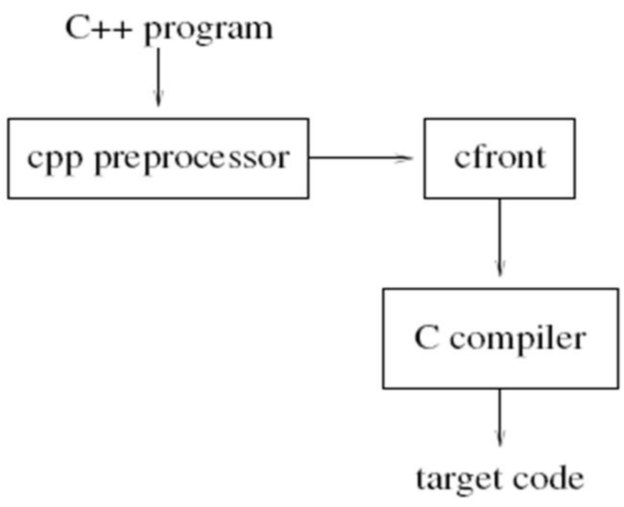


Figure 10.1: Use of `cfront` to translate C++ to C.

Another Example

- LaTeX
  - TeX: designed by Donald Knuth
  - dvi: device-independent intermediate representation
  - Ps: PostScript
  - pixels
- Portability enhanced

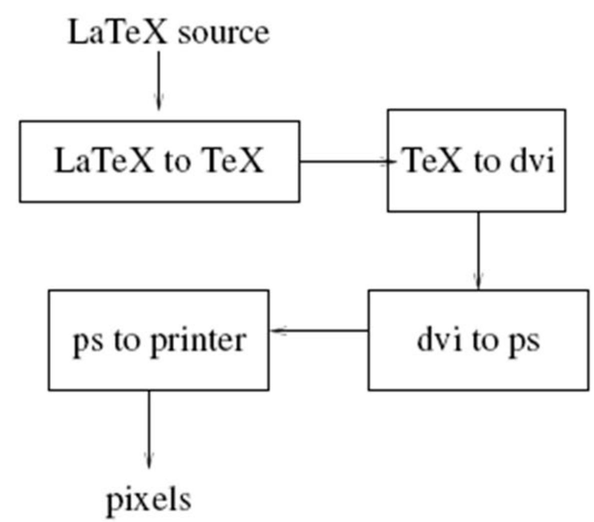


Figure 10.2: Translation from LaTeX into print.

Challenges

- Challenges
  - An intermediate language (IL) must be precisely defined
  - Translators and processors must be crafted for an IL
  - Connections must be made between levels so that feedback from intermediate steps can be related to the source program
- Other concerns
  - Efficiency

- Compiler suites that host multiple source languages and target multiple instruction sets obtain great leverage from a middle-end
  - Ex: s source languages, t target languages
    - s*t vs. s+t
-

s*t vs. s+t

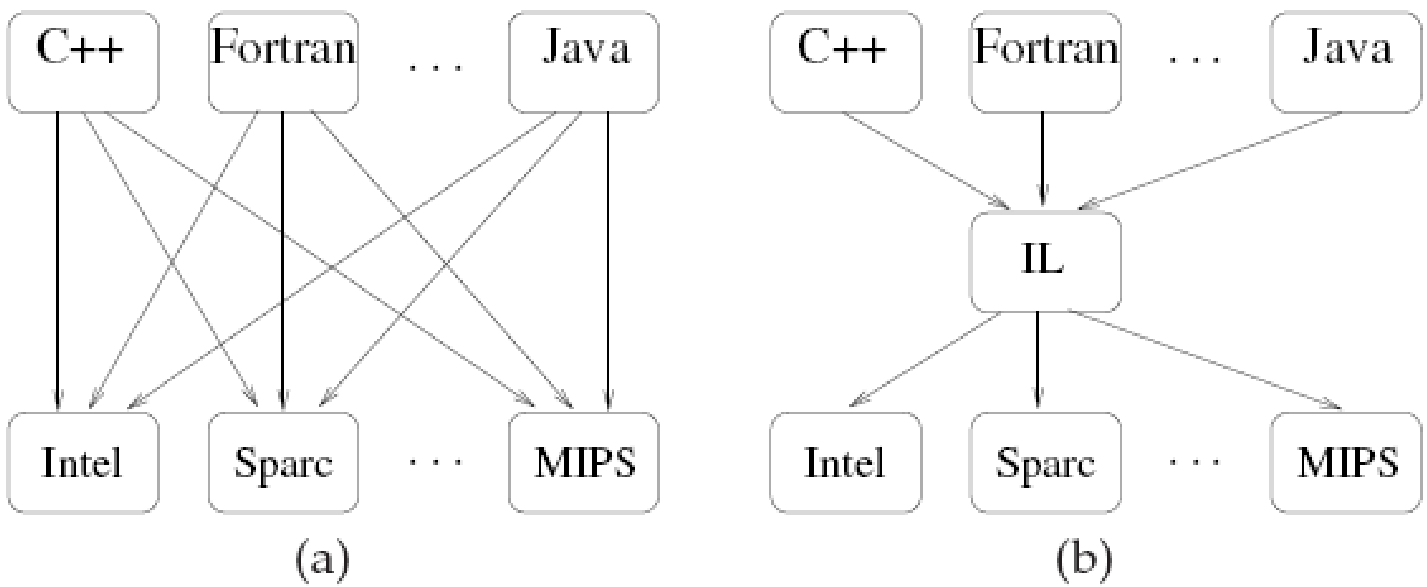


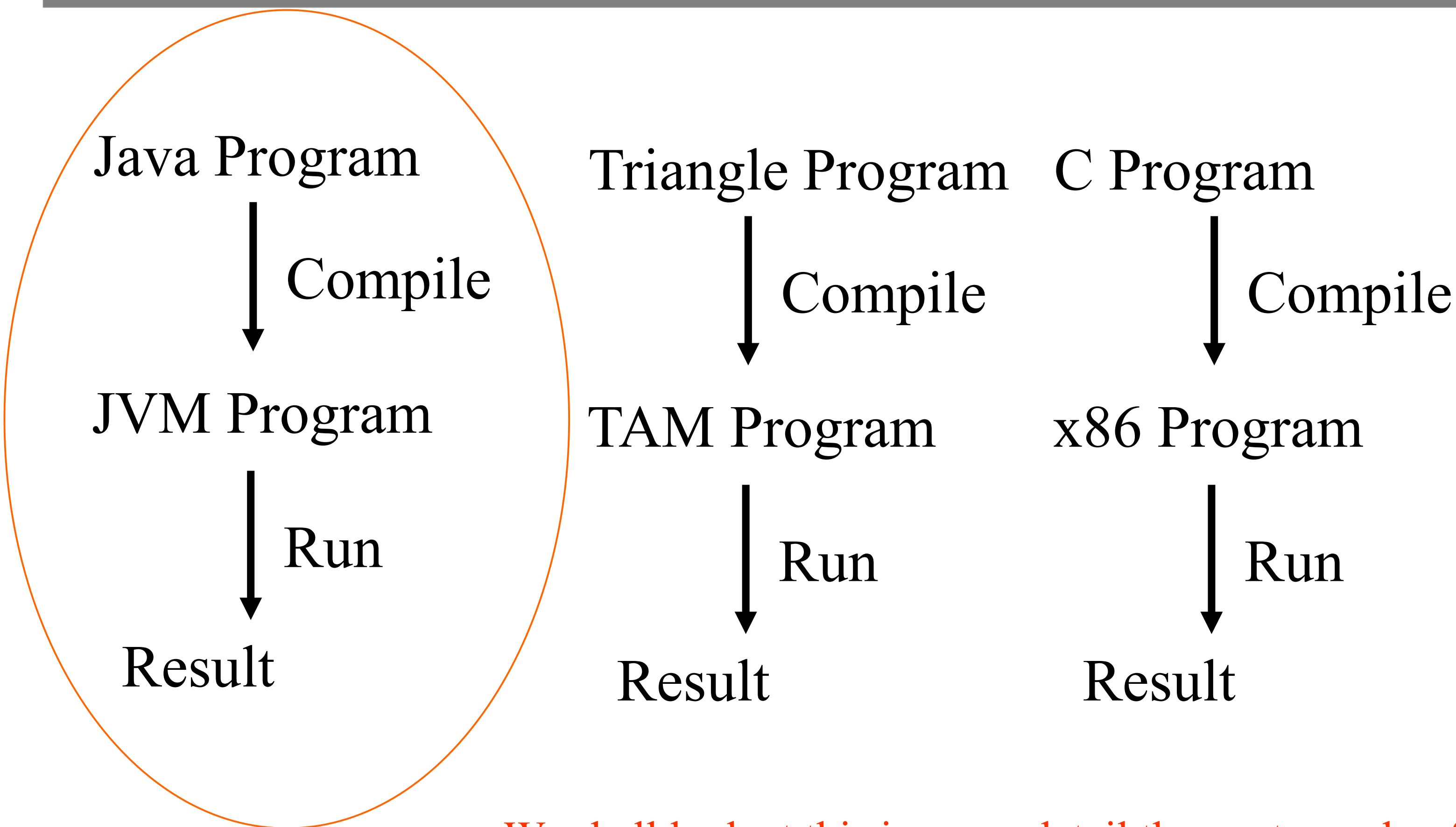
Figure 10.3: A middle-end and its ILs simplify construction of a compiler suite that must support multiple source languages and multiple target architectures.

IL Advantages

- An IL simplifies development and testing of system components
  - simplify the pioneering and prototyping of news ideas
- An IL allows various system components to interoperate by facilitating access to information about the program
  - E.g. variable names and types, and source line numbers could be useful in the debugger
  - It allows components and tools to interface with other products
- An IL enables the crafting of a retargetable code generator, which greatly enhances its portability
  - Pascal: P-code
  - Ada: DIANA (Descriptive Intermediate Attributed Notation for Ada)
  - C: RTL
  - Java: JVM
  - C#: CIL
  - Python: Python Byte Code

Code Generation

> A compiler translates a program from a high-level language into an **equivalent** program in a low-level language.

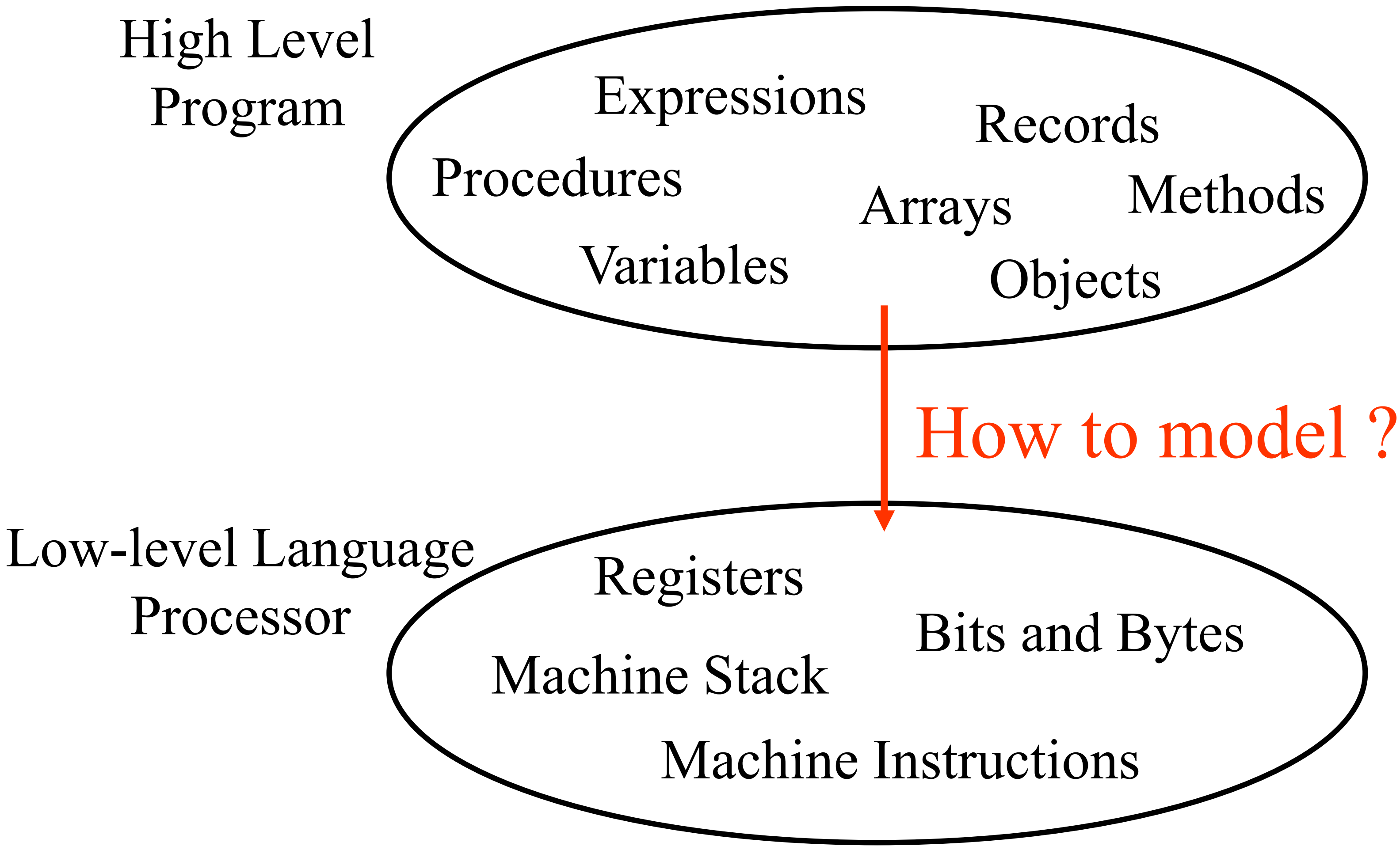


We shall look at this in more detail the next couple of lectures

Note that code generation is specific to the target, but we try to generalize

What are (some of) the issues

How to model high-level computational structures and data structures in terms of low-level memory and machine instructions.



Easy for Java (or Java like) on the JVM

Type	JVM designation
`boolean`	`Z`
`byte`	`B`
`double`	`D`
`float`	`F`
`int`	`I`
`long`	`J`
`short`	`S`
`void`	`V`
Reference type *t*	`L`*t*`;`
Array of type *a*	`[`*a*

Figure 10.4: Java types and their designation in the JVM. All of the integer-valued types are signed. For reference types, *t* is a fully qualified class name. For array types, *a* can be a primitive, reference, or array type.

For other Languages on the JVM some thoughts
Are needed on a suitable mapping

The JVM

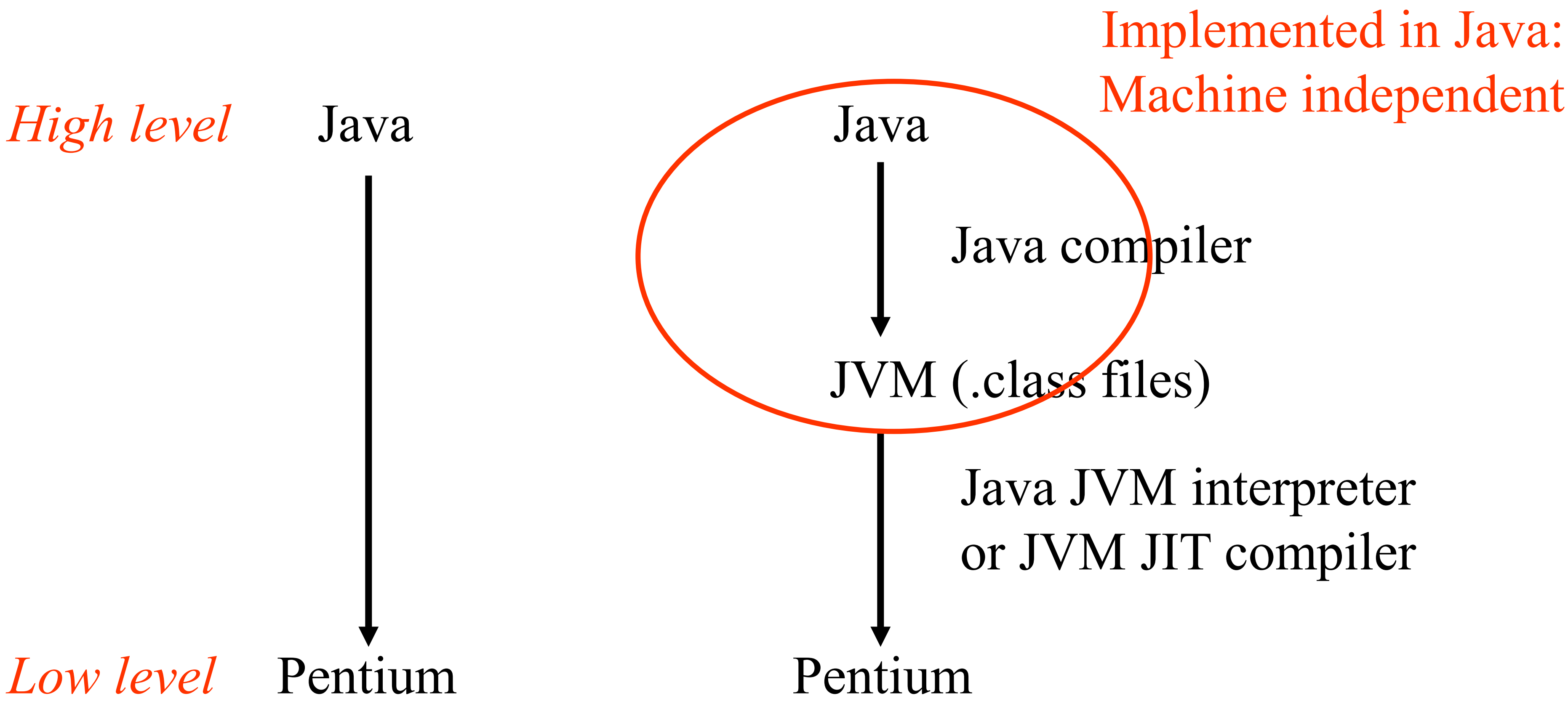
We now look at the JVM as an example of a real-world runtime system for a modern object-oriented programming language.

The material in this lecture is interesting because:

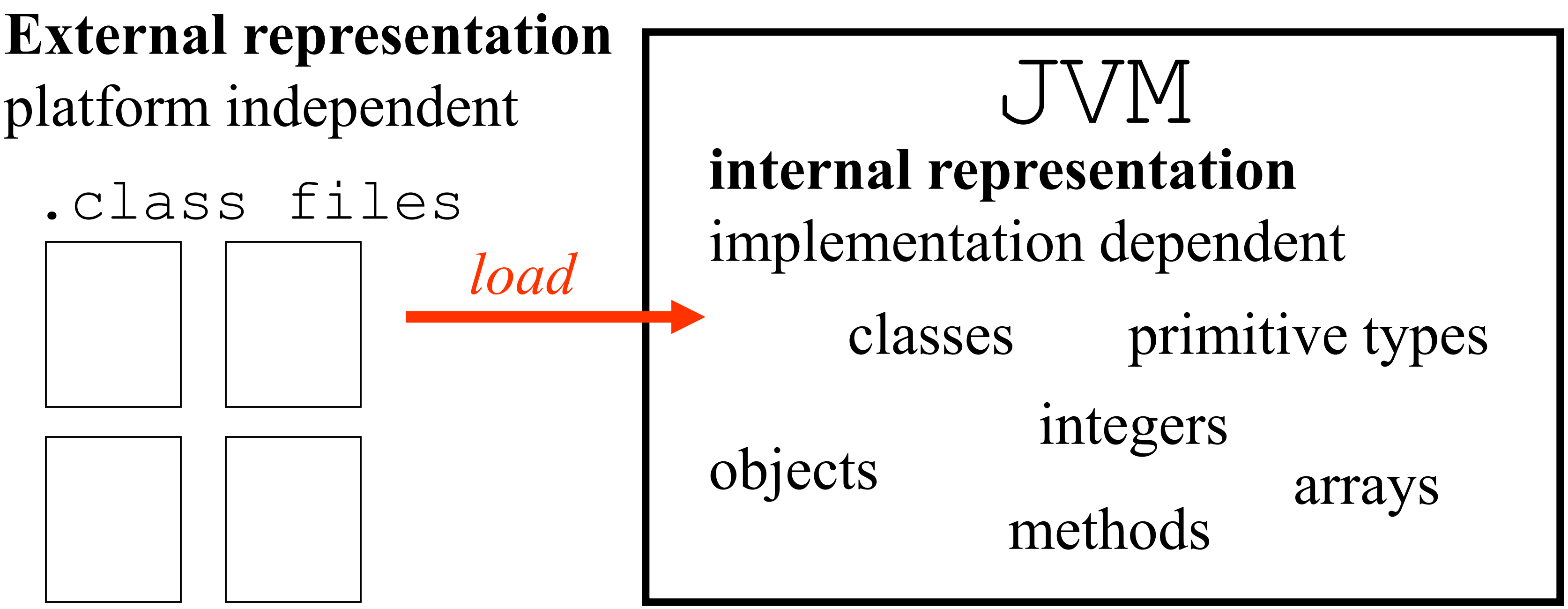
1) it will help understand some things about the JVM
2) JVM is probably the most common and widely used VM in the world.
3) You'll get a better idea what a real VM looks like.
4) You may choose the JVM as a target for your own compiler

Abstract Machines

An abstract machine implements an intermediate language "in between" the high-level language (e.g. Java) and the low-level hardware (e.g. Pentium)



Class Files and Class File Format



The **JVM is** an **abstract** machine in the true sense of the word.

The JVM spec. does not specify implementation details (can be dependent on target OS/platform, performance requirements etc.)

The JVM spec defines a machine independent “**class file format**” that all JVM implementations must support.

Java Virtual Machine

- Class files:
  - binary encodings of the data and instructions in a Java program
- Design principles
  - Compactness
    - Instructions in nearly zero-address form
- Class file contains:
  - Table of constants.
  - Tables describing the class
    - name, superclass, interfaces
    - attributes, constructor
  - Tables describing fields and methods
    - name, type/signature
    - attributes (private, public, etc)
  - The code for methods.

```
ClassFile {
      u4 magic; //always (0xCAFEBABE)
      u2 minor_version;
      u2 major_version;
      u2 constant_pool_count;
      cp_info constant_pool[constant_pool_count-1];
      u2 access_flags;
      u2 this_class;
      u2 super_class;
      u2 interfaces_count;
      u2 interfaces[interfaces_count];
      u2 fields_count;
      field_info fields[fields_count];
      u2 methods_count;
      method_info methods[methods_count];
      u2 attributes_count;
      attribute_info attributes[attributes_count];
   }
```

Data Types

JVM (and Java) distinguishes between two kinds of types:

Primitive types:

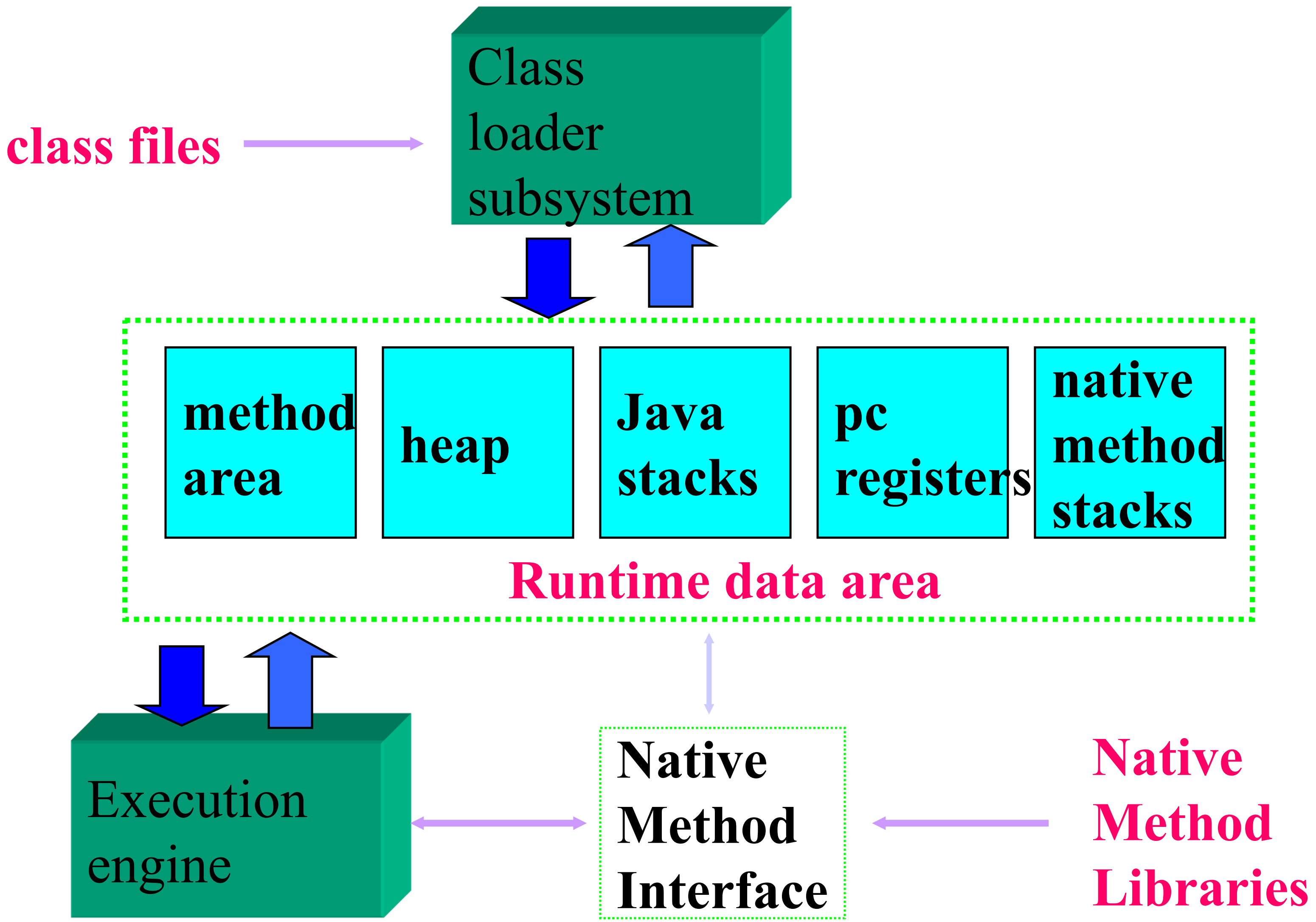
- boolean: `boolean`
- numeric integral: `byte, short, int, long, char`
- numeric floating point: `float, double`
- internal, for exception handling: `returnAddress`
    - `Used by jsr, jsr_w, ret instructions`

Reference types:

- class types
- array types
- interface types

Note: Primitive types are represented directly, reference types are represented indirectly (as pointers to array or class instances)

Internal Architecture of JVM



Class Loading

- Classes are loaded lazily when first accessed
  - Though some JVMs do eager loading
- Class name must match file name
- Super classes are loaded first (transitively)
- The bytecode is verified
- Static fields are allocated and given default values
- Static initializers are executed

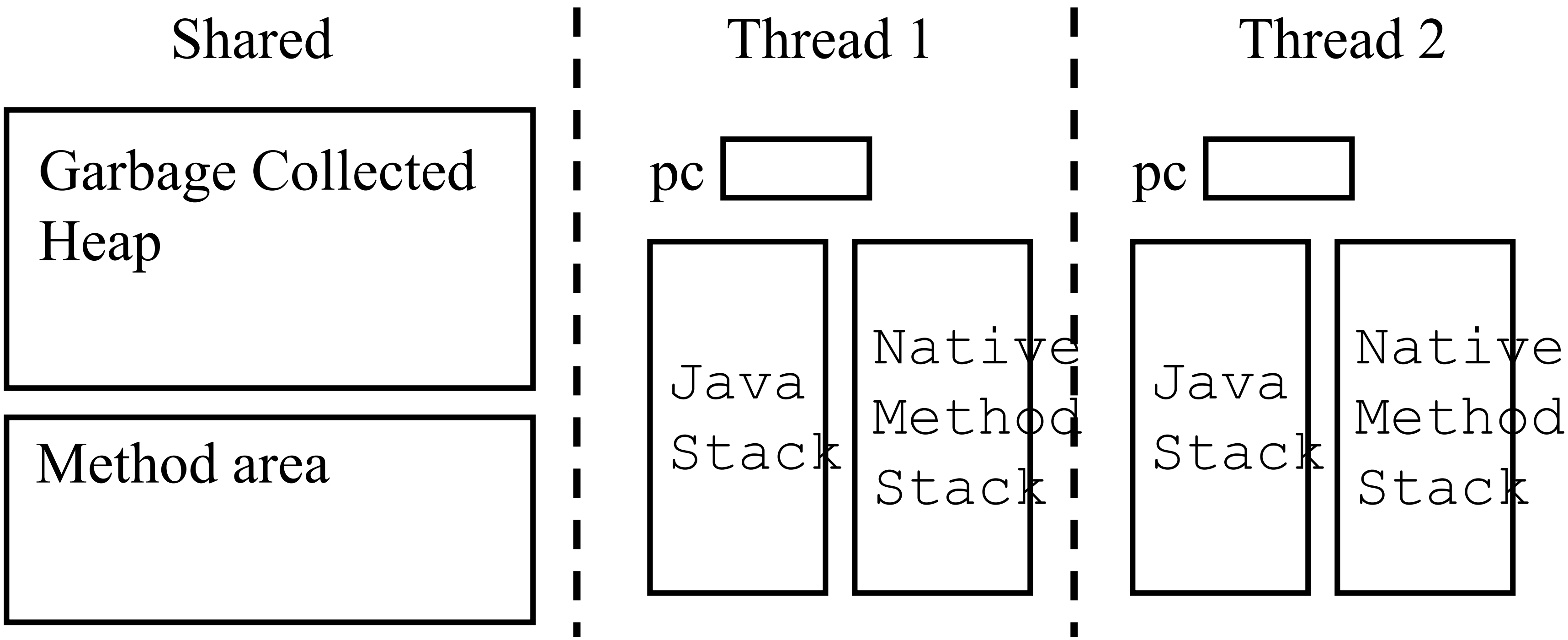
JVM: Runtime Data Areas

Besides OO concepts, JVM also supports multi-threading. Threads are directly supported by the JVM.

=> Two kinds of runtime data areas:

1) shared between all threads

2) private to a single thread



Java Stacks

JVM is a stack based machine

JVM instructions

- implicitly take arguments from the stack top
- put their result on the top of the stack

The stack is used to

- pass arguments to methods
- return result from a method
- store intermediate results in evaluating expressions
- store local variables

Expression Evaluation on a Stack Machine

Example 1: Computing `(a * b) + (1 - (c * 2))` on a stack machine.

```
LOAD a          //stack: a
LOAD b          //stack: a b
MULT            //stack: (a*b)
LOAD #1         //stack: (a*b) 1
LOAD c          //stack: (a*b) 1 c
LOAD #2         //stack: (a*b) 1 c 2
MULT            //stack: (a*b) 1 (c*2)
SUB             //stack: (a*b) (1-(c*2))
ADD             //stack: (a*b)+(1-(c*2))
```

Note the correspondence between the instructions and the expression written in postfix notation: `a b * 1 c 2 * - +`

Expression Evaluation on a Stack Machine

Example 2: Computing `(0 < n) && odd(n)` on a stack machine.

```
LOAD #0          //stack: 0
LOAD n           //stack: 0 n
LT               //stack: (0<n)
LOAD n           //stack: (0<n) n
CALL odd         //stack: (0<n) odd(n)
AND              //stack: (0<n)&&odd(n)
```

This example illustrates that calling functions/procedures fits in just as naturally with the stack machine evaluation model as operations that correspond to machine instructions.

In register machines this is much more complicated, because a stack must be created in memory for managing subroutine calls/returns.

JVM Interpreter

The core of a JVM interpreter is basically this:

```
do {
    byte opcode = fetch an opcode;
    switch (opcode) {
      case opCode1 :
            fetch operands for opCode1;
            execute action for opCode1;
            break;
      case opCode2 :
            fetch operands for opCode2;
            execute action for opCode2;
            break;
      case ...
} while (more to do)
```

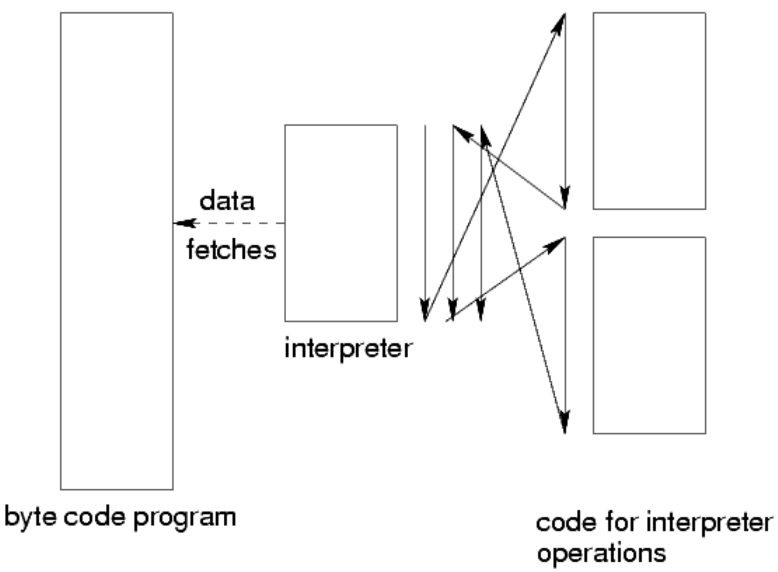
The JVM interpreter loop in the HVM

```
static int32 methodInterpreter(const
    MethodInfo* method, int32* fp) {
  unsigned char *method_code;
  int32* sp;
  const MethodInfo* methodInfo;

  start: method_code = (unsigned char *)
        pgm_read_pointer(&method->code, unsigned
         char**);
  sp = &fp[pgm_read_word(&method->maxLocals)
        +2];

  loop: while (1) {
    unsigned char code = pgm_read_byte(
          method_code);
    switch (code) {
    case ICONST_0_OPCODE:
    //ICONST_X Java Bytecodes
    case ICONST_5_OPCODE:
      *sp++ = code - ICONST_0_OPCODE;
      method_code++;
      continue;
    case FCONST_0_OPCODE:
    //Remaining Java Bytecode impl...
    }
  }
}
```

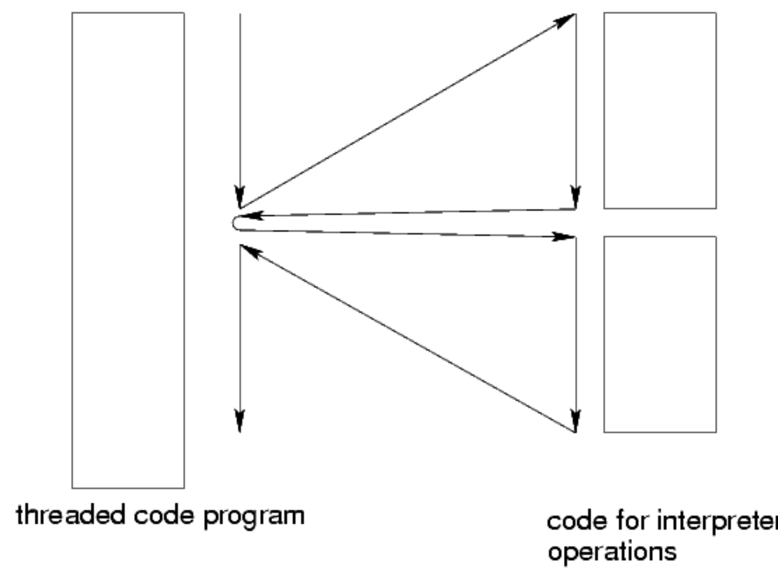
Threaded Code

```
Switch-Cased statement often translated into a jump table
Indexing through a jump table is expensive.
Idea: Use the address of the code for an operation as the
opcode for that operation.
```

byte code:



threaded code:



Control flow behavior:

[based on: James R. Bell. Threaded Code. *Communications of the ACM*, vol. 16 no. 6, June 1973, pp. 370–372]

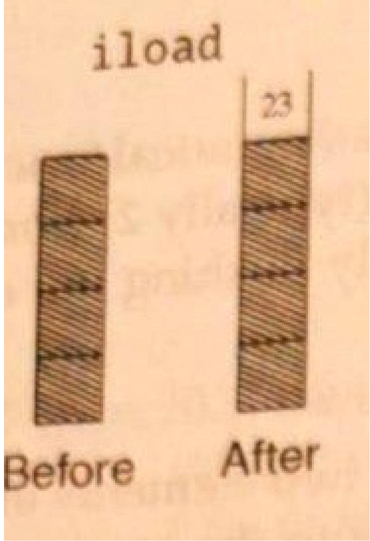
Instruction-set: typed instructions!

JVM instructions are explicitly typed: different opCodes for instructions for integers, floats, arrays and reference types.

This is reflected by a naming convention in the first letter of the opCode mnemonics:

Example: different types of "load" instructions

`iload`	integer load
`lload`	long load
`fload`	float load
`dload`	double load
`aload`	reference-type load



Instruction set: kinds of operands

JVM instructions have three kinds of operands:

- from the top of the operand stack
- from the bytes following the opCode
- part of the opCode

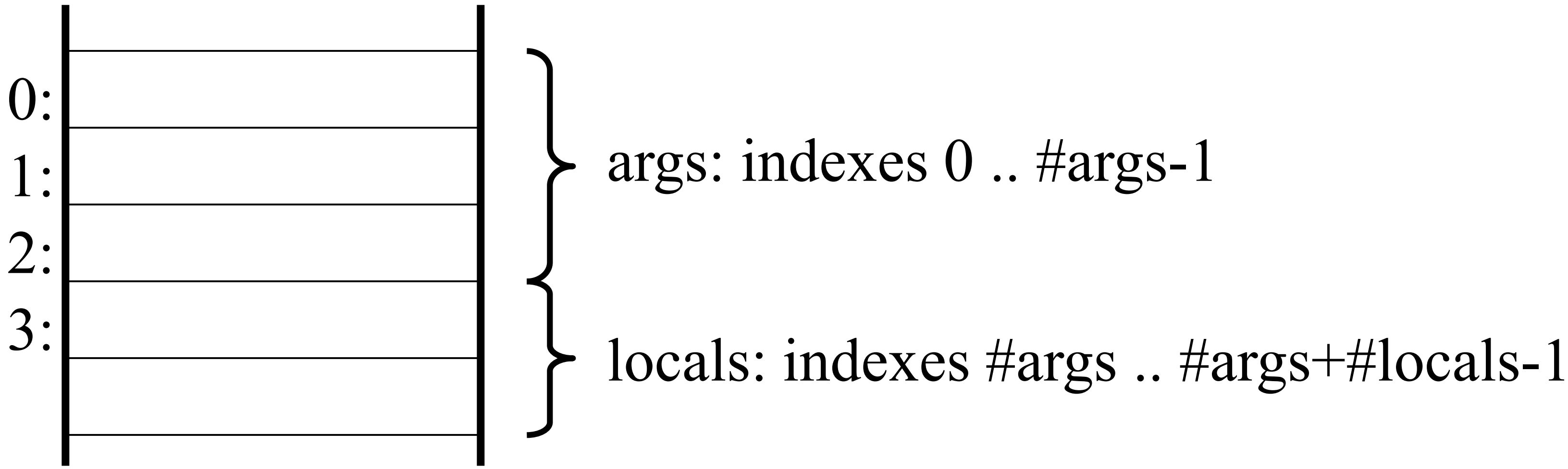
One instructions may have different "forms" supporting different kinds of operands.

Example: different forms of "iload".

Assembly code	Binary instruction code layout		
`iload_0`	26		
`iload_1`	27		
`iload_2`	28		
`iload_3`	29		
`iload` *n*	21	*n*	
`wide iload` *n*	196	21	*n*

Instruction-set: accessing arguments and locals

arguments and locals area inside a stack frame



Instruction examples:

iload_1	istore_1
iload_3	astore_1
aload 5	fstore_3
aload_0	

- A load instruction: loads something from the args/locals area to the top of the operand stack.
- A store instruction takes something from the top of the operand stack and stores it in the argument/local area

Instruction-set: non-local memory access

In the JVM, the contents of different "kinds" of memory can be accessed by different kinds of instructions.

accessing locals and arguments: `load` and `store` instructions

accessing fields in objects: `getfield, putfield`

accessing static fields: `getstatic, putstatic`

Note: static fields are a lot like global variables. They are allocated in the "method area" where also code for methods and representations for classes are stored.

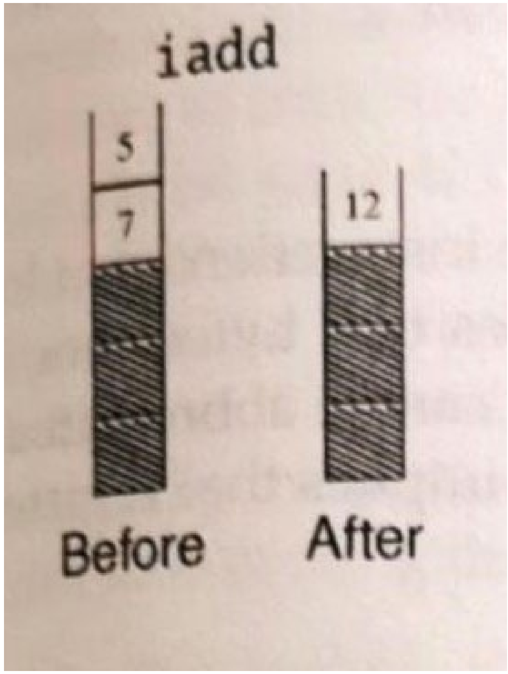
Q: what memory area are getfield and putfield accessing?

Instruction-set: operations on numbers

Arithmetic

add: `iadd, ladd, fadd, dadd`
subtract: `isub, lsub, fsub, dsub`
multiply: `imul, lmul, fmul, dmul`

...



Conversion

`i2l, i2f, i2d`
`l2f, l2d, f2s`

`f2i, d2i, …`

Instruction-set …

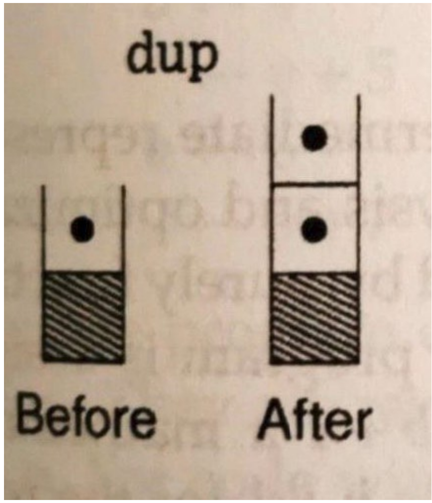
Operand stack manipulation

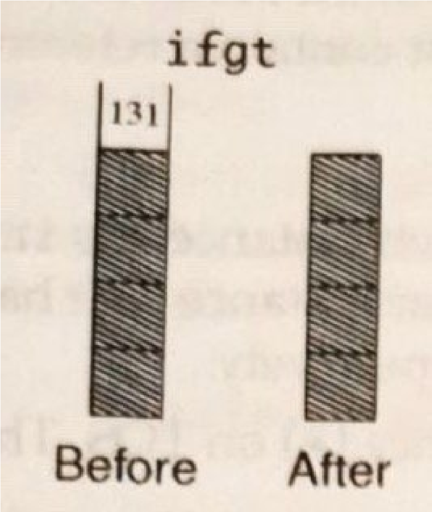
`pop, pop2, dup, dup2, dup_x1, swap, …`

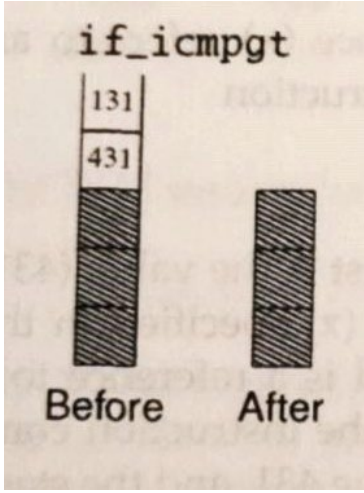
Control transfer

Unconditional : `goto, goto_w, jsr, ret, …`

Conditional: `ifeq, iflt, ifgt, …`







Instruction-set …

Method invocation:

`invokevirtual`

usual instruction for calling a method on an object.

`invokeinterface`

same as invokevirtual, but used when the called method is declared in an interface. (requires different kind of method lookup)

`invokespecial`

for calling things such as constructors. These are not dynamically dispatched (also known as `invokenonvirtual`)

`invokestatic`

for calling methods that have the "static" modifier (these methods "belong" to a class, rather an object)

Returning from methods:

`return, ireturn, lreturn, areturn, freturn,` …

Instruction-set: Heap Memory Allocation

Create new class instance (object):

```
new
```

Create new array:

```
newarray
  for creating arrays of primitive types.
anewarray, multianewarray
  for arrays of reference types
```

Example

As an example on the JVM, we will take a look at the compiled code of the following simple Java class declaration.

```
class Factorial {

    int fac(int n) {
        int result = 1;
        for (int i=2; i<n; i++) {
            result = result * i;
        }
        return result;
    }
}
```

Compiling and Disassembling

```
% javac Factorial.java
% javap -c -verbose Factorial
Compiled from Factorial.java
public class Factorial extends java.lang.Object {
    public Factorial();
        /* Stack=1, Locals=1, Args_size=1 */
    public int fac(int);
        /* Stack=2, Locals=4, Args_size=2 */
}

Method Factorial()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return
```

Compiling and Disassembling ...

```
                    // address: 0    1 2       3
Method int fac(int) // stack: this n result i
  0 iconst_1        // stack: this n result i 1
  1 istore_2        // stack: this n result i
  2 iconst_2        // stack: this n result i 2
  3 istore_3        // stack: this n result i
  4 goto 14
  7 iload_2         // stack: this n result i result
  8 iload_3         // stack: this n result i result i
  9 imul            // stack: this n result i result i
  10 istore_2
  11 iinc 3 1
  14 iload_3        // stack: this n result i i
  15 iload_1        // stack: this n result i i n
  16 if_icmple 7    // stack: this n result i
  19 iload_2        // stack: this n result i result
  20 ireturn
```

JASMIN

- JASMIN is an assembler for the JVM
  - Takes an ASCII description of a Java class
  - Input written in a simple assembler like syntax
    - Using the JVM instruction set
  - Outputs binary class file
  - Suitable for loading by the JVM

- Running JASMIN
  - `jasmin myfile.j`
- Produces a .class file with the name specified by the .class directive in myfile.j

Example: out.j

```
.class public out
.super java/lang/Object

.method public <init>()V
        aload_0
        invokespecial java/lang/Object/<init>()V
        return
.end method

.method public static main([Ljava/lang/String;)V
        .limit stack 2

        getstatic java/lang/System/out Ljava/io/PrintStream;
        ldc "Hello World"
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

        return
.end method
```

The result: out.class

```
00000000h: CA FE BA BE 00 03 00 2D 00 1B 0C 00 17 00 1A 01 ; 杇瑣...-........
00000010h: 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 ; ..([Ljava/lang/S
00000020h: 74 72 69 6E 67 3B 29 56 01 00 10 6A 61 76 61 2F ; tring;)V...java/
00000030h: 6C 61 6E 67 2F 4F 62 6A 65 63 74 01 00 06 3C 69 ; lang/Object...<i
00000040h: 6E 69 74 3E 07 00 03 0C 00 04 00 08 07 00 10 01 ; nit>............
00000050h: 00 03 28 29 56 07 00 13 01 00 04 43 6F 64 65 01 ; ..()V......Code.
00000060h: 00 04 6D 61 69 6E 09 00 09 00 01 01 00 0A 53 6F ; ..main........So
00000070h: 75 72 63 65 46 69 6C 65 01 00 05 6F 75 74 2E 6A ; urceFile...out.j
00000080h: 0C 00 11 00 19 01 00 13 6A 61 76 61 2F 69 6F 2F ; ........java/io/
00000090h: 50 72 69 6E 74 53 74 72 65 61 6D 01 00 07 70 72 ; PrintStream...pr
000000a0h: 69 6E 74 6C 6E 0A 00 05 00 06 01 00 10 6A 61 76 ; intln........jav
000000b0h: 61 2F 6C 61 6E 67 2F 53 79 73 74 65 6D 01 00 0B ; a/lang/System...
000000c0h: 48 65 6C 6C 6F 20 57 6F 72 6C 64 0A 00 07 00 0F ; Hello World.....
000000d0h: 08 00 14 01 00 03 6F 75 74 07 00 17 01 00 15 28 ; ......out......(
000000e0h: 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E ; Ljava/lang/Strin
000000f0h: 67 3B 29 56 01 00 15 4C 6A 61 76 61 2F 69 6F 2F ; g;)V...Ljava/io/
00000100h: 50 72 69 6E 74 53 74 72 65 61 6D 3B 00 21 00 18 ; PrintStream;.!..
00000110h: 00 05 00 00 00 00 00 02 00 01 00 04 00 08 00 01 ; ................
00000120h: 00 0A 00 00 00 11 00 01 00 01 00 00 00 05 2A B7 ; ..............*?
00000130h: 00 12 B1 00 00 00 00 00 09 00 0B 00 02 00 01 00 ; ..?............
00000140h: 0A 00 00 00 15 00 02 00 01 00 00 00 09 B2 00 0C ; ..............?.
00000150h: 12 16 B6 00 15 B1 00 00 00 00 00 01 00 0D 00 00 ; ..?.?..........
00000160h: 00 02 00 0E                                     ; ....
```

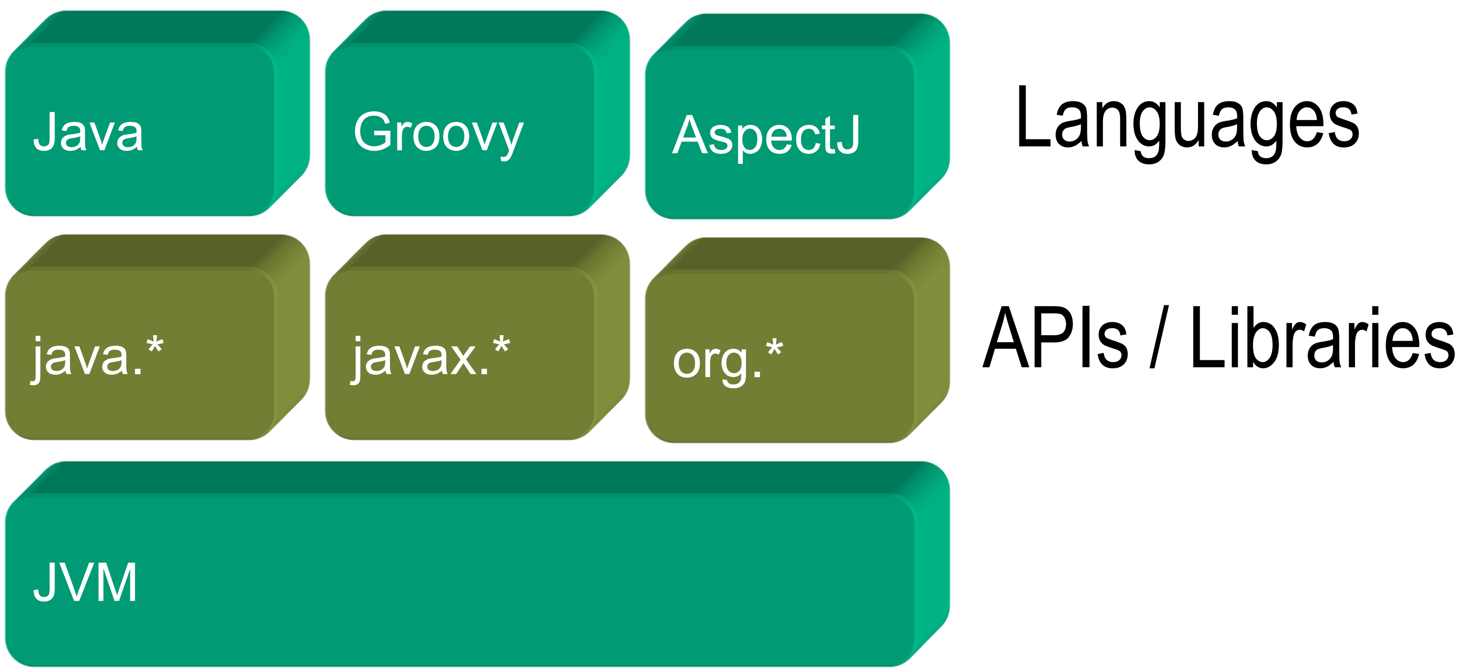
Jasmin file format

- Directives
  - .catch . Class .end .field .implements .interface .limit .line
  - .method .source .super .throws .var
- Instructions
  - JVM instructions: ldc, iinc bipush
- Labels
  - Any name followed by : - e.g. Foo:
  - Cannot start with = : . *
  - Labels can only be used within method definitions

The JVM as a target for different languages

When we talk about Java what do we mean?

- "Java" isn't just a language, *it is a platform*
- *The list of languages targeting the JVM is very long!*
  - *(Fortress), Scala, Clojure, Kotlin are currently very hot*
  - *http://en.wikipedia.org/wiki/List_of_JVM_languages*



Reusability

- Java has a lot of APIs and libraries
  - Core libraries (java[x].*)
  - Open source libraries
  - Third party commercial libraries
- What is it that we are *reusing* when we use these tools?
  - We are reusing the bytecode
  - We are reusing the fact that the JVM has a nice spec
- This means that we can innovate on top of this binary class file nonsense ☺

Not just <u>one</u> JVM, but a whole family

- JVM (J2EE & J2SE)
  - SUN Classis, SUN HotSpots, Oracle, IBM, BEA, …
- CVM, KVM (J2ME)
  - Small devices.
  - Reduces some VM features to fit resource-constrained devices.
- JCVM (Java Card)
  - Smart cards.
  - It has least VM features.

- And there are also lots of other JVMs
  - E.g. HVM (www.icelab.dk)

Java Platform & VM & Devices

Java Technology Targets a Broad Range of Devices

Hardware implementations of the JVM

Figure 6. aJ-100 package (larger than actual size).

Pause

s*t vs. s+t

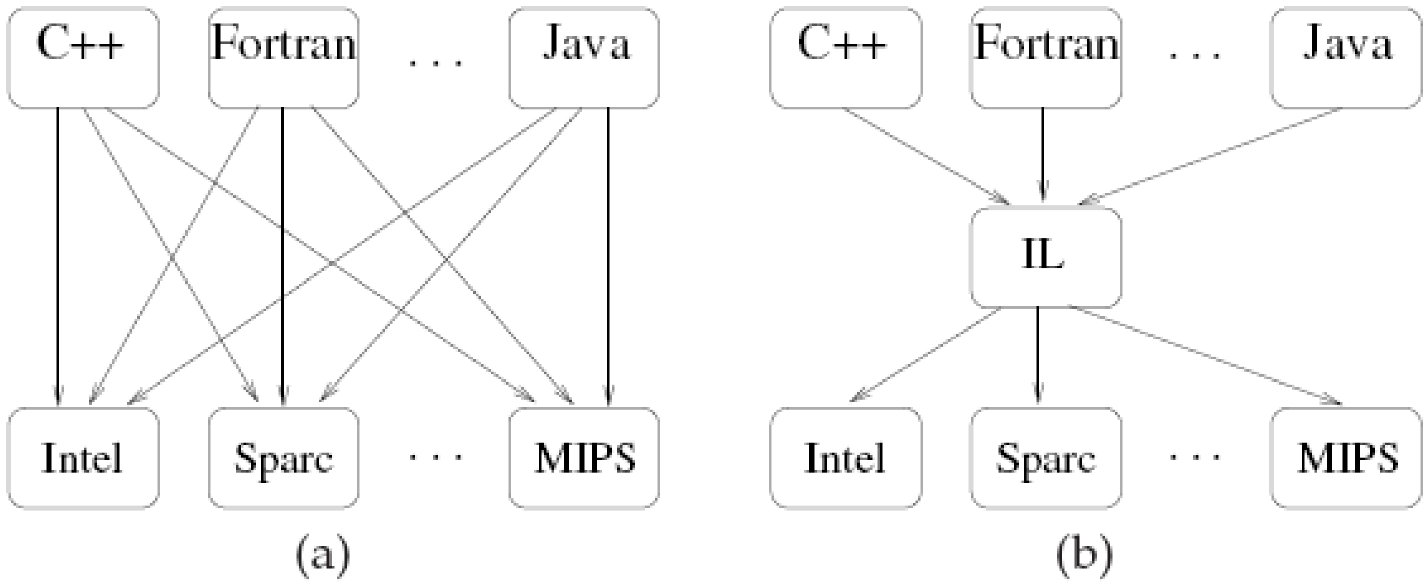
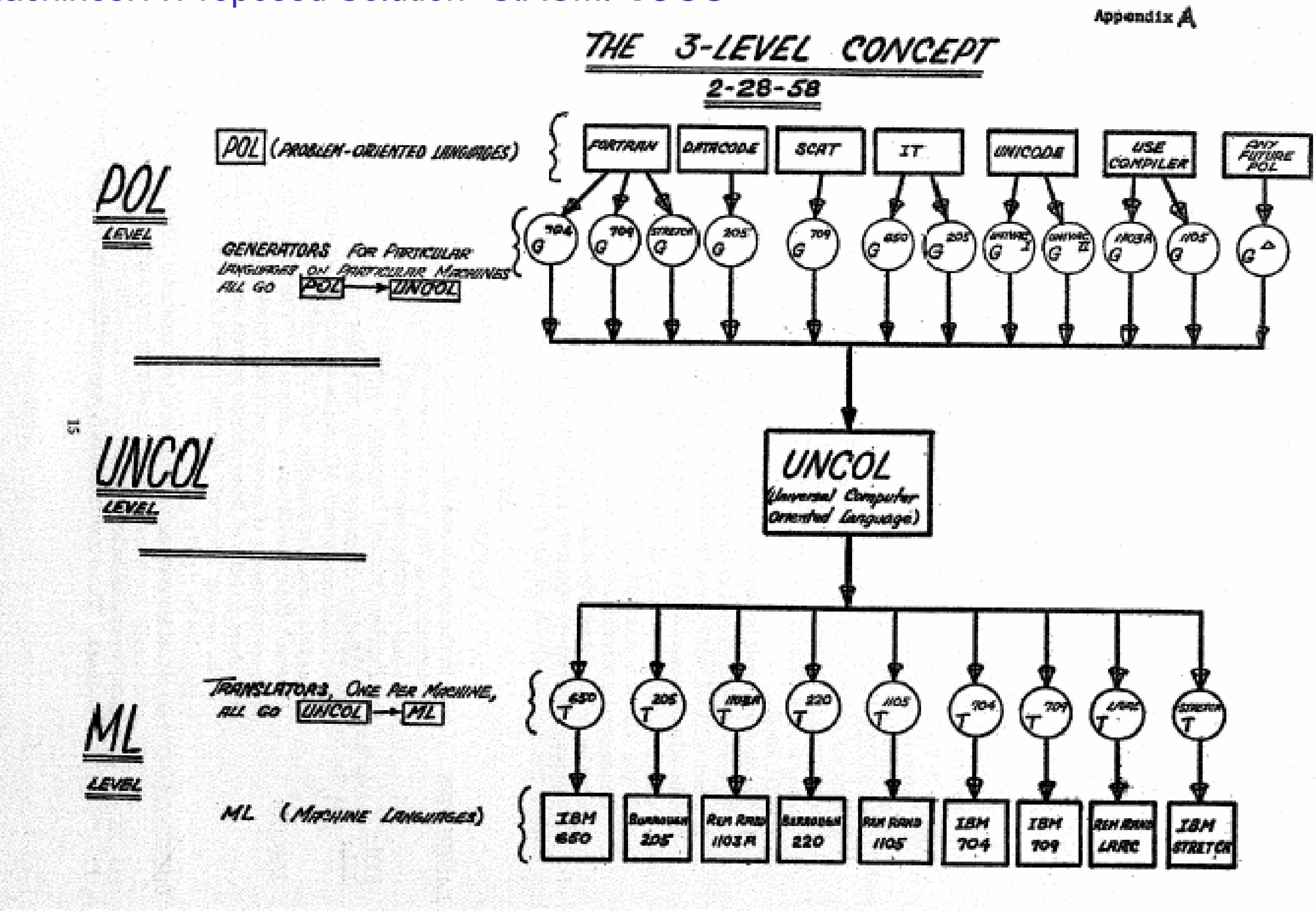


Figure 10.3: A middle-end and its ILs simplify construction of a compiler suite that must support multiple source languages and multiple target architectures.

The common intermediate format nirvana

- If we have `n` languages and need to have them running on `m` machines we need `m*n` compilers!
- If we have <u>one</u> common intermediate format we only need `n` front-ends and m back-ends, i.e. `m+n`
- ”Why haven’t you taught us about <u>the</u> common intermediate language?”

Strong et al. "The Problem of Programming Communication with Changing Machines: A Proposed Solution" C.ACM. **1958**



Quote

This concept is not particularly new or original. It has been discussed by many independent persons as long ago as 1954. It might not be difficult to prove that “this was well-known to Babbage,” so no effort has been made to give credit to the originator, if indeed there was a unique originator.

Interlanguage Working

- Smooth interoperability between components written in different programming languages is a dream with a long history

- Distinct from, more ambitious and more interesting than, UNCOL
  - The benefits accrue to users, not to compiler-writers!

- Interoperability is more important than performance, especially for niche languages, e.g.
  - For years we thought nobody used functional languages because they were too slow
  - But a bigger problem was that you couldn't really write programs that did useful things (graphics, guis, databases, sound, networking, crypto,...)
  - We didn't notice, because we never tried to write programs which did useful things...
  - However, with languages like F# and Scale, interoperating via .Net resp. the JVM, things are changing …

Interlanguage Working

- Bilateral or Multilateral?
- Unidirectional or bidirectional?
- How much can be mapped?
- Explicit or implicit or no marshalling?
- What happens to the languages?
  - All within the existing framework?
  - Extended?
  - Pragmas or comments or conventions?
- External tools required?
- Work required on both sides of an interface?

Calling C bilaterally

- All compilers for high-level languages have some way of calling C
  - Often just hard-wired primitives for implementing libraries
    - Extensibility by recompiling the runtime system ☹
  - Sometimes a more structured FFI
  - Typically implementation-specific

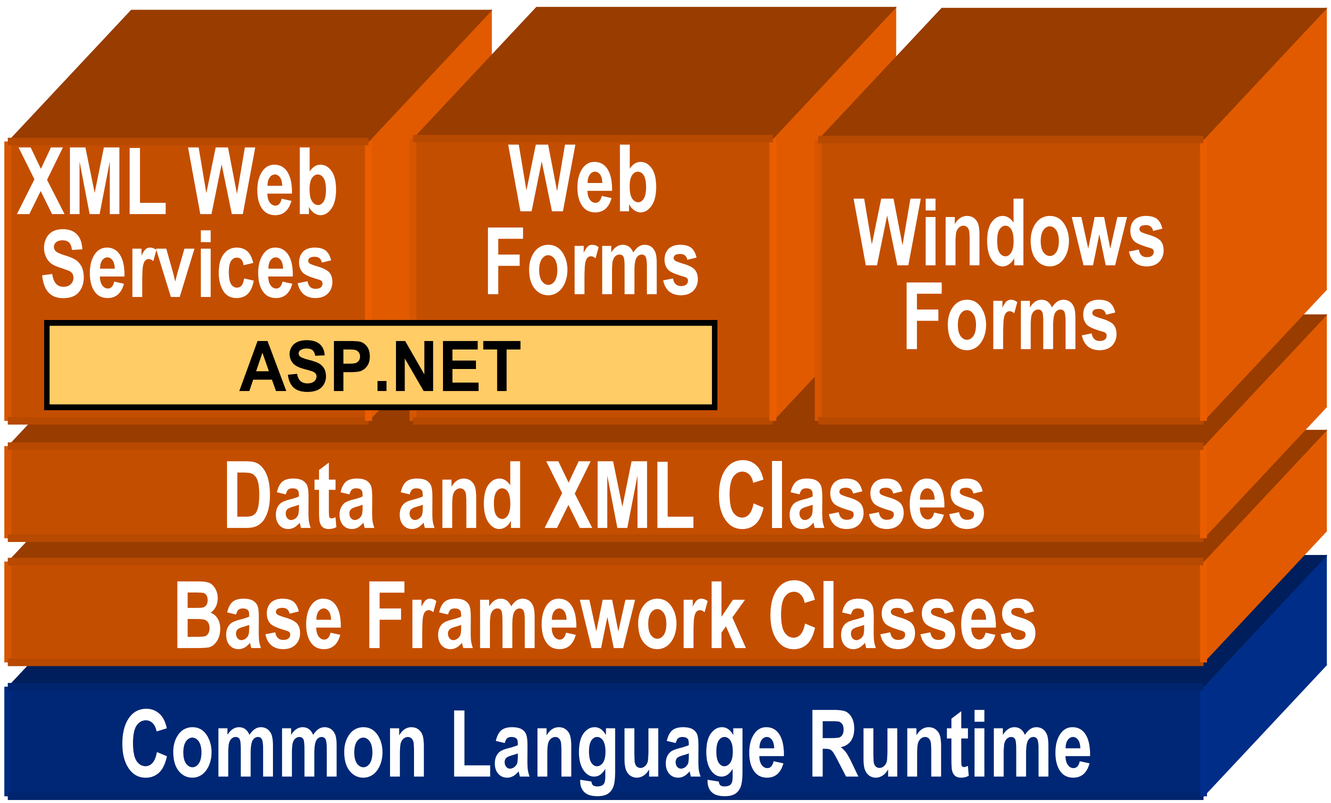
- Issues:
  - Data representation (31/32 bit ints, strings, record layout,...)
  - Calling conventions (registers, stack,..)
  - Storage management (especially copying collectors)

- It's a dirty job, but somebody's got to do it

Is there a better way?

- Well we saw the JVM earlier …
  - Most JVM support JNI
  - But this only works for calling from Java to C
  - Note HVM supports calling Java from C!
- And there are problems with languages which are not ”Java”-like
- What then? …

Common Programming Model - .NET



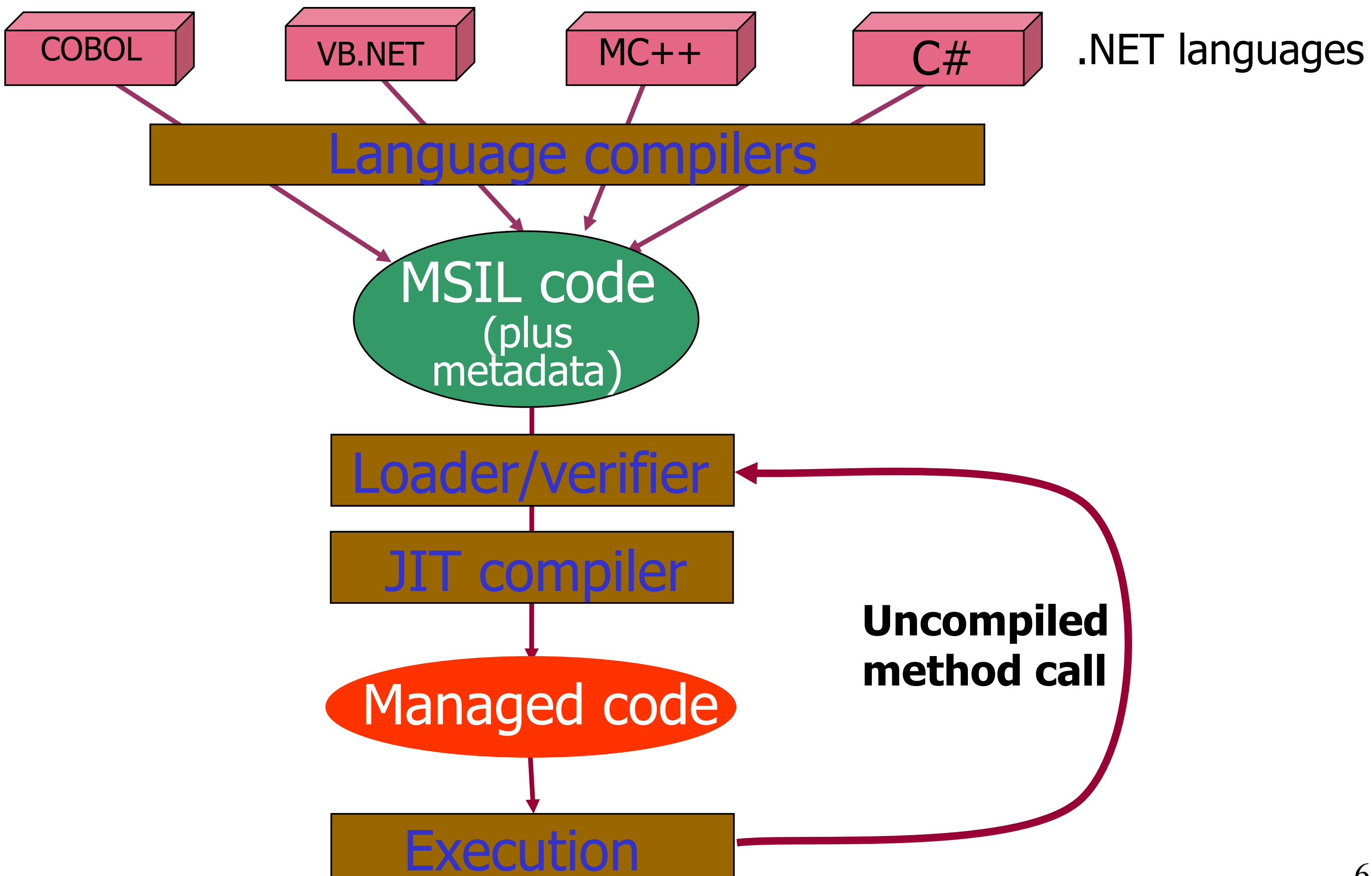
Overview of the CLI

- A common type system…
  …and a specification for language integration (**CLS**)
  - Execution engine with garbage collector and exception handling
  - Integral security system with verification
- A factored class library
  - A "modern" equivalent to the C runtime
- An intermediate language
  - CIL: Common Intermediate Language
- A file format
  - PE/COFF format, with extensions
  - An extensible metadata system
- Access to the underlying platform!

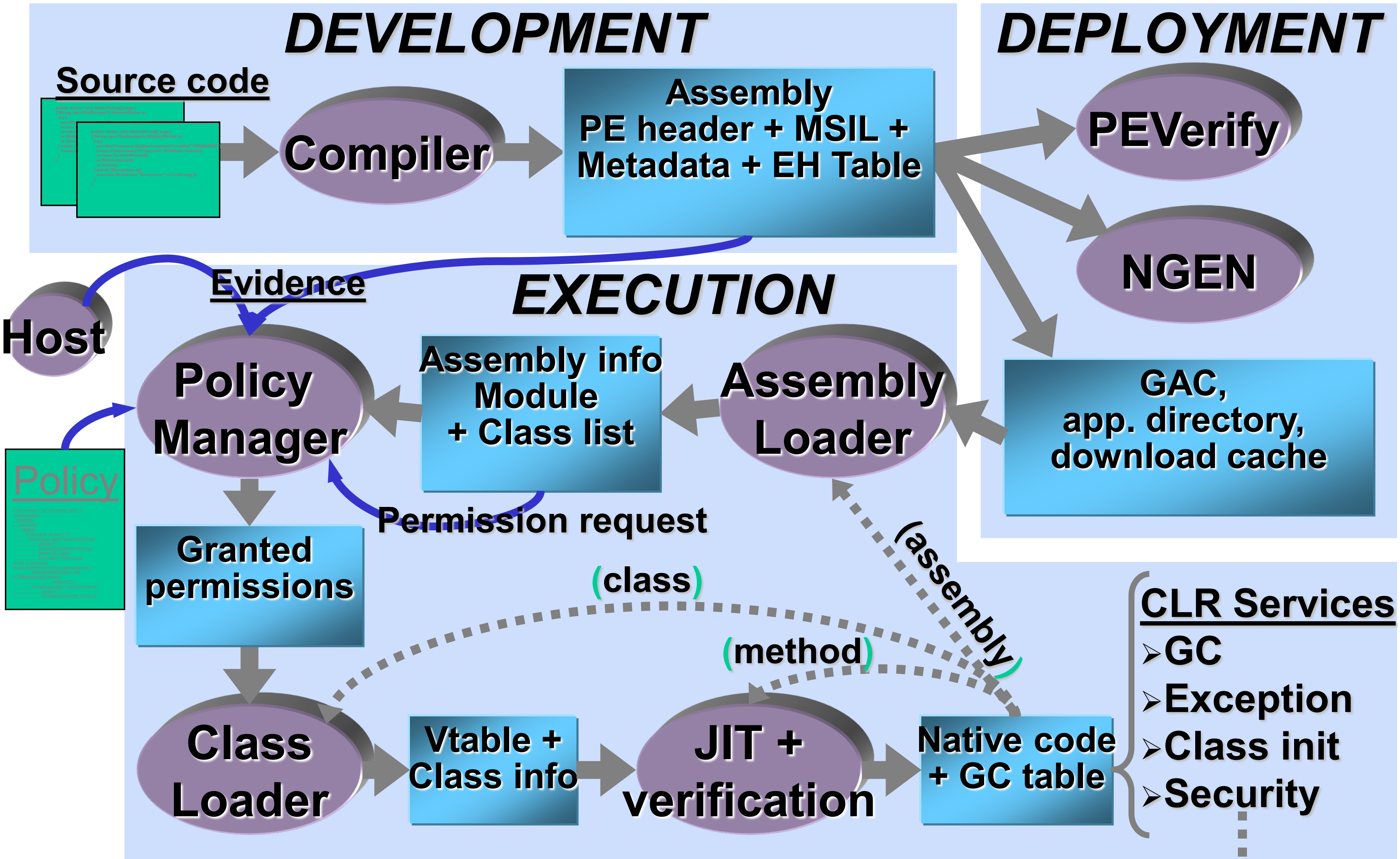
Terms to swallow

- CLI (Common Language Infrastructure)
- CLS (Common Language Specification)
- CTS (Common Type System)
- MSIL (Microsoft Intermediate Language)
  - CIL (Common Intermediate Language)
- CLR (Common Language Runtime)
- GAC (Global Assembly Cache)

Execution model



Managed Code Execution



What is the Common Language Runtime (CLR)?

- The CLR is the execution engine for .NET
- Responsible for key services:
  - Just-in-time compilation
  - heap management
  - garbage collection
  - exception handling
- Rich support for component software
- Language-neutral

The CLR Virtual Machine

- Stack-based, no registers
  - All operations produce/consume stack elements
  - Locals, incoming parameters live on stack
  - Stack is of arbitrary size; stack elements are "slots"
  - May or may not use real stack once JITted
- Core components
  - Instruction pointer (IP)
  - Evaluation stack
  - Array of local variables
  - Array of arguments
  - Method handle information
  - Local memory pool
  - Return state handle
  - Security descriptor

- Execution example

```
int add(int a, int b)
{
   int c = a + b;
   return c;
}
```

Offset	Instruction	Parameters
IL_0000	ldarg	0
IL_0001	ldarg	1
IL_0002	add	
IL_0003	stloc	0
IL_0004	ldloc	0
IL_0005	ret	

CIL Basics

- Data types
  - `void`
  - `bool`
  - `char`, `string`
  - `float32`, `float64`
  - [`unsigned`] `int8`, `int16`, `int32`, `int64`
  - `native` [`unsigned`] `int`: native-sized integer value
  - `object`: System.Object reference
  - Managed pointers, unmanaged pointers, method pointers(!)
- Names
  - All names must be assembly-qualified fully-resolved names
    - [*assembly*]*namespace*.*class*::*Method*
    - `[mscorlib]System.Object::WriteLine`

CIL Instructions

- Stack manipulation
  - `dup`: Duplicate top element of stack (pop, push, push)
  - `pop`: Remove top element of stack
  - `ldloc`, `ldloc.`*n*, `ldloc.s` *n*: Push local variable
  - `ldarg`, `ldarg.`*n*, `ldarg.s` *n*: Push method arg
    - “this” pointer arg 0 for instance methods
  - `ldfld` *type class::fieldname*: Push instance field
    - requires “this” pointer on top stack slot
  - `ldsfld` *type class::fieldname*: Push static field
  - `ldstr` *string*: Push constant string
  - `ldc.`<*type*> *n*, `ldc.`<*type*>`.`*n*: Push constant numeric
    - <type> is i4, i8, r4, r8

CIL Instructions

- Branching, control flow
  - `beq`, `bge`, `bgt`, `ble`, `blt`, `bne`, `br`, `brtrue`, `brfalse`
    - Branch target is label within code
  - `jmp` <*method*>: Immediate jump to method (goto, sort of)
  - `switch (`*t1, t2, ... tn*`)`: Table switch on value
  - `call` *retval Class::method(Type, ...)*: Call method
    - Assumes arguments on stack match method expectations
    - Instance methods require "this" on top
    - Arguments pushed in right-to-left order
  - `calli` *callsite-description*: Call method through pointer
  - `ret`: Return from method call
    - Return value top element on stack

CIL Instructions

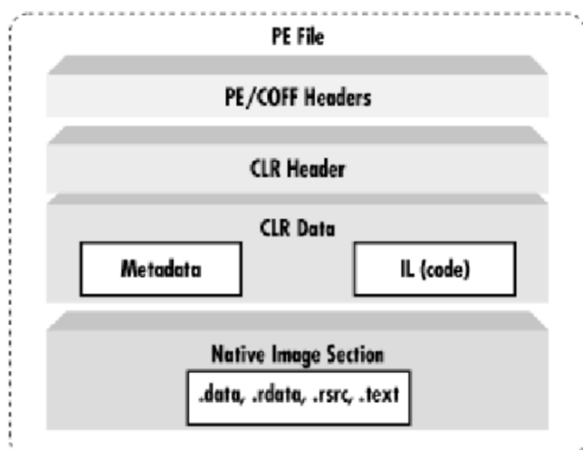
- Object model instructions
  - `newobj` *ctor*: Create instance using ctor method
  - `initobj` *type*: Create value type instance
  - `newarr` *type*: Create vector (zero-based, 1-dim array)
  - `ldelem`, `stelem`: Access vector elements
  - `isinst` *class*: Test cast (C# "is")
  - `castclass` *class*: Cast to type
  - `callvirt` *signature*: Call virtual method
    - Assumes "this" in slot 0--cannot be null
    - vtable lookup on object on *signature*
  - `box`, `unbox`: Convert value type to/from object instance

CIL Instructions

- Exception handling
  - `.try`: Defines guarded block
  - Dealing with exception
    - catch: Catch exception of specified type
    - fault: Handle exceptions but not normal exit
    - filter: Handle exception if filter succeeds
    - finally: Handle exception and normal exit
  - throw, rethrow: Put exception object into exception flow
  - leave: Exit guarded block

CIL assembler

- ILAsm (IL Assembly) closest to raw CIL
  - Assembly language
    - CIL opcodes and operands
    - Assembler directives
    - Intimately aware of the CLI (objects, interfaces, etc)
  - ilasm.exe (like JASMIN for Java/JVM)
  - Ships with FrameworkSDK, Rotor, along with a few samples
  - Creates a PE (portable executable) file (.exe or .dll)

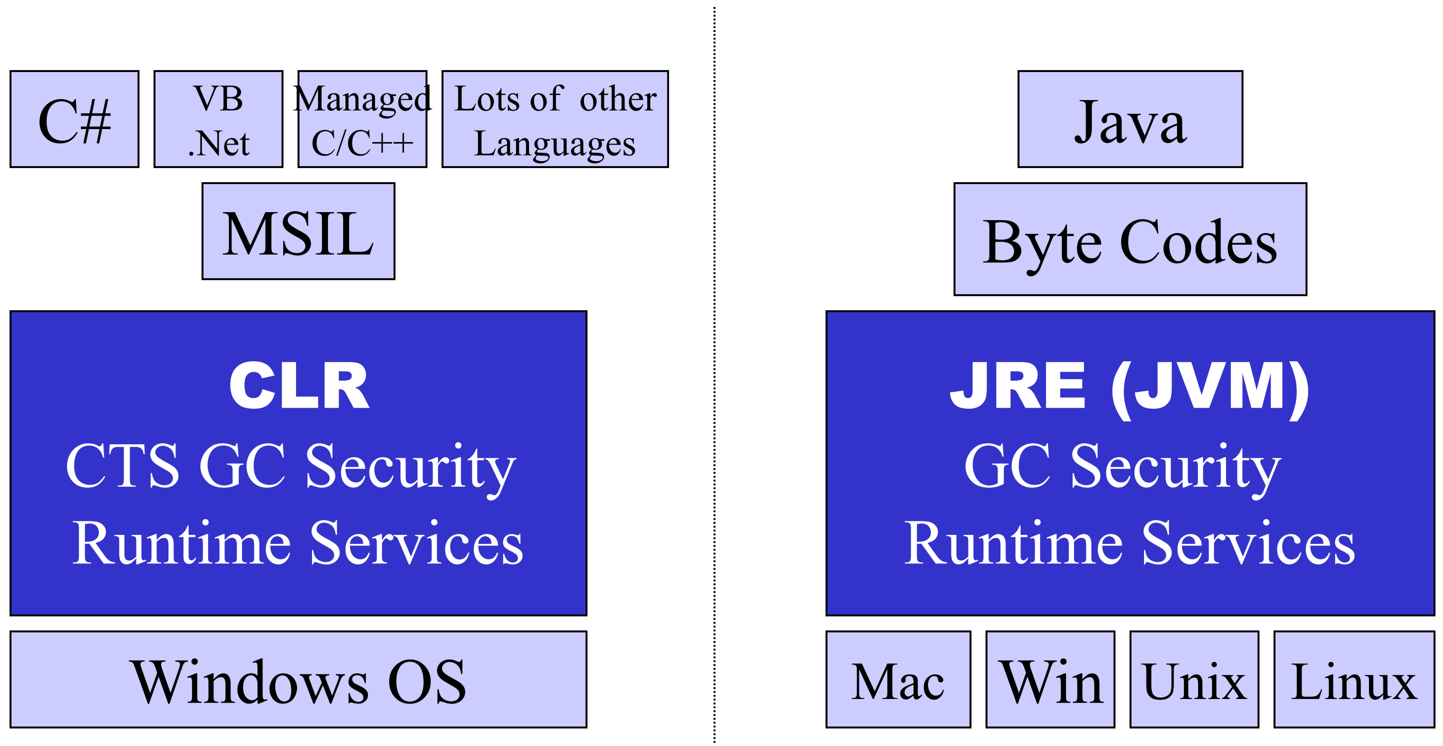


Example 1

- Hello, CIL!

```
.assembly extern mscorlib { }
.assembly Hello { }

.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
  .method private hidebysig static void Main() cil managed
  {
    .entrypoint
    .maxstack  1
    ldstr      "Hello, CIL!"
    call       void [mscorlib]System.Console::WriteLine(string)
    ret
  } // end of method App::Main
} // end of class App
```

CLR vs JVM



Both are 'middle layers' between an intermediate language & the underlying OS

Java Byte Code and MSIL

- Java byte code (or JVML) is the low-level language of the JVM.
- MSIL (or CIL or IL) is the low-level language of the .NET Common Language Runtime (CLR).
- Superficially, the two languages look very similar.

JVML:

```
iload 1
iload 2
iadd
istore 3
```

MSIL:

```
ldloc.1
ldloc.2
add
stloc.3
```

- One difference is that MSIL is designed only for JIT compilation.
- The generic add instruction would require an interpreter to track the data type of the top of stack element, which would be prohibitively expensive.

JVM vs. CLR

- JVM's storage locations are all 32-bit therefore e.g. a 64-bit int takes up two storage locations
- The CLR VM allows storage locations of different sizes

- In the JVM all pointers are put into one reference type
- CLR has several reference types e.g. valuetype reference and reference type

JVM vs. CLR

- CLR provides ”typeless” arithmetic instructions
- JVM has separate arithmetic instruction for each type (iadd, fadd, imul, fmul...)

- JVM requires manual overflow detection
- CLR allows user to be notified when overflows occur

- Java has a maximum of 64K branches (if...else)
- No limit of branches in CLR

JVM vs. CLR

- JVM distinguishes between invoking methods and interface (invokevirtual and invokeinterface)
- CLR makes no distinction

- CLR supports tail calls (iteration in Scheme)
- Must resort to tricks in order to make JVM discard stack frames

Alternatives to JVM and CLR

- C
  - (or C++ or Java or C# or ..)
- JavaScript
- WebAssembly
- GENERIC, GIMPLE and RTL for gcc
- Dalvik VM
- LLVM IR

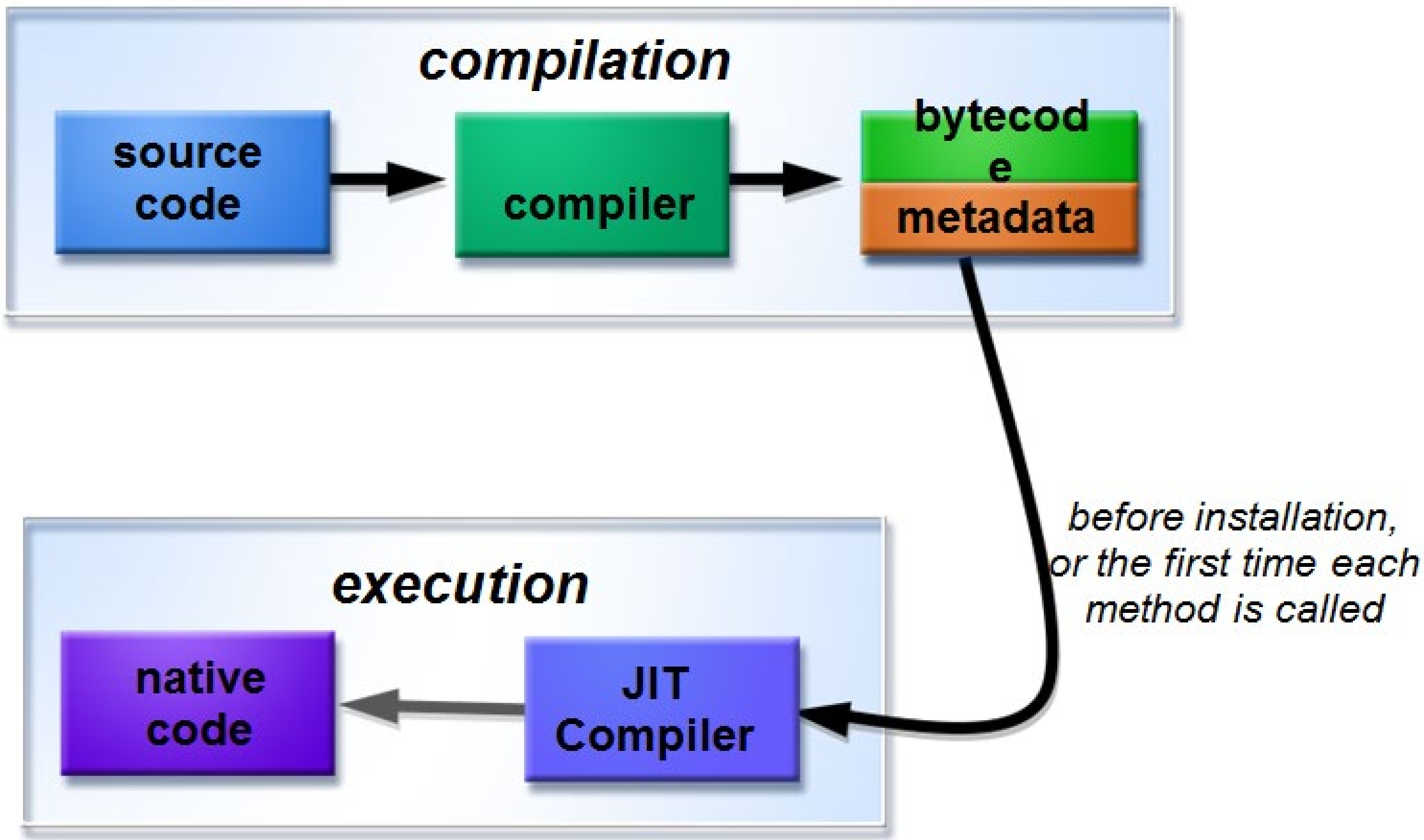
Comparison of Various VMs

Virtual machine	Machine model	Memory management	Code security	Interpreter	JIT	AOT	Shared libraries	Common Language Object Model	Dynamic typing
Android Runtime (ART)	register	automatic	Yes	No	No	Yes	?	No	No
BEAM (Erlang)	register	automatic	?	Yes	Yes	Yes	Yes	Yes	Yes
Common Language Runtime (CLR)	stack	automatic or manual	Yes	No	Yes	Yes	Yes	Yes	Yes
Dalvik	register	automatic	Yes	Yes	Yes	No	?	No	No
Dis (Inferno)	register	automatic	Yes	Yes	Yes	Yes	Yes	No	No
DotGNU Portable.NET	stack	automatic or manual	No	No	Yes	Yes	Yes	Yes	No
Java virtual machine (JVM)	stack	automatic	Yes	Yes	Yes	Yes	Yes	Yes	Yes[1]
JikesRVM	stack	automatic	No	No	Yes	No	?	No	No
LLVM	register	manual	No	Yes	Yes	Yes	Yes	Yes	No
Mono	stack	automatic or manual	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Parrot	register	automatic	No	Yes	No[2]	Yes	Yes	Yes	Yes
Squeak	stack	automatic	No	Yes	Yes	No	Yes	No	Yes

http://en.wikipedia.org/wiki/Comparison_of_application_virtual_machines

Just-In-Time Compilation

- JIT compilers in JRE (JVM) and .NET runtimes



Just-In-Time Compilation (cont)

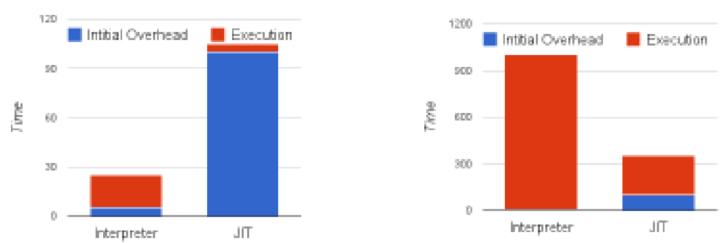
- At the time of code execution, the JIT compiler will compile some or all of it to native machine code for better performance.
  - Can be done per-file, per-function or even on any arbitrary code fragment (e.g. tracing JIT)
- The compiled code is cached and reused later without needing to be recompiled (unlike interpretation).

Java Virtual Machine - HotSpot

- Interpreter mode (-Xint)
- server mode (-server)
  - aggressive and complex optimizations
  - slow startup
  - fast execution
- client mode (-client)
  - less optimizations
  - fast startup
  - slower execution

Just-In-Time Overhead

JIT: 4x speedup, but 20x initial overhead



Matthew Arnold, Stephen Fink, David Grove, and Michael Hind, ACACES'06, 2006

What can you do with this in your project ?

- Consider generation code for a VM
  - JVM via JASMIN
  - CLR via ILASM
  - Some other VM
    - Python VM
    - Smalltalk VM
    - BEAM (Erlang)

Languages and Compilers (SProg og Oversættere)

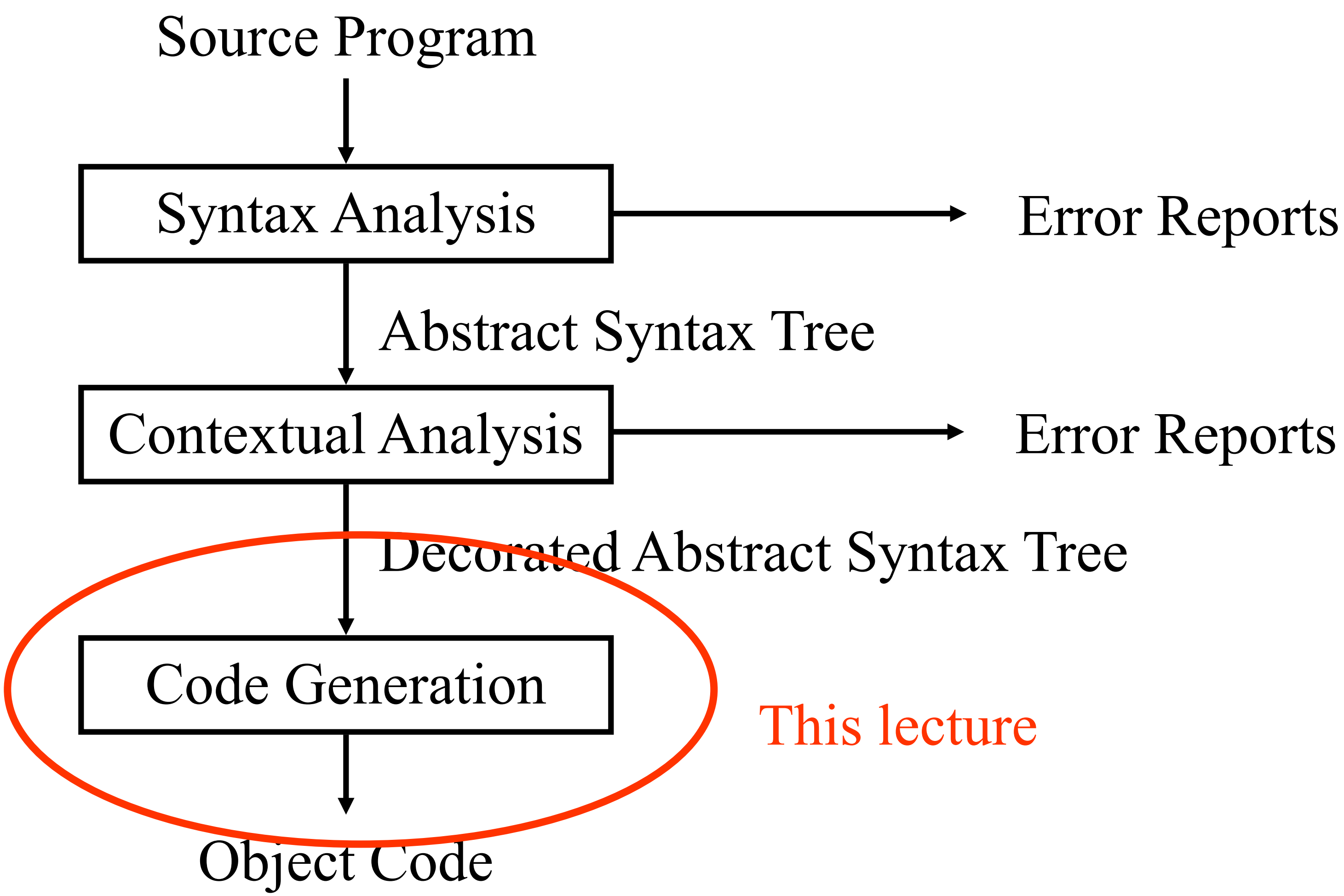
Lecture 16
Code Generation for the JVM

Bent Thomsen
Department of Computer Science
Aalborg University

Learning Goals

- Understand intermediate code generation
  - In particular IM generation for the JVM
- Understand the distinction between compile time and run time

The "Phases" of a Compiler

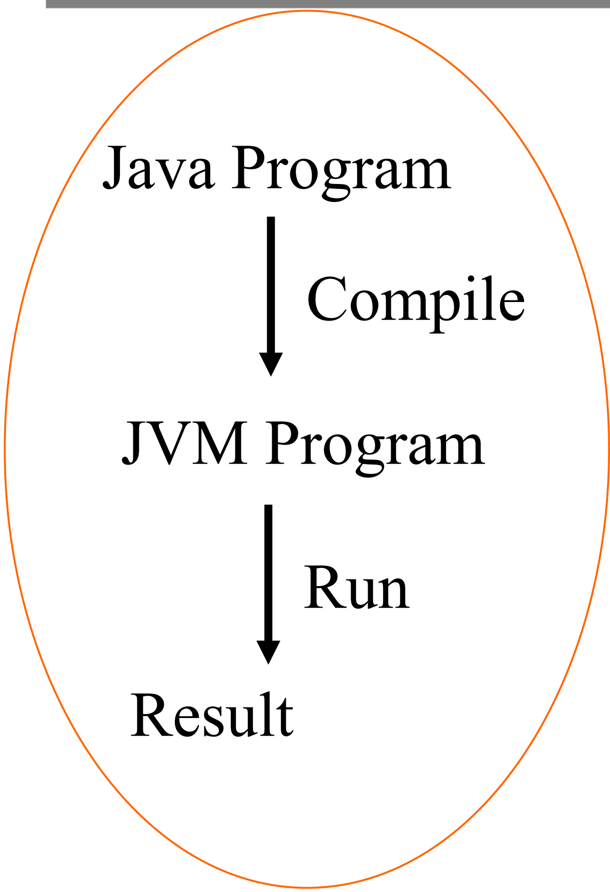


Programming Language specification

- A Language specification has (at least) three parts:
  - Syntax of the language: usually formal: EBNF
  - Contextual constraints:
    - scope rules (often written in English, but can be formal)
    - type rules (formal or informal)
  - **Semantics:**
    - **defined by the implementation**
    - **informal descriptions in English**
    - **formal using operational or denotational semantics**

Code Generation

A compiler translates a program from a high-level language into an **equivalent** program in a low-level language.



This lecture

Issues in Code Generation

- Code Selection:

  Deciding which sequence of target machine instructions will be used to implement each phrase in the source language.

- Storage Allocation

  Deciding the storage address for each variable in the source program. (static allocation, stack allocation etc.)

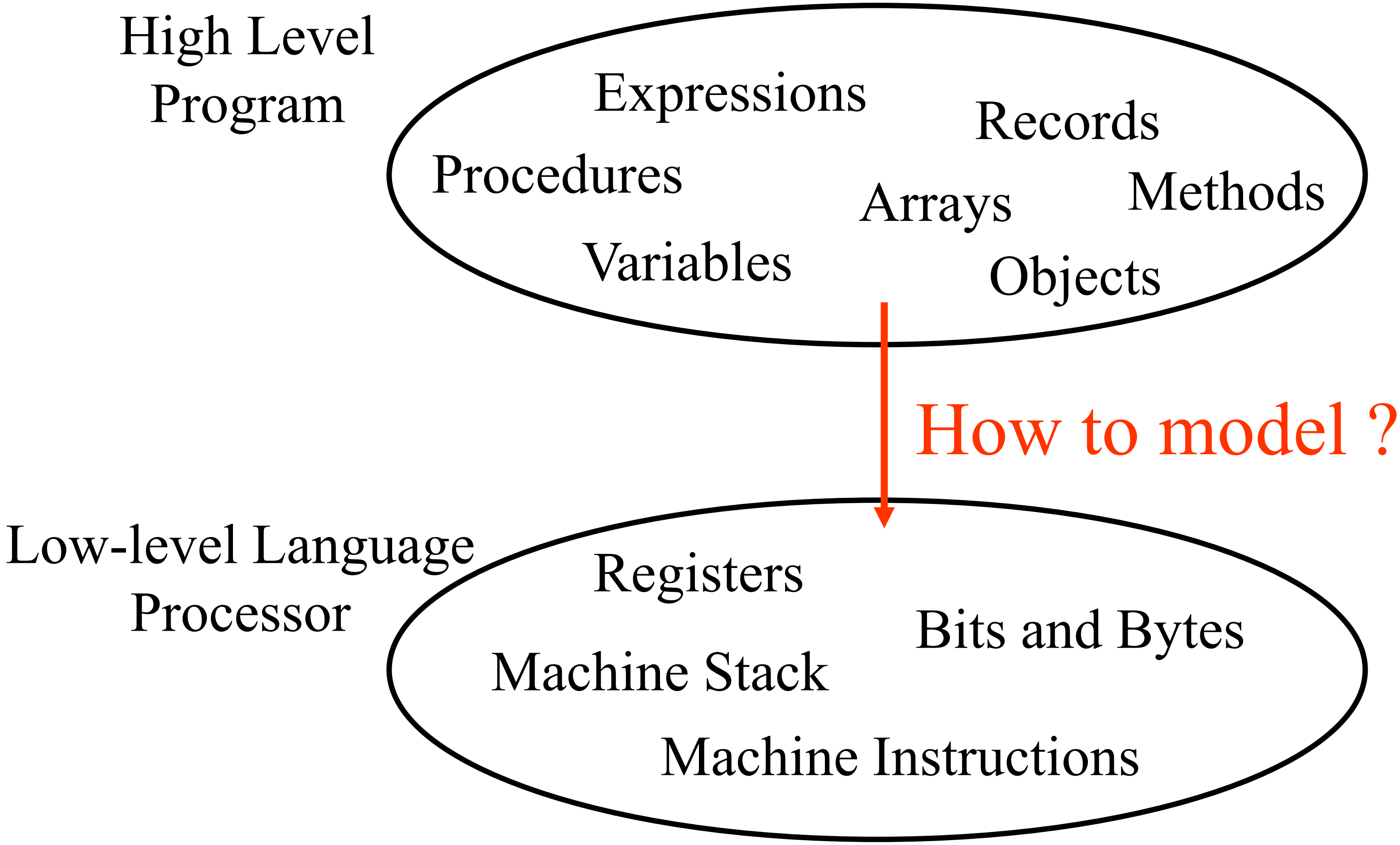
- Register Allocation (for register-based machines)

  How to use registers efficiently to store intermediate results.

This is not an issue for us because we look at generating code for the JVM
We will look at these issues in later lectures

What are (some of) the issues

How to model high-level computational structures and data structures in terms of low-level memory and machine instructions.



Easy for Java (or Java like) on the JVM

Type	JVM designation
`boolean`	`Z`
`byte`	`B`
`double`	`D`
`float`	`F`
`int`	`I`
`long`	`J`
`short`	`S`
`void`	`V`
Reference type *t*	`L`*t*`;`
Array of type *a*	`[`*a*

Figure 10.4: Java types and their designation in the JVM. All of the integer-valued types are signed. For reference types, *t* is a fully qualified class name. For array types, *a* can be a primitive, reference, or array type.

For other Languages on the JVM some thoughts
Are needed on a suitable mapping

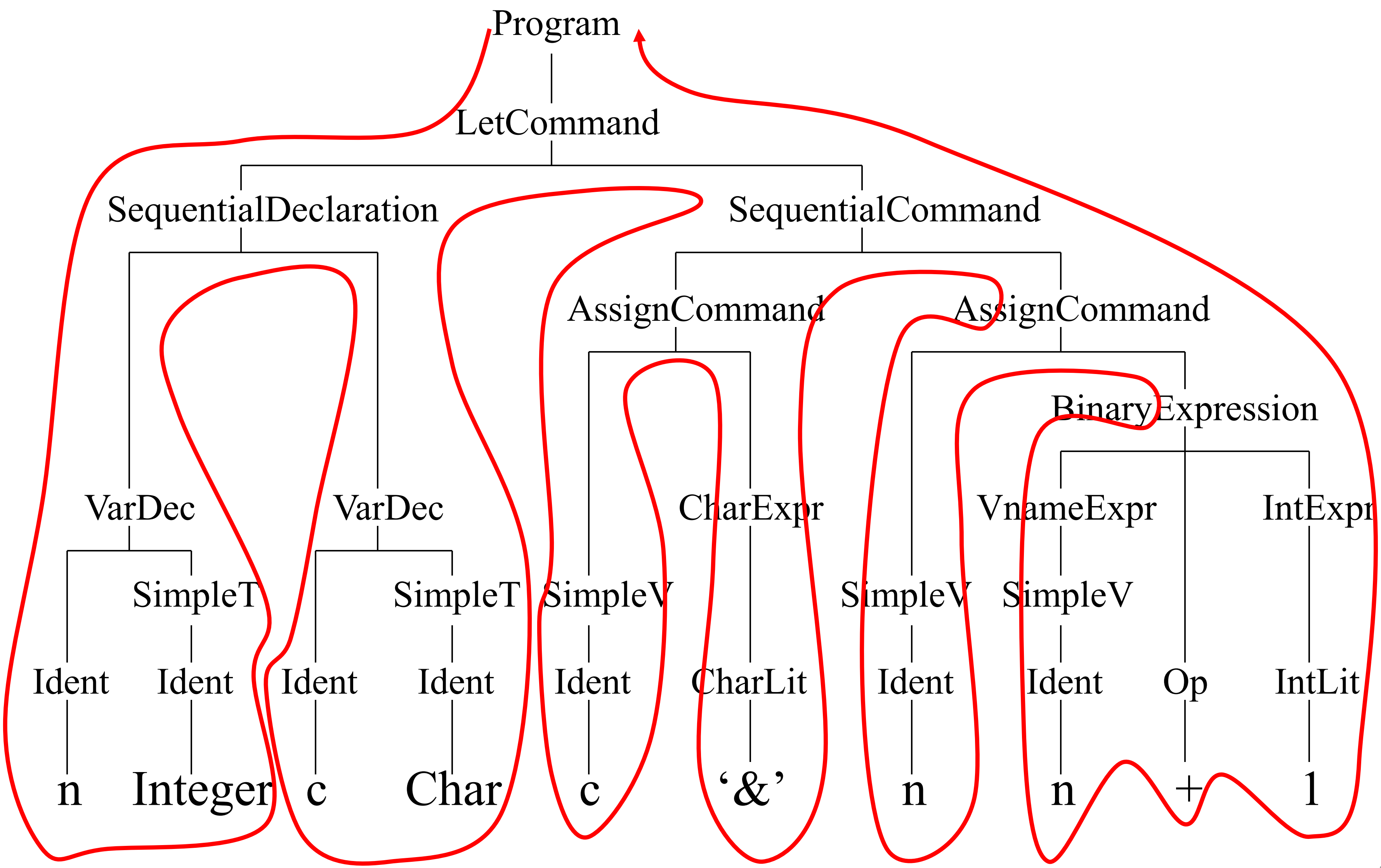
Code Gen: from AST to JVM

- Code Generation refers to translating the processed/decorated AST to an executable form
  - For Java, the target is the Java Virtual Machine
    - Translated to Bytecode
      - We talk about emitting Bytecode
    - Bytecode is "executed" by the JVM interpreter/JIT

- Terminology:
  - Compile time vs. Run time
    - Compile time AST traversal order
      - i.e. the order the compiler goes through the program
    - Run time code execution order
      - i.e. the order the thread of control goes through the program

Code Generation from AST

Code Generation in general follows a depth-first traversal of the AST.



CodeGenVisitor

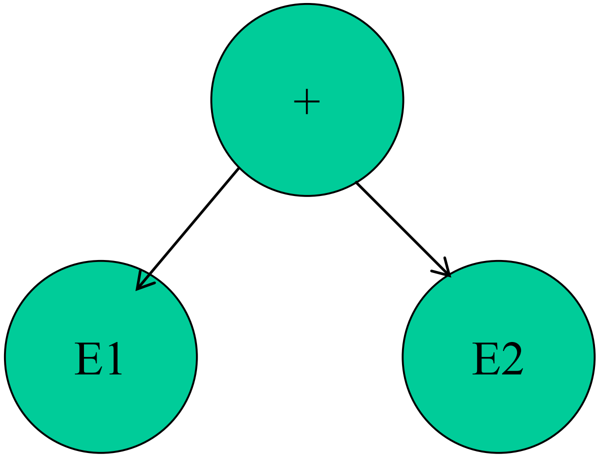
- We process the AST with a visitor
  - Could also use classic OO composit or a functional approach.
- Code Generation Visitors are usually divided into narrow-focus visitors for specific tasks
  - Class and method declarations
  - Statements
  - Expressions
  - Left-Hand Side processing
  - Method Signatures
- Others possible/needed in other languages

- Note Fischer et. Al. uses the reflexive visitor pattern

Code Emmision

- Generating the actual instructions is usually called emission
  - a CodeGenVisitor emits instructions
- Example:
  - MethodBodyVisitor.visit(Plus n)
    - visit(n.E1)
    - visit(n.E2)
    - emit("iadd\n")

```
/*   Visitor code for Marker (7)
procedure VISIT( Computing n )
    VISITCHILDREN( n )
    loc ← ALLOCLOCAL( )
    n.SETRESULTLOCAL( loc )
    call EMITOPERATION( n )
end
```



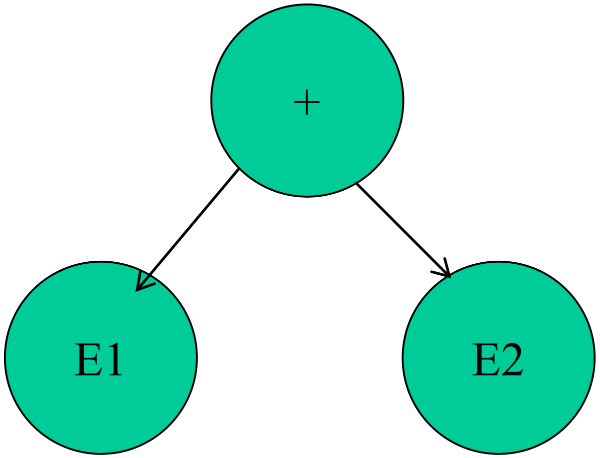
Code Emmision

- Code generator needs type decorations in AST from Semantic analysis
  - MethodBodyVisitor.visit(Plus n)

    if n.type == int
    - visit(n.E1)
    - visit(n.E2)
    - emit("iadd\n")

    else if n.type == float
    - visit(n.E1)
    - visit(n.E2)
    - emit(“fadd\n")

    else if …



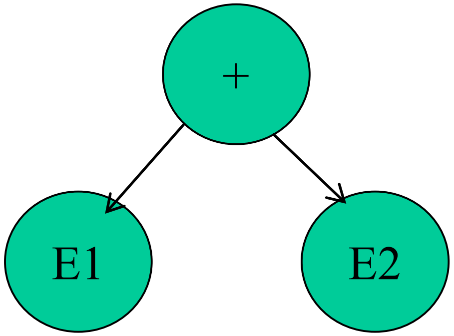
Code Emmision

- Code generator needs type decorations in AST from Semantic analysis
  - MethodBodyVisitor.visit(Plus n)

    … else if n.type == string

    - emit (new #4) // class StringBuilder
    - emit(dup)
    - emit(invokespecial #5) // Method StringBuilder."<init>"
    - Visit(n.E1) // String from E1
    - emit(invokevirtual #6) // Method StringBuilder.append:(LString;)LStringBuilder;
    - Visit(n.E2) // String from E2
    - emit(invokevirtual #6) // Method StringBuilder.append:(LString;)LStringBuilder;
    - emit(invokevirtual #7) // Method StringBuilder.toString:()LString;

Note that String is not a primitive type in Java



CodeGenVisitor

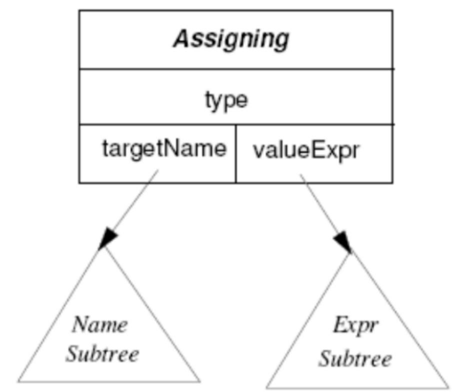
- TopVisitor
  - Top-level visitor – starts at root of AST
  - handles class/method declarations
  - calls others for specific needs (E.g., method bodies)
- MethodBodyVisitor
  - Generates most of the actual code
  - Calls others for specific needs (E.g., assignment LHS)

```
class NodeVisitor
    procedure visitChildren(n)
        foreach c ∈ n.getChildren( ) do
            call c.accept(this)                                    (1)
    end
end

class TopVisitor extends NodeVisitor
    procedure visit(ClassDeclaring cd)                             (2)
        /*   Section 11.2.1 on page 422                        */
    end
    procedure visit(MethodDeclaring md)                            (3)
        /*   Section 11.2.2 on page 424                        */
    end
end
/*   Continued in Figure 11.2                                  */
```

Figure 11.1: Structure of the code-generation visitors, with references to sections addressing specific constructs.

CodeGenVisitor

- SignatureVisitor
  - Handles AST subtrees for method definition or invocation
    - method name, parameter types, return type
  - Used by MethodBodyVisitor for invocations



- LHSVisitor
  - Generates code for LHS of assignments
  - May call other visitors if LHS contains subexpressions
    - Java example: a[x+y] = ...
    - Remember that LHS of assignment use the address of a variable, whereas the RHS uses the value.

Postludes

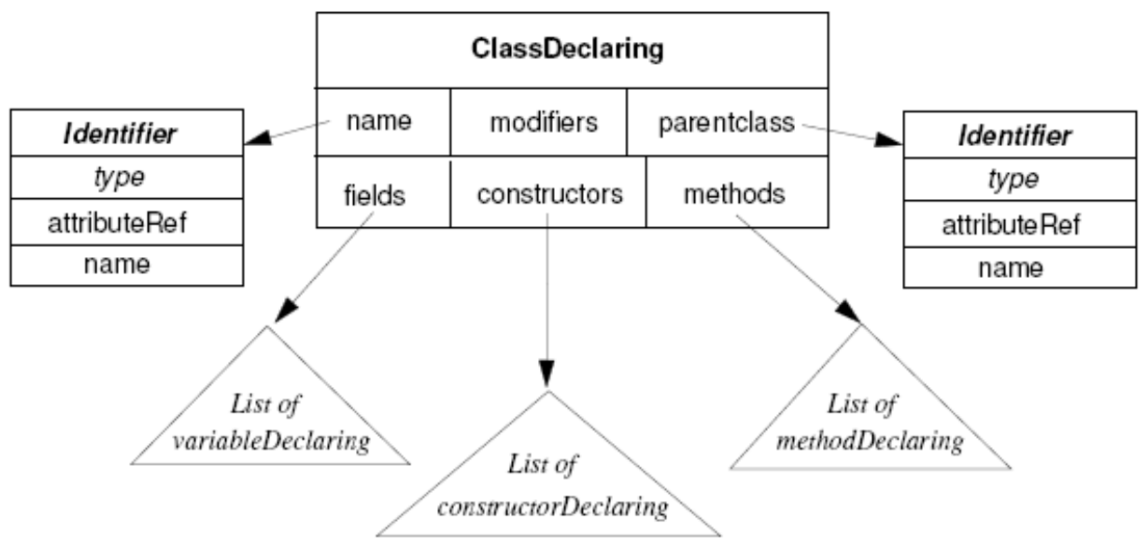
- Sometimes a single emission isn't enough
- Assignments:
  - Must visit LHS to find the storage location and type
  - Must visit RHS to compute the value
  - Must re-visit LHS to emit storage operations
- Inefficient!
- Better:
  - LHS visitor builds storage operation
  - Saves in a Postlude
  - Parent requests postlude emission

TopVisitor

- Handles class and method declarations
- visit(ClassDeclaring)
  - For jasmin, emits our class skeleton.
  - Name, modifiers, superclass, interfaces, fields
    - .class public foo
    - .super java/lang/Object
    - .field public myField I
- Note: no postlude needed

11.2.1 Class Declarations

```
/*  Visitor code for Marker (2)                                          */
procedure VISIT( ClassDeclaring cd )
    call EMITCLASSNAME( cd.GETCLASSNAME( ) )                            (14)
    foreach superclass ∈ cd.GETSUPERCLASSES( ) do
        call EMITEXTENDS( superclass )                                  (15)
    foreach field ∈ cd.GETFIELDS( ) do
        call EMITFIELDDECLARATION( field )                              (16)
    foreach static ∈ cd.GETSTATICS( ) do
        call EMITSTATICDECLARATION( static )                            (17)
    foreach node ∈ cd.GETMETHODS( ) do node.ACCEPT( this )              (18)
end
```



```java
/**
 * This outputs a standard prelude, with the class extending Object, a dummy
 * method for main(String[] args) that calls main431 Thus, your test file
 * must have a static main431 to kick things off
 */
public void visit(ClassDeclaring c) {
    emitComment("CSE431 automatically generated code file");
    emitComment("");
    emit(c, ".class public " + c.getName());
    emit(".super java/lang/Object");
    emit("; standard initializer\n\n" + ".method public <init>()V\n"
            + "   aload_0\n"
            + "   invokenonvirtual java/lang/Object/<init>()V\n"
            + "   return\n" + ".end method\n\n");
    emitComment("dummy main to call our main because we don't handle arrays");
    skip(2);
    emit(".method public static main([Ljava/lang/String;)V\n"
            + "   .limit locals 1\n" + "   .limit stack  3\n"
            + "   invokestatic " + c.getName() + "/main431()V\n"
            + "   return\n" + ".end method\n\n");
    visitChildren((AbstractNode) c);
}
```

TopVisitor

- visit(MethodDeclaring)
  - For jasmin, emits our method skeleton.
  - Name, modifiers, parameters, return types, limits
    - .method public static bar(S)I
    - .limit locals 2
    - .limit stack 4
- However, we need a postlude:
  - .end method
- How can we get the limits?
  - locals: from the method's Symbol Table
  - stack: from data flow analysis

Computing Offsets

- Each formal and local variables must have an offset in the stack frame
- The `this` object always has offset 0
- The naive solution:
  - enumerate all formals and locals
- The better solution:
  - reuse offsets for locals in disjoint scopes
- The clever solution:
  - exploit liveness information
  - must still respect the runtime types of locals
    - Each slot in the local array must have a unique type at each location (but not necessarily unique across the whole method)

Naive Offsets

```
public void m(int p [1] , int q [2] , Object r [3] ) {
   int x [4] = 42;
   int w [5] ;
   { int z [6] ;
      z = 87;
   }
   { boolean a [7] ;
      Object o [8] ;
      { boolean b [9] ;
         int z [10] ;
         b = true;
         boolean c [11] ;
         c = b && (x==87);
      }
      { int y [12] ;
         y = x;
      }
   }
}
```

max = 12

Better Offsets

```
public void m(int p [1] , int q [2] , Object r [3] ) {
   int x [4] = 42;
   int w [5] ;
   { int z [6] ;
     z = 87;
   }
   { boolean a [6] ;
     Object o [7] ;
     { boolean b [8] ;
       int z [9] ;
       b = true;
       boolean c [10] ;
       c = b && (x==87);
     }
     { int y [8] ;
       y = x;
     }
   }
}
```

max = 10

Clever Offsets

```
public void m(int p [1] , int q [2] , Object r [3] ) {
   int x [1] = 42;
   int w [2] ;
   { int z [2] ;
     z = 87;
   }
   { boolean a [2] ;
     Object o [2] ;
     { boolean b [2] ;
       int z [3] ;
       b = true;
       boolean c [3] ;
       c = b && (x==87);
     }
     { int y [2] ;
       y = x;
     }
   }
}
```

max = 3

11.2.2 Method Declarations

```
/*  Visitor code for Marker (3)                                          */
procedure VISIT( MethodDeclaring md )
    sigVisitor ← new SignatureVisitor( )                                 (19)
    call md.ACCEPT( sigVisitor )
    signature ← sigVisitor.GETSIGNATURE( )
    call EMITMETHODNAME( signature )                                     (20)
    call EMITMETHODALLOC( md.GETLOCALS( ), md.GETSTACK( ) )              (21)
    postludeLabel ← GENLABEL( )
    bodyVisitor ← new MethodBodyVisitor( postludeLabel )                 (22)
    md.GETBODY( ).ACCEPT( bodyVisitor )
    call EMITMETHODPOSTLUDE( postludeLabel )                             (23)
end
```

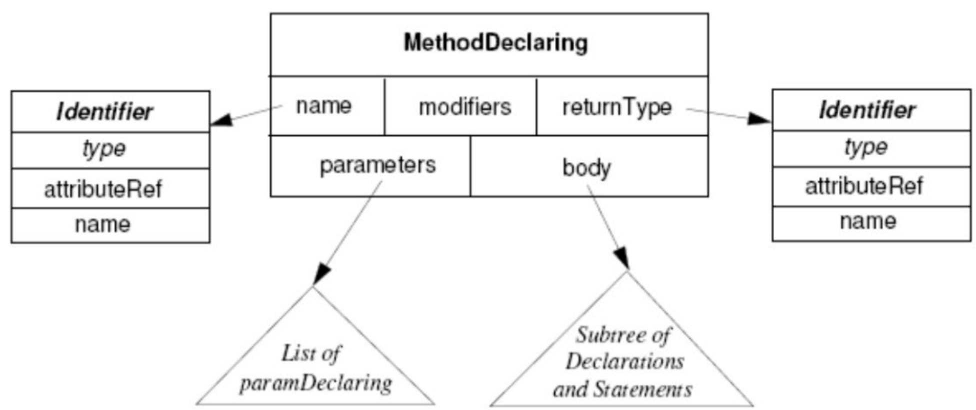


Figure 8.30: Abstract Syntax Tree for a Method Declaration

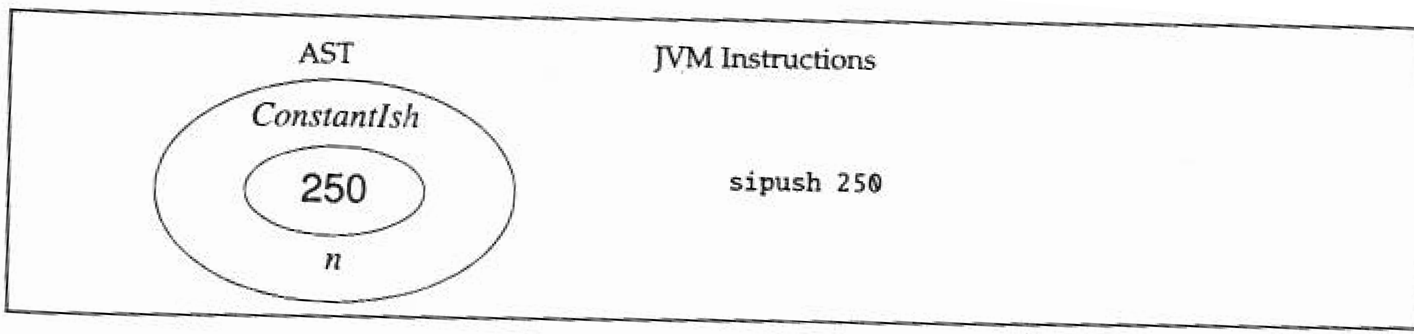
MethodBodyVisitor

- Generates code for the majority of nodes
  - LocalReferencing
  - ConstReferencing
  - StaticReferencing
  - FieldReferencing
  - ArrayReferencing
  - Computing – most binary and unary operators
  - Assigning – but remember the LHSVisitor!
  - Invoking – but remember the SignatureVisitor!
  - Control Structures

```
/*    Continued from Figure 11.1                                         */

class MethodBodyVisitor extends NodeVisitor
    procedure VISIT( ConstReferencing n )                                   (4)
        /*    Section 11.3.1 on page 425                                 */
    end
    procedure VISIT( LocalReferencing n )                                   (5)
        /*    Section 11.3.2 on page 426                                 */
    end
    procedure VISIT( StaticReferencing n )                                  (6)
        /*    Section 11.3.3 on page 427                                 */
    end
    procedure VISIT( Computing n )                                          (7)
        /*    Section 11.3.4 on page 427                                 */
    end
    procedure VISIT( Assigning n )                                          (8)
        /*    Section 11.3.5 on page 429                                 */
    end
    procedure VISIT( Invoking n )                                           (9)
        /*    Section 11.3.6 on page 430                                 */
    end
    procedure VISIT( FieldReferencing n )                                   (10)
        /*    Section 11.3.7 on page 432                                 */
    end
    procedure VISIT( ArrayReferencing n )                                   (11)
        /*    Section 11.3.8 on page 433                                 */
    end
    procedure VISIT( CondTesting n )                                        (12)
        /*    Section 11.3.9 on page 435                                 */
    end
    procedure VISIT( WhileTesting n )                                       (13)
        /*    Section 11.3.10 on page 436                                */
    end
end
```

Figure 11.2: Continuation of the code-generation visitors from Figure 11.1.

```
/*   Visitor code for Marker (4)                                        */
procedure VISIT( ConstReferencing n )
    loc ← ALLOCLOCAL( )                                                 (24)
    n.SETRESULTLOCAL( loc )                                             (25)
    call EMITCONSTANTLOAD( loc, n.GETCONSTANTVALUE( ) )
end
```

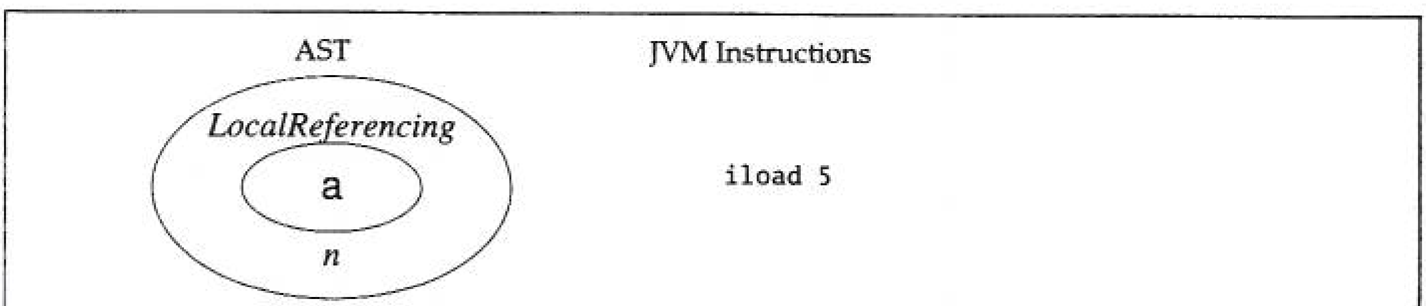


Choice of instructions: bipush (for 8 bit values), sipush, ldc or iconst

Note: loc <- allocLocal() is unnecessary for JVM/stack machines as loc is always top of stack, but for register machines it is needed. Ficher et. Al. are trying to be general here!

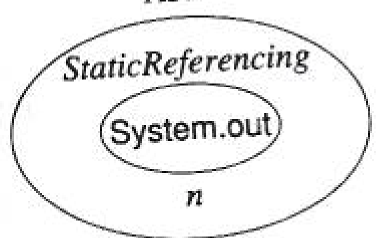
11.3.2 References to Local Storage

```
/*  Visitor code for Marker (5)                                   */
procedure VISIT( LocalReferencing n )
    loc ← ALLOCLOCAL( )                                            (26)
    n.SETRESULTLOCAL( loc )
    call EMITLOCALLOAD( loc, n.GETLOCATION( ) )                    (27)
end
```



```
/*  Visitor code for Marker ⑥
procedure VISIT( StaticReferencing n )
    call EMITSTATICREFERENCE( n.GETTYPE( ), n.GETNAME( ) )
end
```

AST



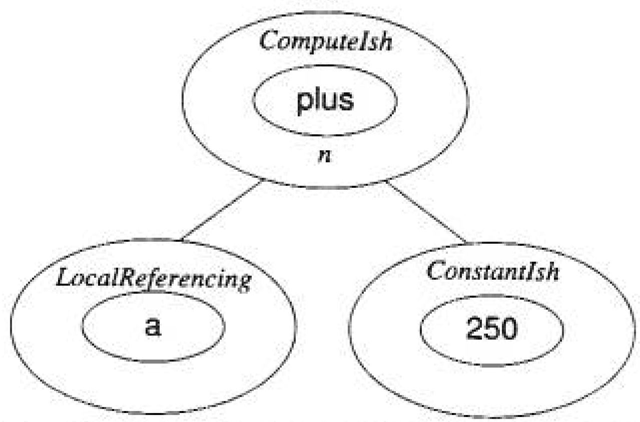
JVM Instructions

```
getstatic
    Ljava/io/PrintStream; java/lang/System/out
```

11.3.4 Expressions

```
/*   Visitor code for Marker (7)                          */
procedure VISIT( Computing n )
    VISITCHILDREN( n )                                  (28)
    loc ← ALLOCLOCAL( )                                 (29)
    n.SETRESULTLOCAL( loc )
    call EMITOPERATION( n )                             (30)
end
```

AST



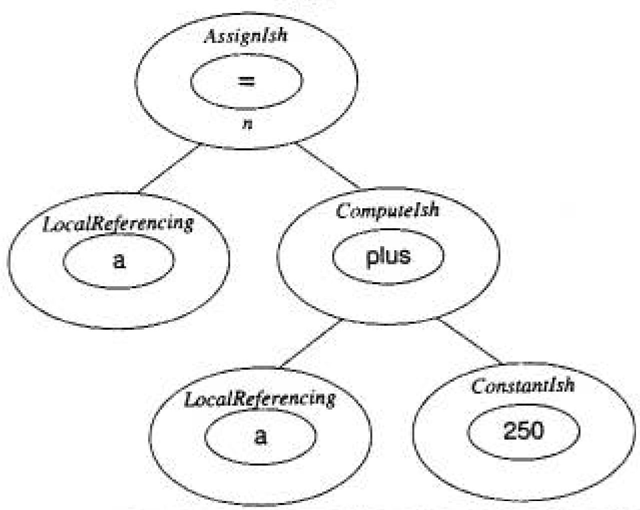
JVM Instructions

```
;   Code emitted for left child
    iload 5
;   Code emitted for right child
    sipush 250
;   Code for the Computing node
    iadd
```

11.3.5 Assignment

```
/*  Visitor code for Marker (8)                                          */
procedure VISIT(Assigning n)                                           (31)
    lhsVisitor ← new LHSVisitor(this)                                  (32)
    call n.GETLHS().ACCEPT(lhsVisitor)                                 (33)
    call n.GETRHS().ACCEPT(this)                                       (34)
    call lhsVisitor.EMITSTORE(n.GETRHS().GETRESULTLOCAL())
```

AST

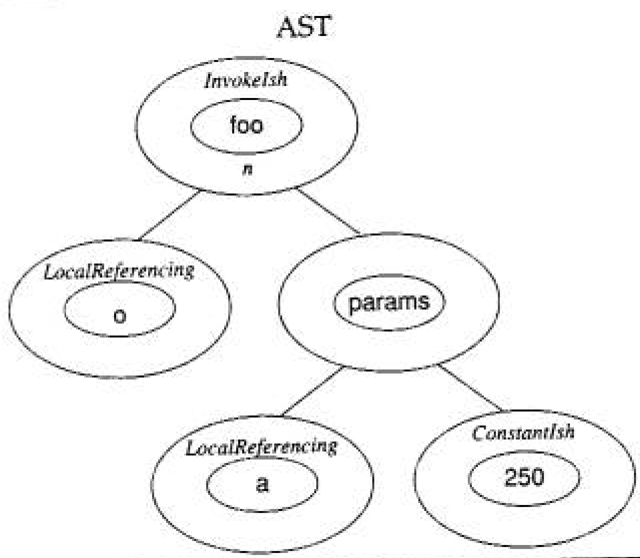


JVM Instructions

```
;   Code emitted for Computing
    iload 5
    sipush 250
    iadd
;   Code emitted by LHSVisitor
    istore 5
```

11.3.6 Method Calls

```
/*  Visitor code for Marker (9)                                        */
procedure VISIT( Invoking n )
    sigVisitor ← new SignatureVisitor( )
    call n.ACCEPT(sigVisitor)
    usageSignature ← sigVisitor.GETSIGNATURE( )                        (35)
    matchedSignature ← FINDSIGNATURE( usageSignature )                 (36)
    if not n.ISVOID( )                                                 (37)
    then
        loc ← ALLOCLOCAL( )
        call n.SETRESULTLOCAL( loc )
    if not n.ISSTATIC( )
    then
        call n.GETINSTANCE( ).ACCEPT( this )                           (38)
    foreach param ∈ n.GETPARAMS( ) do call param.ACCEPT( this )        (39)
    if n.ISVIRTUAL( )                                                  (40)
    then call EMITVIRTUALMETHODCALL( n )
    else call EMITNONVIRTUALMETHODCALL( n )
end
```

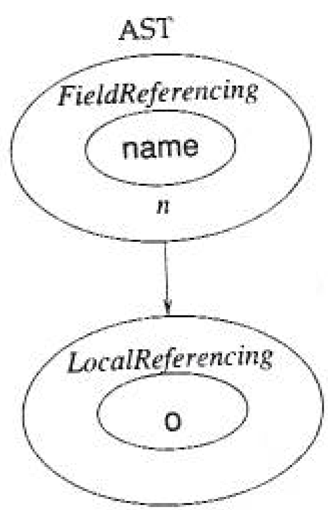


JVM Instructions

```
;    AST and code for invoking the
;    method foo from class MyClass:
;
;          o.foo(a, 250)
;
;    Marker (38) emits code
;    for the instance o:
     aload 4
;    Marker (39) emits code
;    for the parameters:
     iload 5
     sipush 250
;    The method call
     invokevirtual MyClass/foo(II)Z
```

11.3.7 Field References

```
/*   Visitor code for Marker (10)                                   */
procedure visit( FieldReferencing n )                              (41)
    call n.getInstance( ).accept(this)
    call emitFieldReference(n.getType( ), n.getName( ))
end
```



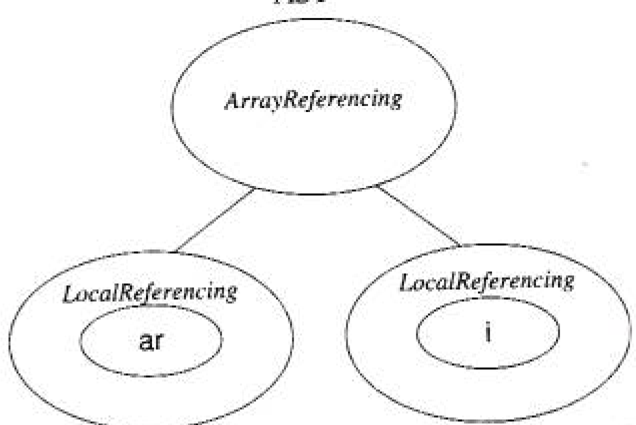
JVM Instructions

```
;   AST and code for fetching the
;   field name from class MyClass:
;
;           o.name
;
;   Code generated for the instance
    aload 4
;   Get the field's value
    getfield Ljava/lang/String; MyClass/name
```

11.3.8 Array References

```
/*  Visitor code for Marker (11)                                              */
procedure VISIT( ArrayReferencing n )
    call n.GETARRAY( ).ACCEPT( this )                                         (42)
    call n.GETINDEX( ).ACCEPT( this )                                         (43)
    call EMITARRAYREFERENCE( n.GETARRAY( ).GETTYPE( ) )                        (44)
end
```

AST



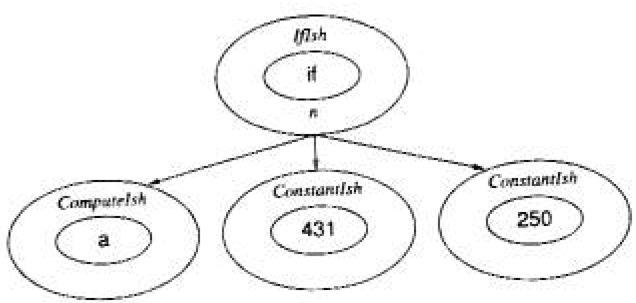
JVM Instructions

```
;    AST and code for fetching the
;    array element:
;
;           ar[i]
;
;    Code generated for the array, ar
     aload 3
;    Code generated for the index, i
     aload 4
;    Get the array element value
     iaload
```

11.3.9 Conditional Execution

```
/*   Visitor code for Marker (12)                                         */
procedure visit( CondTesting n )                                        (45)
    falseLabel ← genLabel( )
    endLabel ← genLabel( )                                              (46)
    call n.getPredicate( ).accept( this )
    predicateResult ← n.getPredicate( ).getResultLocal( )               (47)
    call emitBranchIfFalse( predicateResult, falseLabel )               (48)
    call n.getTrueBranch( ).accept( this )
    call emitBranch( endLabel )
    call emitLabelDef( falseLabel )                                     (49)
    call n.getFalseBranch( ).accept( this )
    call emitLabelDef( endLabel )
end
```

AST



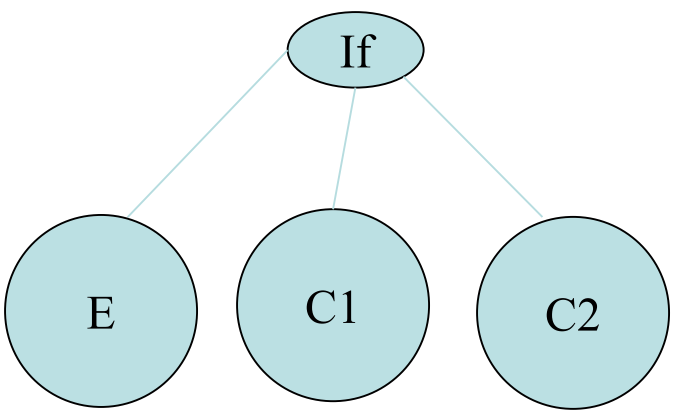
JVM Instructions

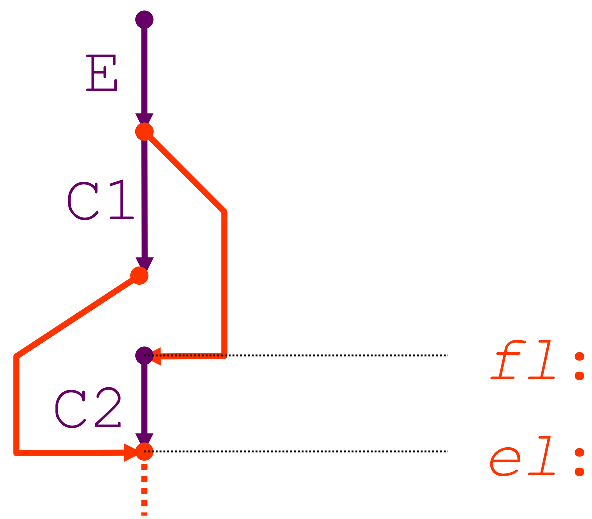
```
;   Predicate code (Marker (46))
    iload 5
;   Is the predicate false?
    ifeq falseLabel
;   True branch (Marker (48))
    sipush 431
    goto endLabel
falseLabel:
;   False branch (Marker (49))
    sipush 250
endLabel:
```

Code Templates

visit [**if** E **then** C1 **else** C2] =
      visit [E]
      JUMPIFFALSE *fl*
      visit [C1]
      JUMP *el*
fl: *visit* [C2]
el:

```
/*  Visitor code for Marker (12)
procedure visit( CondTesting n )
    falseLabel ← genLabel( )
    endLabel ← genLabel( )
    call n.getPredicate( ).accept( this )
    predicateResult ← n.getPredicate( ).getResultLocal( )
    call emitBranchIfFalse( predicateResult, falseLabel )
    call n.getTrueBranch( ).accept( this )
    call emitBranch( endLabel )
    call emitLabelDef( falseLabel )
    call n.getFalseBranch( ).accept( this )
    call emitLabelDef( endLabel )
end
```



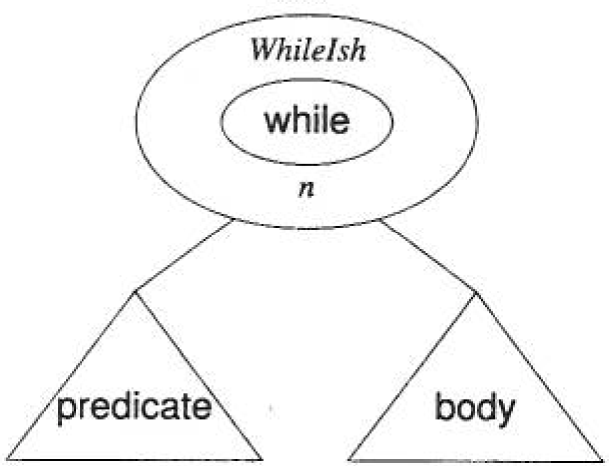


Pause

11.3.10 Loops

```
/*   Visitor code for Marker (13)                                   */
procedure VISIT( WhileTesting n )
    doneLabel ← GENLABEL( )                                          (50)
    loopLabel ← GENLABEL( )
    call EMITLABELDEF( loopLabel )
    n.GETPREDICATE( ).ACCEPT( this )                                 (51)
    predicateResult ← n.GETPREDICATE( ).GETRESULTLOCAL( )
    call EMITBRANCHIFFALSE( predicateResult, doneLabel )
    n.GETLOOPBODY( ).ACCEPT( this )                                  (52)
    call EMITBRANCH( loopLabel )
    call EMITLABELDEF( doneLabel )
end
```

AST



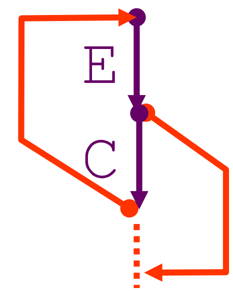
JVM Instructions

```
loopLabel:
;    Predicate code (Marker (51))
;    Predicate false?  → exit the loop
;    ifeq doneLabel
;    Loop body code(Marker (52))
     goto loopLabel
endLabel:
     Done with loop
```

Code Templates

While Command:

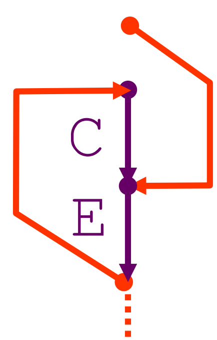
```
visit [while E do C] =
    l: visit [E]
        JUMPIFFALSE d
       visit[C]
       JUMP l
    d:
```

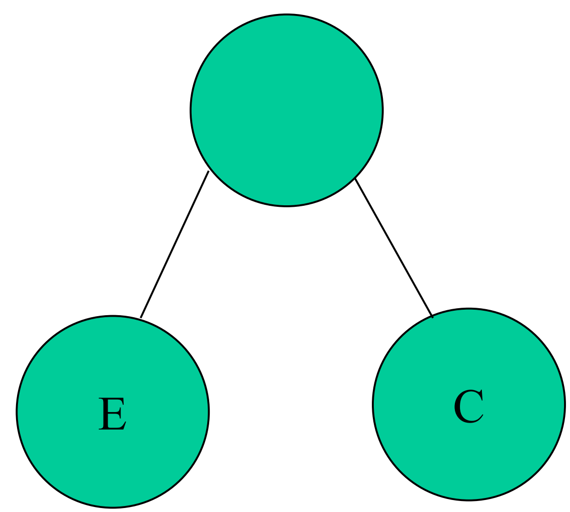



```
/*   Visitor code for Marker (13)
procedure visit( WhileTesting n )
    doneLabel ← genLabel( )
    loopLabel ← genLabel( )
    call emitLabelDef( loopLabel )
    n.getPredicate( ).accept( this )
    predicateResult ← n.getPredicate( ).getResultLocal( )
    call emitBranchIfFalse( predicateResult, doneLabel )
    n.getLoopBody( ).accept( this )
    call emitBranch( loopLabel )
    call emitLabelDef( doneLabel )
end
```

Alternative While Command code template:

```
visit [while E do C] =
        JUMP h
    l: visit [C]
    h: visit[E]
        JUMPIFTRUE l
```





LHSVisitor

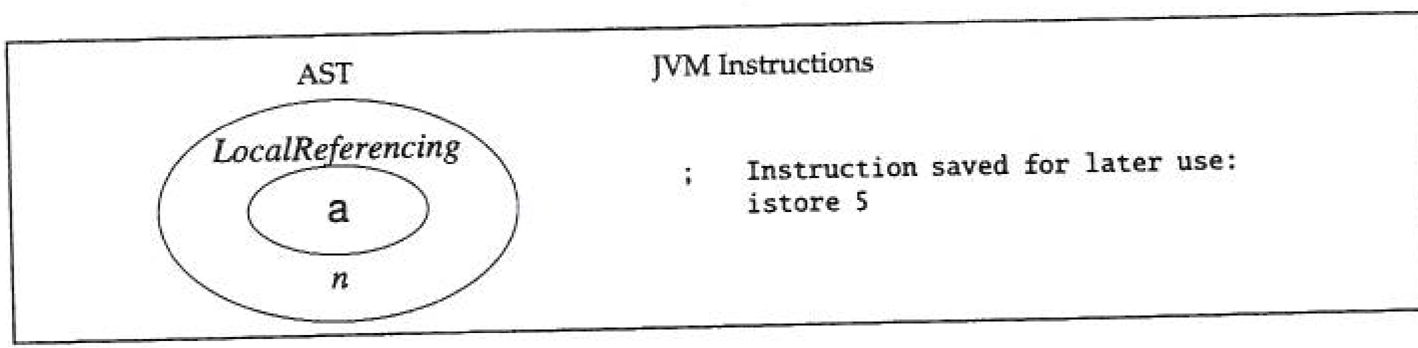
- Generates the correct address and postlude for a LHS
- May need to call other visitors for expressions (e.g., a[5])
  - Locals
    - No emission
    - Postlude: [type]store N
      - N from Localreferencing.getRegister()
  - Statics:
    - No emission
    - postlude: putstatic <Type> <name>
  - Fields
    - Emits object reference
    - Postlude: putfield <Type> <name>
  - Arrays:
    - Emits array reference and index
    - postlude: <type>astore

```
class LHSVisitor extends NodeVisitor
    constructor LHSVisitor( MethodBodyVisitor valueVisitor )
        this.valueVisitor ← valueVisitor                          (53)
    end
    procedure visit( LocalReferencing n )                          (54)
        /*   Section 11.4.1 on page 437                          */
    end
    procedure visit( StaticReferencing n )                         (55)
        /*   Section 11.4.2 on page 438                          */
    end
    procedure visit( FieldReferencing n )                          (56)
        /*   Section 11.4.3 on page 439                          */
    end
    procedure visit( ArrayReferencing n )                          (57)
        /*   Section 11.4.4 on page 439                          */
    end
end
```

Figure 11.3: Structure of the left-hand side visitor.

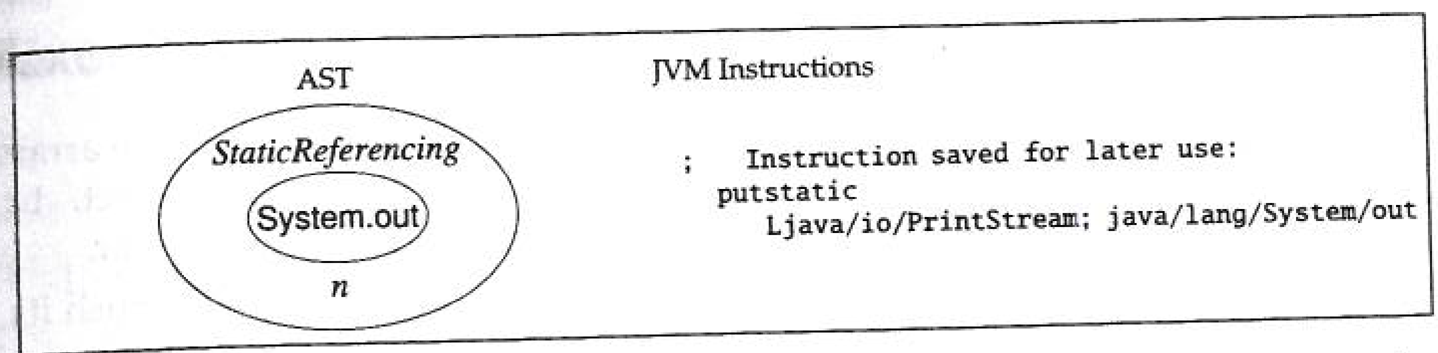
11.4.1 Local References

```
/*   Visitor code for Marker (54)                                          */
procedure VISIT( LocalReferencing n )                                      (58)
    call SETSTORE(new LocalStore(n.GETTYPE( ), n.GETLOCATION( )))
end
```



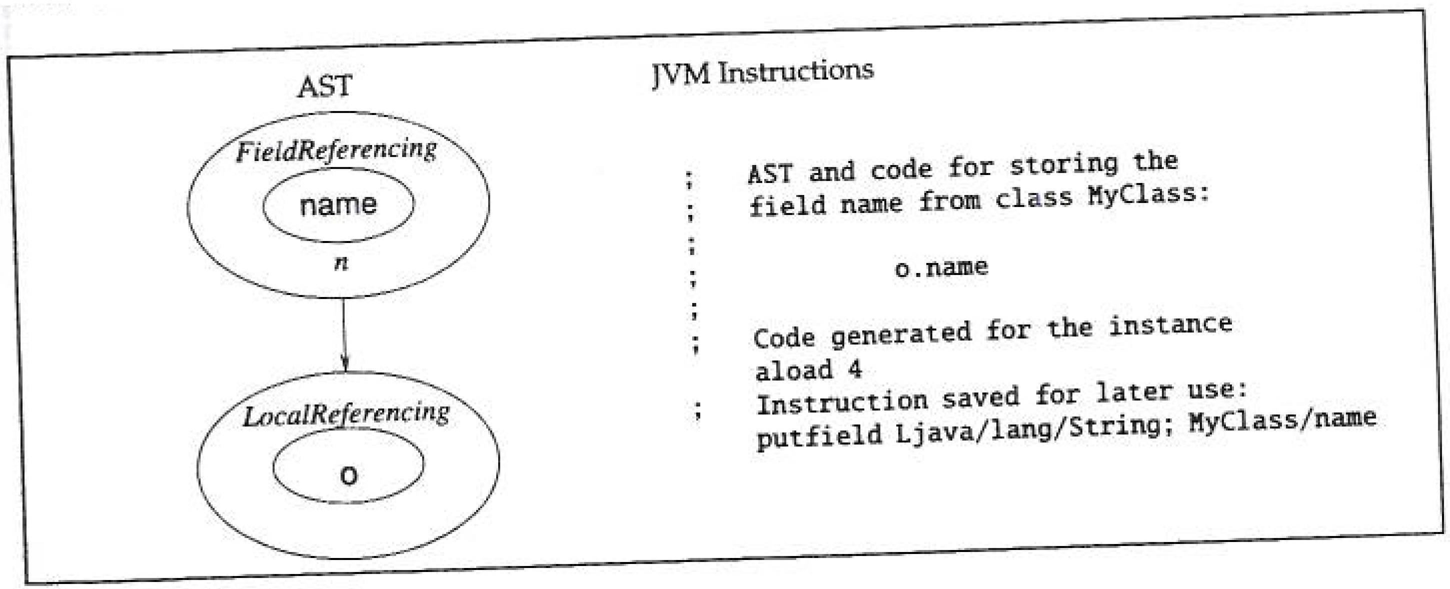
11.4.2 Static References

```
/*  Visitor code for Marker (55)                                   */
procedure VISIT( StaticReferencing n )
    call SETSTORE( new StaticStore( n.GETTYPE( ), n.GETNAME( ) ) )      (59)
end
```



11.4.3 Field References

```
/*  Visitor code for Marker (56)                                          */
procedure VISIT( FieldReferencing n )
    call n.GETINSTANCE( ).ACCEPT( valueVisitor )                          (60)
    call SETSTORE( new FieldStore( n.GETTYPE( ), n.GETNAME( ) ) )         (61)
end
```

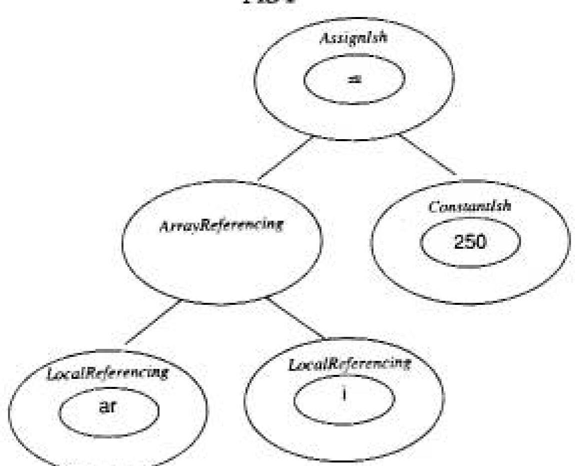


11.4.4 Array References

/★ Visitor code for Marker (57) ★/

procedure VISIT(*ArrayReferencing n*)
  call *n*.GETARRAY( ).ACCEPT(*valueVisitor*) (62)
  call *n*.GETINDEX( ).ACCEPT(*valueVisitor*) (63)
  call SETSTORE(**new** *ArrayStore* (*n*.GETARRAY( ).GETTYPE( ))) (64)
end

AST



JVM Instructions

```
;    AST and code for the
;    assignment:
;
;            ar[i] = 250
;
;    Code generated for the array, ar
     aload 3
;    Code generated for the index, i
     aload 4
;    Code generated for 250
     sipush 250
;    Code to store into the array element
     iastore
```

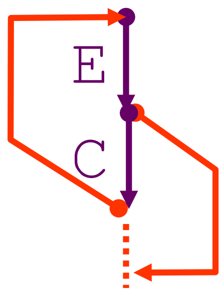
How to design the CodeVisitor?

- Idea from Brown & Watt
- Start with Code templates
  - Each statement and expression generates a sequence of bytecodes
  - A code template shows how to generate bytecodes for a given language construct and its constituents
- The template ignores the surrounding context
- And it ignores uniqueness of label names
  - The given label names are symbolic; you have to make sure they are unique via some genLabel method
- This yields a simple, recursive strategy for the code generation

Code Templates

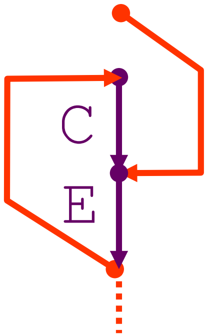
While Command:

visit [**while** E **do** C] =
  l: *visit* [E]
    JUMPIFFALSE d
    visit [C]
    JUMP l
  d:



Alternative While Command code template:

visit [**while** E **do** C] =
    JUMP h
  l: *visit* [C]
  h: *visit* [E]
    JUMPIFTRUE l



Code Template

- do-while

  visit [`do C while E`] =

- For loop

visit [`for ( C-init ; E ; C-update) C-body`] =

Code Template

- do-while

```
visit [do C while E] =
    l: visit [C]
       visit [E]
       JUMPIFTRUE l
```

- For loop

```
visit [for ( C-init ; E ; C-update) C-body] =
       visit [C-init]
  l:   visit [E]
       JUMPIFFALSE e
  u:   visit [C-body]
       visit [C-update]
       JUMP l
  e:
```

Examples

```
if(E) S1 else S2   ⇒   E
                       ifeq false
                       S1
                       goto endif
                       false:
                       S2
                       endif:
                       nop
```

```
if(E) S   ⇒   E
              ifeq false
              S
              false:
              nop
```

`nop` not strictly necessary b/c reachability constraints guarantee this is not the last instruction

```
while(E) S   ⇒   goto cond:
                 loop:
                 S
                 cond:
                 E
                 ifne loop
```

```
while(true) S   ⇒   loop:
                    S
                    goto loop
```

Code Template Invariants

- A statement and a void expression leaves the stack height unchanged
- A non-void expression increases the stack height by one
- This is a local property of each template
- The generated code must be verifiable
- This is not a local property, since the verifier performs a global static analysis

Representing Java types

Type	JVM designation
`boolean`	`Z`
`byte`	`B`
`double`	`D`
`float`	`F`
`int`	`I`
`long`	`J`
`short`	`S`
`void`	`V`
Reference type *t*	`L`*t*`;`
Array of type *a*	`[`*a*

Figure 10.4: Java types and their designation in the JVM. All of the integer-valued types are signed. For reference types, *t* is a fully qualified class name. For array types, *a* can be a primitive, reference, or array type.

Computing Signatures (1/2)

- The function sig(σ) encodes a type:

```
sig(void) = V            sig(byte) = B
sig(short) = S           sig(int) = I
sig(char) = C            sig(boolean) = Z
sig(σ[]) = [desc(σ)
sig(C1.C2. ... .Ck) = C1/C2/.../Ck

desc(void) = V           desc(byte) = B
desc(short) = S          desc(int) = I
desc(char) = C           desc(boolean) = Z
desc(σ[]) = [desc(σ)
desc(C1.C2. ... .Ck) = LC1/C2/.../Ck;
```

Computing Signatures (2/2)

- This extends to fields, methods, and constructors
- The field named x in class C:

```
sig(C)/x
```

- The method σ m (σ_1 x_1, ..., σ_k x_k) in class C:

```
sig(C)/m (desc(σ1)...desc(σk)) desc(σ)
```

- The constructor C (σ_1 x_1, ..., σ_k x_k) in class C:

```
sig(C)/<init> (desc(σ1)...desc(σk)) V
```

{ σ n = E; S } ⟹ E
σstore offset(n)
S

σstore is either istore or astore depending on σ

{ σ n; S } ⟹ S

E; ⟹ E

type(E) = void

E; ⟹ E
pop

type(E) ≠ void

throw E; ⟹ E
athrow

return E; ⟹ E
σreturn

`new` C (E_1, ..., E_k) @ δ →

```
new sig(C)
dup
E1
...
Ek
invokespecial sig(δ)
```

`this` (E_1, ..., E_k) @ δ →

```
aload 0
E1
...
Ek
invokespecial sig(δ)
```

@ δ indicates that δ is the corresponding resolved declaration `super` ($E_1, \ldots, E_k$) @ δ →

```
aload 0
```
E_1

...

E_k

`invokespecial` $sig(\delta)$

```
aload 0
```
I_1

`putfield` $sig(x_1)$ $desc(\sigma_1)$

...

```
aload 0
```
I_n

`putfield` $sig(x_n)$ $desc(\sigma_n)$

The current class contains the non-static field initializations:

σ_1 x_1 = I_1;

...

σ_n x_n = I_n;

`E.m(`E_1`,...,`E_k`)` @ δ →

E
E_1
...
E_k
`invokevirtual` sig(δ)

sig(δ) is a class

`E.m(`E_1`,...,`E_k`)` @ δ →

E
E_1
...
E_k
`invokeinterface` sig(δ)

sig(δ) is an interface

`C.m(`E_1`,...,`E_k`)` @ δ →

E_1
...
E_k
`invokestatic` sig(δ)

`(C)` E → E
`checkcast` sig(C)

`(char)` E → E
`i2c`

E `instanceof` C → E
`instanceof` sig(C)

`this` ⇒ `aload 0`

n ⇒ σ`load` offset(n)

type(n) = σ

E.f ⇒ E
`getfield` sig(f) desc(type(E.f))

C.f ⇒ `getstatic` sig(f) desc(type(C.f))

$E_1[E_2]$ ⇒ E_1
E_2
σ`aload`

type($E_1[E_2]$) = σ

σ`aload` is either `iaload`, `baload`, `saload`, `caload`, or `aaload` depending on σ

$n = E$ → E
dup
σstore offset(n)

type(n) = σ

$E_1.f = E_2$ → E_1
E_2
dup_x1
putfield sig(f) desc(type(E_1.f))

$C.f = E$ → E
dup
putstatic sig(f) desc(type(C.f))

$E_1[E_2] = E_3$ → E_1
E_2
E_3
dup_x2
σastore

type($E_1[E_2]$) = σ

new σ [E] → E
multianewarray desc(σ’) 1

type(new σ [E]) = σ’

E.length → E
arraylength

E.clone()

↓

E
invokevirtual sig(type(E))/clone()Ljava/lang/Object;

```
42      ⟹  ldc_int 42
```

```
true    ⟹  ldc_int 1
```

```
null    ⟹  aconst_null
```

```
"abc"   ⟹  ldc_string "abc"
```

```
private void emitICONST(int value) {
  if (value == -1)
    emit(JVM.ICONST_M1);
  else if (value >= 0 && value <= 5)
    emit(JVM.ICONST + "_" + value);
  else if (value >= -128 && value <= 127)
    emit(JVM.BIPUSH, value);
  else if (value >= -32768 && value <= 32767)
    emit(JVM.SIPUSH, value);
  else
    emit(JVM.LDC, value);
}
```

$E_1 + E_2$ ⟹ E_1
E_2
`iadd`

type(E_1+E_2) = `int`

$E_1 + E_2$ ⟹ E_1
E_2
`invokevirtual S/concat(LS;)LS;`

type(E_1+E_2) = `String`

S ≡ `java/lang/String`

– E ⟹ E
`ineg`

E_1 || E_2 →

E_1
```
dup
ifne firsttrue
pop
```
E_2
```
firsttrue:
nop
```

E_1 && E_2 →

E_1
```
dup
ifeq firstfalse
pop
```
E_2
```
firstfalse:
nop
```

`nop`s not strictly necessary b/c reachability constraints guarantee this is not the last instruction

An Example

```java
public int m(int x) {
   if (x < 0)
      return (x * x);
   else
      return (x * x * x);
}
```

```
.method public m(I)I
.limit locals 2
.limit stack 2
   iload_1
   iconst_0
   if_icmplt true_2
   iconst_0
   goto stop_3
true_2:
   iconst_1
stop_3:
   ifeq else_0
   iload_1
   iload_1
   imul
   ireturn
   goto stop_1
else_0:
   iload_1
   iload_1
   imul
   iload_1
   imul
   ireturn
stop_1:
   nop
.end method
```

Code generation summary

- Create code templates inductively
  - There may be special case templates generating equivalent, but more efficient code
  - Keep in mind what goes on at compile time
    - AST traversal order
  - Keep in mind what goes on at run time
    - Control flow order
- Use visitors pattern to walk the AST recursively emitting code as you go along

What can you do in your project now?

- Use the idea of code templates for defining the code generation phase of your compiler
- Generate code for the JVM
  - At least for a (small) part of your language

Languages and Compilers
(SProg og Oversættere)

Lecture 17
Storage Allocations and Run Time Management

Bent Thomsen

Department of Computer Science

Aalborg University

Learning goals

- Understand
  - Data representation (direct vs. indirect)
  - Storage allocation strategies:
    - static vs. dynamic (stack and heap)
  - Activation records (sometimes called frames)
  - Why may we need heap allocation
- Gain an overview of
  - Garbage collection strategies (Types of GCs)

Issues in Code Generation

- Code Selection:

  Deciding which sequence of target machine instructions will be used to implement each phrase in the source language.

- Storage Allocation

  Deciding the storage address for each variable in the source program. (static allocation, stack allocation etc.)

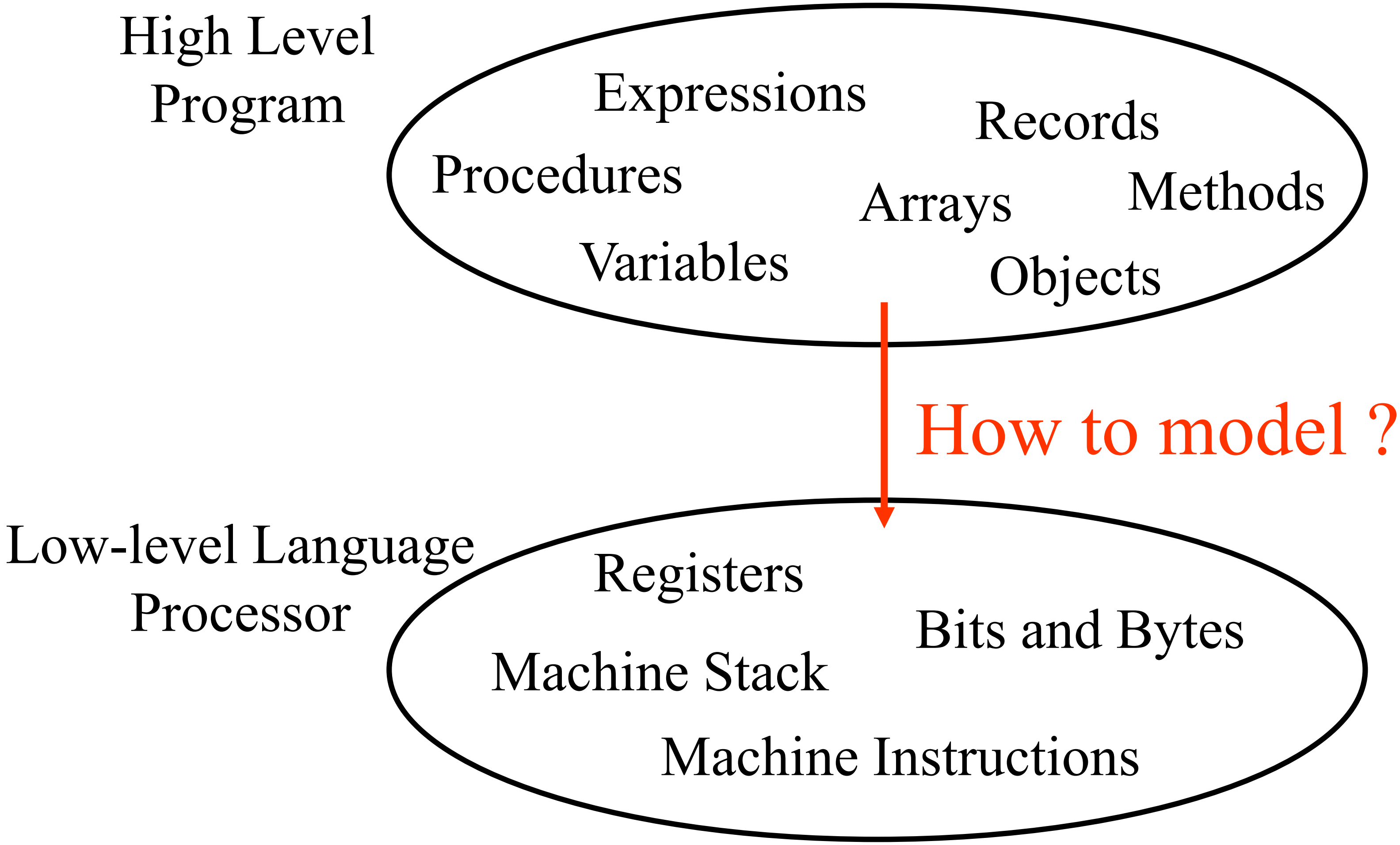
- Register Allocation (for register-based machines)

  How to use registers efficiently to store intermediate results.

We will look at register allocation in later lectures

What are (some of) the issues

How to model high-level computational structures and data structures in terms of low-level memory and machine instructions.



Easy for Java (or Java like) on the JVM

Type	JVM designation
`boolean`	`Z`
`byte`	`B`
`double`	`D`
`float`	`F`
`int`	`I`
`long`	`J`
`short`	`S`
`void`	`V`
Reference type *t*	L*t*;
Array of type *a*	[*a*

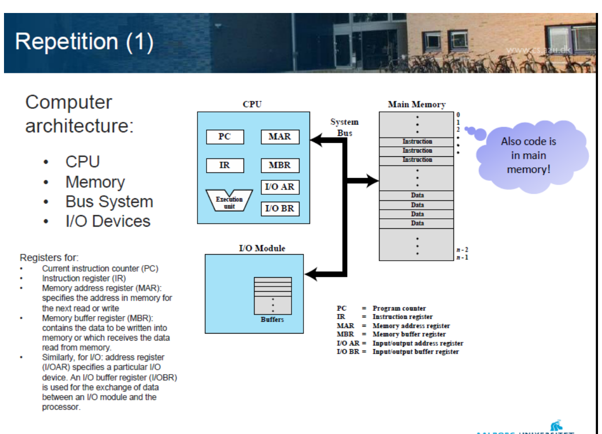
Figure 10.4: Java types and their designation in the JVM. All of the integer-valued types are signed. For reference types, *t* is a fully qualified class name. For array types, *a* can be a primitive, reference, or array type.

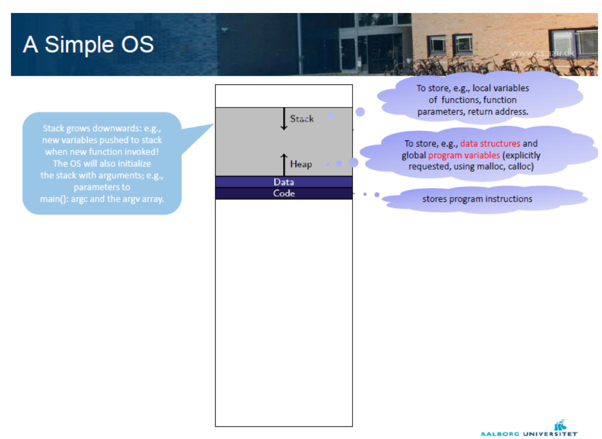
For other Languages on the JVM some thoughts
Are needed on a suitable mapping

Back in the olden days....

- No memory organization
- Programs had access to all of memory
- Memory was one big array of bytes
- No distinction between code and data

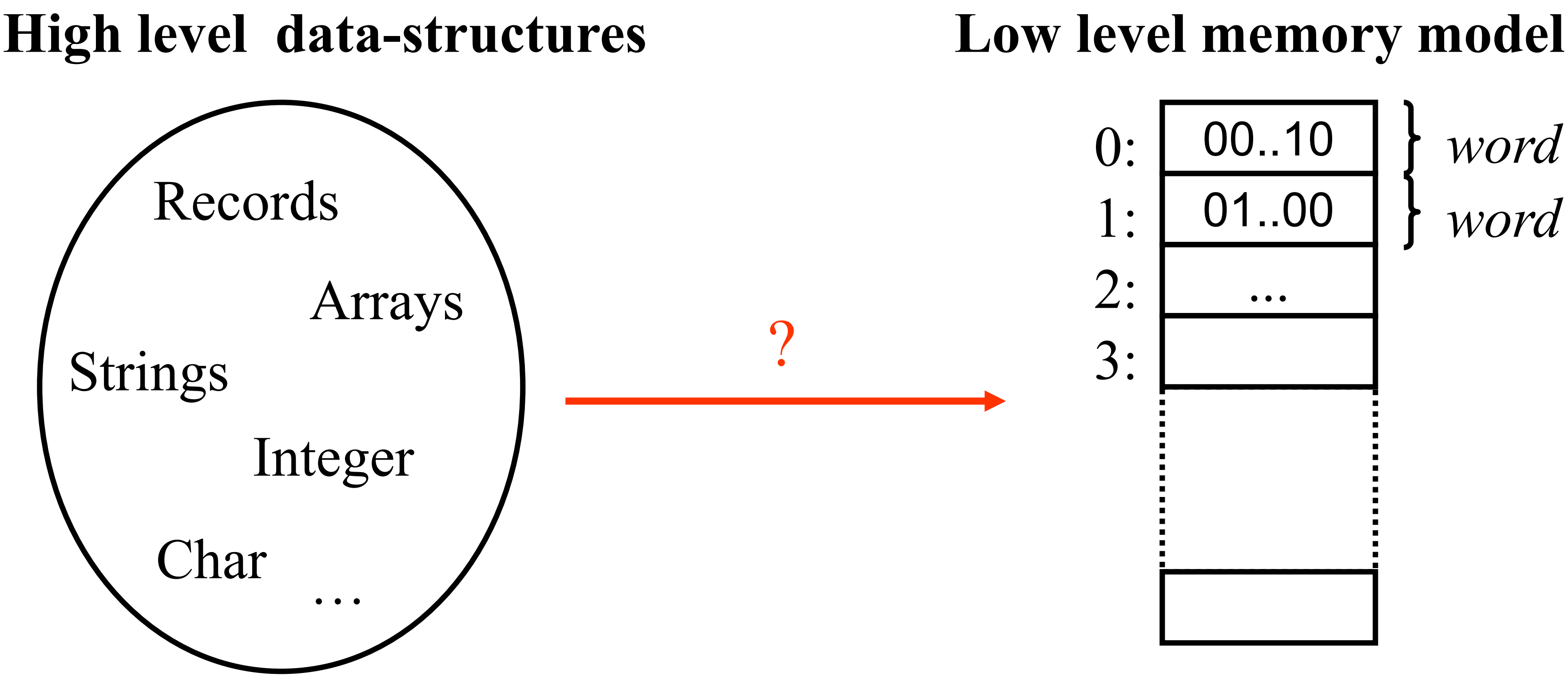
- Not just so in the old days also so for:
  - Low level VMs
  - Assember/Machine code
  - Connection with the CART and PSS courses:





Data Representation

- **Data Representation:** how to represent values of the source language on the target machine.



Note: addressing schema and size of "memory units" may vary

Data Representation

Important properties of a representation schema:

- **non-confusion:** different values of a given type should have different representations
- **uniqueness:** Each value should always have the same representation.

These properties are very desirable, but in practice they are not always satisfied:

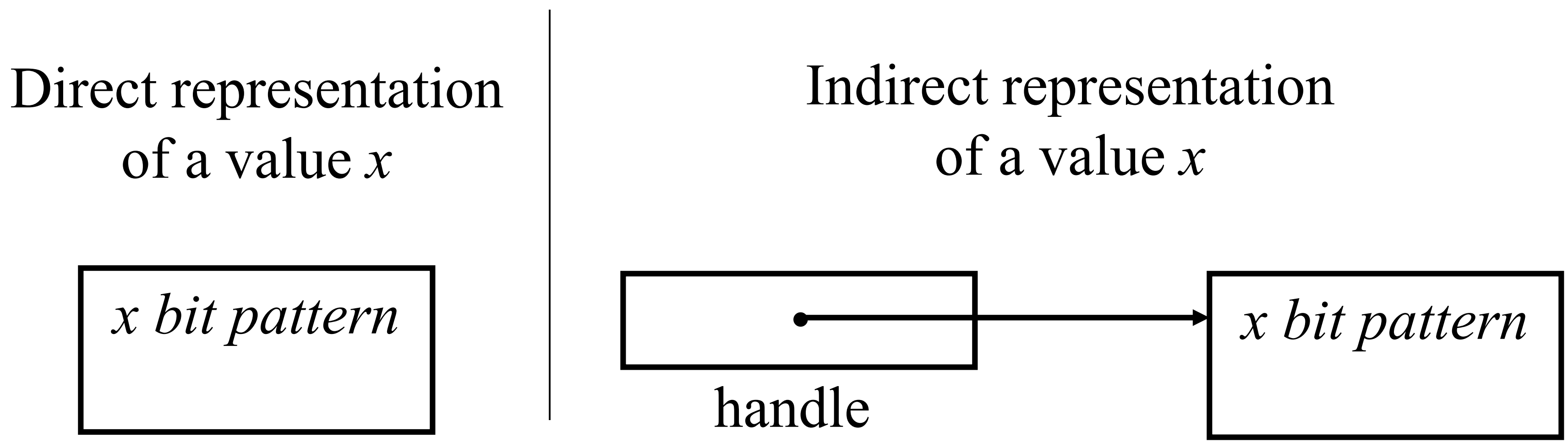
Example:

- confusion: approximated floating point numbers.
- non-uniqueness: one's complement representation of integers
  +0 and -0

Data Representation

Important issues in data representation:

- **constant-size representation:** The representation of all values of a given type should occupy the same amount of space.
- **direct** versus **indirect** representation

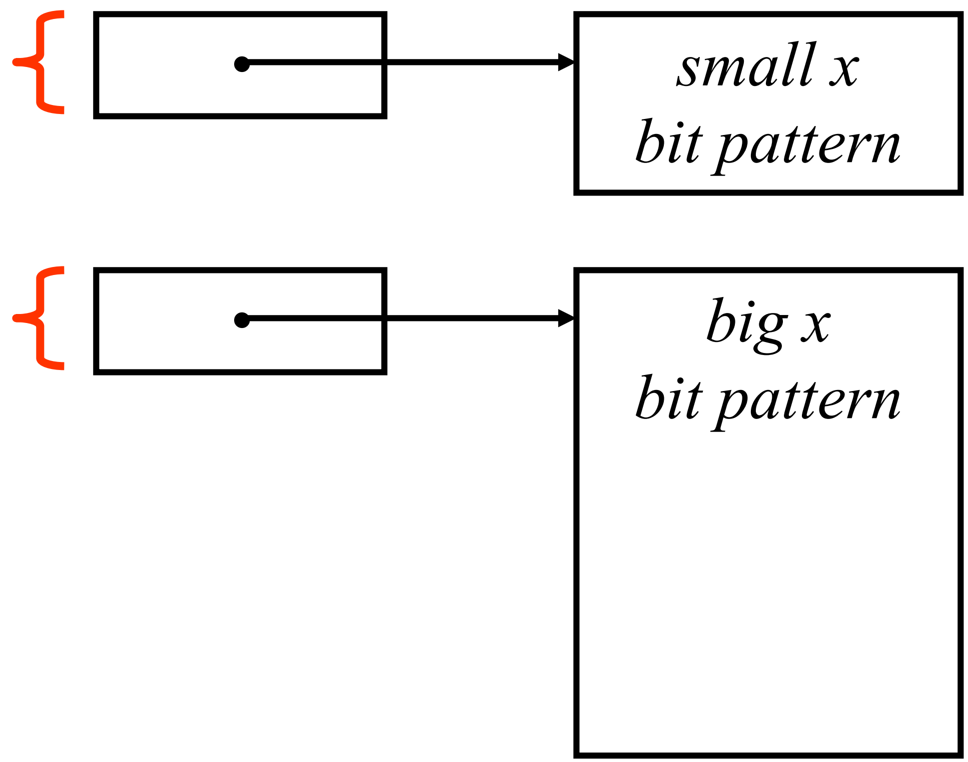


Indirect Representation

Q: What reasons could there be for choosing indirect representations?

To make the representation "constant size" even if representation requires different amounts of memory for different values.

Both are represented by pointers =>Same size



Indirect versus Direct

The choice between indirect and direct representation is a key decision for a language designer/implementer.

- Direct representations are often preferable for efficiency:
  - More efficient access (no need to follow pointers)
  - More efficient "storage class" (e.g stack rather than heap allocation)
- For types with widely varying size of representation it is almost a must to use indirect representation (see previous slide)

Languages like Pascal, C, C++ try to use direct representation wherever possible.

Languages like Scheme, ML, Python use mostly indirect representation everywhere (because of polymorphic higher order functions)

Java: primitive types direct, "reference types" indirect, e.g. objects and arrays.

Data Representation

We now survey representation of the data types found in C-like languages (Triangle), assuming direct representations wherever possible.

We will discuss representation of values of:

- Primitive Types
- Record Types
- Static Array Types
- Dynamic Array Types

We will use the following notations (if T is a type):

#[T] The cardinality of the type (i.e. the number of possible values)

size[T] The size of the representation (in number of bits/bytes)

Data Representation: Primitive Types

What is a primitive type?

The primitive types of a programming language are those types that cannot be decomposed into simpler types. For example `integer, boolean, char`, etc.

Type: `boolean`

Has two values *true* and *false*

=> #[`boolean`] = 2

=> *size*[`boolean`] ≥ 1 bit

	Possible Representation		
Value	1bit	byte(option 1)	byte(option2)
false	0	00000000	00000000
true	1	00000001	11111111

Note: In general if $\#[T] = n$ then $size[T] \geq \log_2 n$ bits

Data Representation: Primitive Types

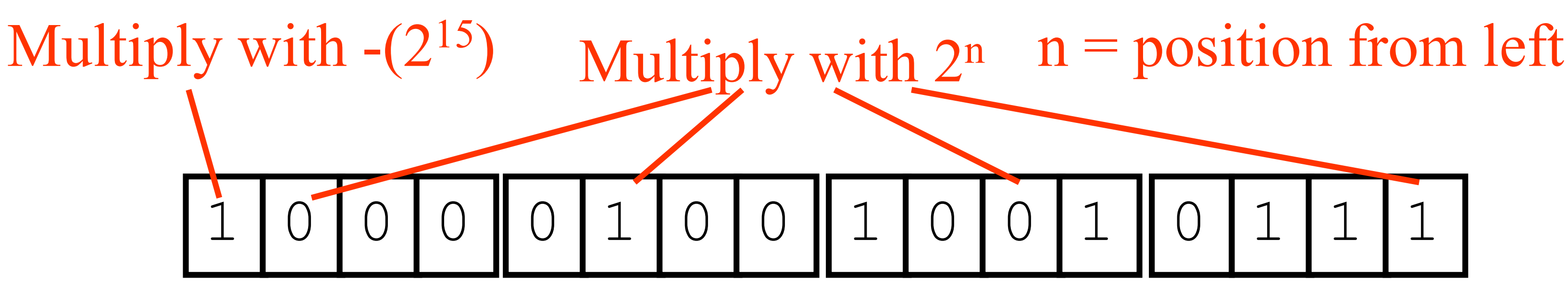
Type: `integer`

Fixed size representation, usually dependent (i.e. chosen based on) what is efficiently supported by target machine. Typically uses one word (16 bits, 32 bits, or 64 bits) of storage.

size[`integer`] = *word (= 16 bits)*

=> #[`integer`] ≤ 2^{16} = 65536

Modern processors use two's complement representation of integers



Value = $-1.2^{15} + 0.2^{14} + \ldots + 0.2^3 + 1.2^2 + 1.2^1 + 1.2^0$

Data Representation: Composite Types

Composite types are types which are not "atomic", but which are constructed from more primitive types.

- Records (called structs in C)
  Aggregates of several values of several different types
- Arrays
  Aggregates of several values of the same type
- Variant Records or Disjoint Unions
- Pointers or References
- (Objects)
- Functions

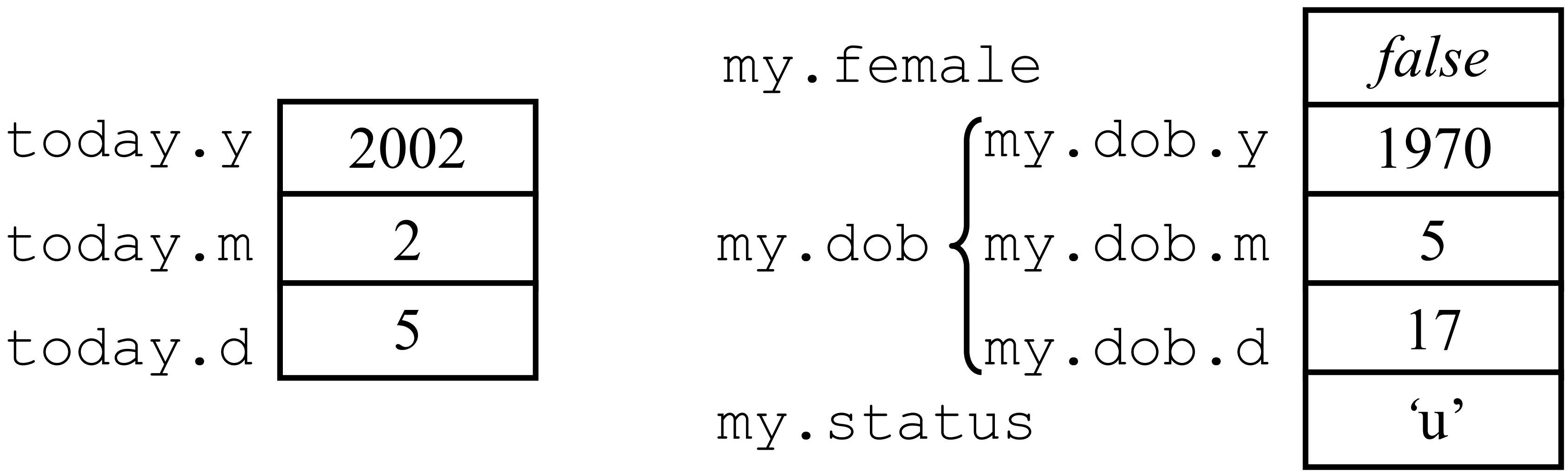
Data Representation: Records

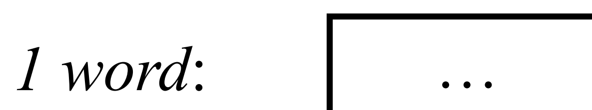
Example: Triangle Records

```
type Date = record
                y : Integer,
                m : Integer,
                d : Integer
             end;
type Details = record
                     female : Boolean,
                     dob :    Date,
                     status : Char
                end;
var today: Date;
var my:   Details
```

Data Representation: Records

Example: Triangle Record Representation

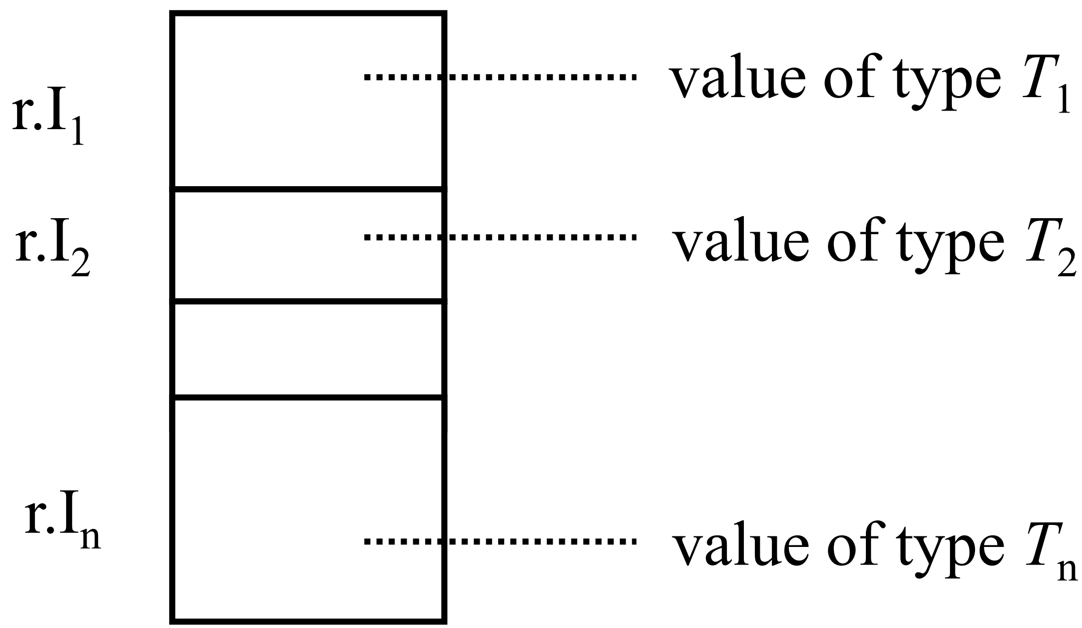




Data Representation: Records

Records occur in some form or other in most programming languages:
Ada, Pascal, Triangle (here they are actually called records)
C, C++, C# (here they are called structs).

The usual representation of a record type is just the concatenation of individual representations of each of its component types.



Data Representation: Records

Q: How much space does a record take up?
And how to access record elements?

Example:

size[`Date`] = 3*size[`integer`] = 3 words

address[today.y] = address[today]+0

address[today.m] = address[today]+1

address[today.d] = address[today]+2

address[my.dob.m] = address[my.dob]+1 = address[my]+2

Note: these formulas assume that addresses are indexes of words (not bytes) in memory (otherwise multiply offsets by 2)

Data Representation: Disjoint Unions

What are disjoint unions?
Like a record, has elements which are of different types. But the elements never exist at the same time. A "type tag" determines which of the elements is currently valid.

Example: Pascal variant records

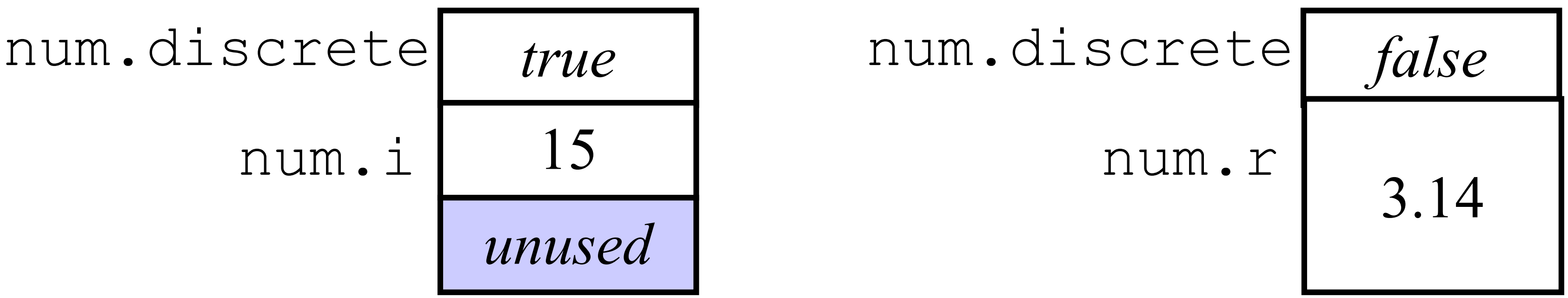
```
type Number = record
                  case discrete: Boolean of
                     true: (i: Integer);
                     false: (r: Real)
              end;
var num: Number
```

Mathematically we write disjoint union types as: $T = T_1 \mid \ldots \mid T_n$

Data Representation: Disjoint Unions

Example: Pascal variant records representation

```
type Number = record
                case discrete: Boolean of
                   true: (i: Integer);
                   false: (r: Real)
              end;
var num: Number
```



Assuming *size*[Integer]=size[Boolean]=1 and *size*[Real]=2, then
size[Number] = *size*[Boolean] + MAX(size[Integer], size[Real])
= 1 + MAX(1, 2) = 3

Data Representation: Disjoint Unions

```
type T = record
  case Itag: Ttag of
    v1: (I1: T1);
    v2: (I2: T2);
   ...
    vn: (In: Tn);
end;
var u: T
```

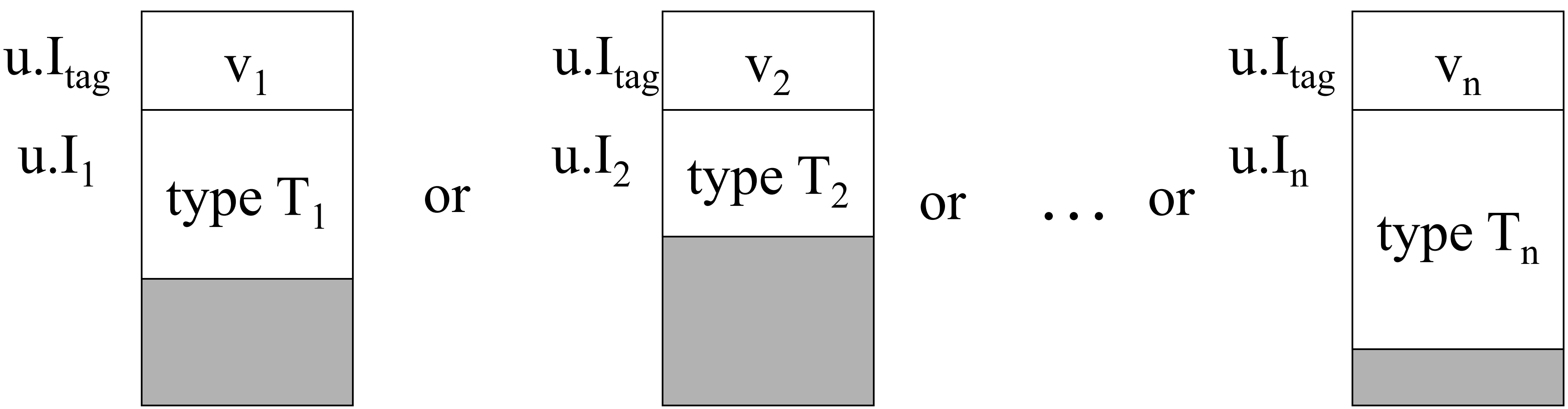
$$size[\mathrm{T}] = size[\mathrm{T}_{tag}] + \mathrm{MAX}(size[\mathrm{T}_1], ..., size[\mathrm{T}_n])$$

$$address[\mathrm{u.I}_{tag}] = address[\mathrm{u}]$$

$$address[\mathrm{u.I}_1] = address[\mathrm{u}]+size[\mathrm{T}_{tag}]$$

...

$$address[\mathrm{u.I}_n] = address[\mathrm{u}]+size[\mathrm{T}_{tag}]$$



Arrays

An array is a composite data type, an array value consists of multiple values of the same type. Arrays are in some sense like records, except that their elements all have the same type.

The elements of arrays are typically indexed using an integer value (In some languages such as for example Pascal, also other "ordinal" types can be used for indexing arrays).

Two kinds of arrays (with different runtime representation schemas):

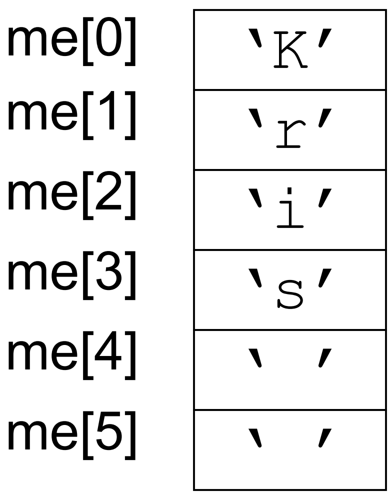
- **static** arrays: their **size** (number of elements) is **known** at **compile time**.
- **dynamic** arrays: their size can not be known at compile time because the number of elements is computed at run-time and sometimes may vary at run-time (Flex-arrays).

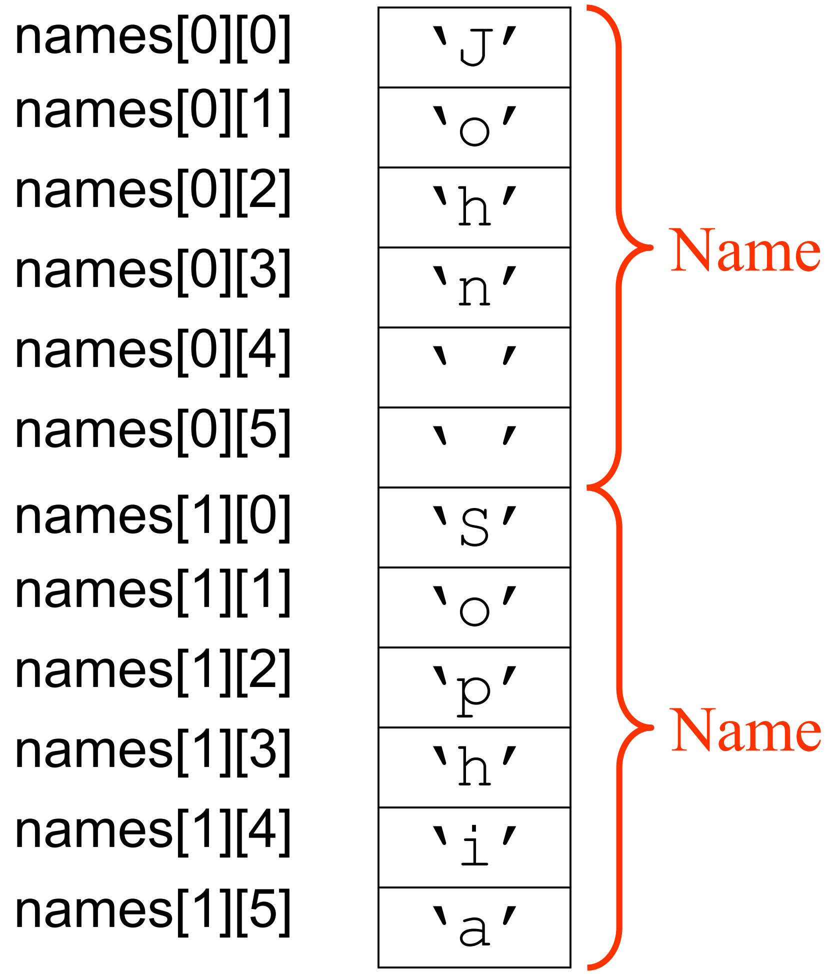
Q: Which are the "cheapest" arrays? Why?

Static Arrays

Example:

type Name = array 6 of Char;
var me: Name;
var names: array 2 of Name

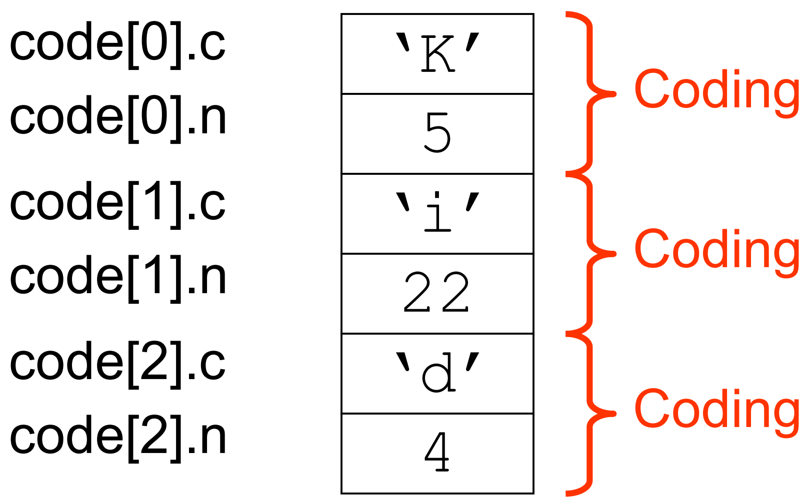




Static Arrays

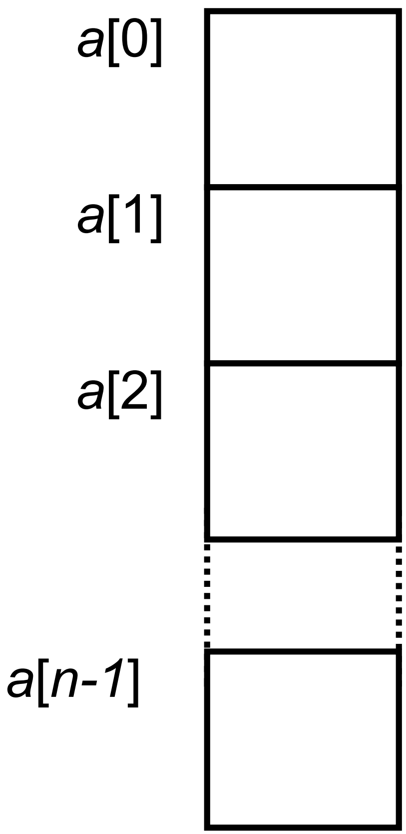
Example:

```
type Coding = record
   Char c, Integer n
end

var code: array 3 of Coding
```



Static Arrays

type *T* = **array** *n* **of** *TE*;
var *a* : *T*;



$$size[\mathrm{T}] = \mathrm{n} * size[\mathrm{TE}]$$

$$address[\mathrm{a}[0]] = address[\mathrm{a}]$$
$$address[\mathrm{a}[1]] = address[\mathrm{a}]+size[\mathrm{TE}]$$
$$address[\mathrm{a}[2]] = address[\mathrm{a}]+2*size[\mathrm{TE}]$$
…
$$address[\mathrm{a}[i]] = address[\mathrm{a}]+i*size[\mathrm{TE}]$$
…

Dynamic Arrays

Dynamic arrays are arrays whose size is not known until run time.

Example: Java Arrays (<u>all</u> arrays in Java are dynamic)

```
char[ ] buffer;

buffer = new char[buffersize];

...
for (int i=0; i<buffer.length; i++)
   buffer[i] = ' ';
```

Dynamic array: no size given in declaration

Array creation at runtime determines size

Can ask for size of an array at run time

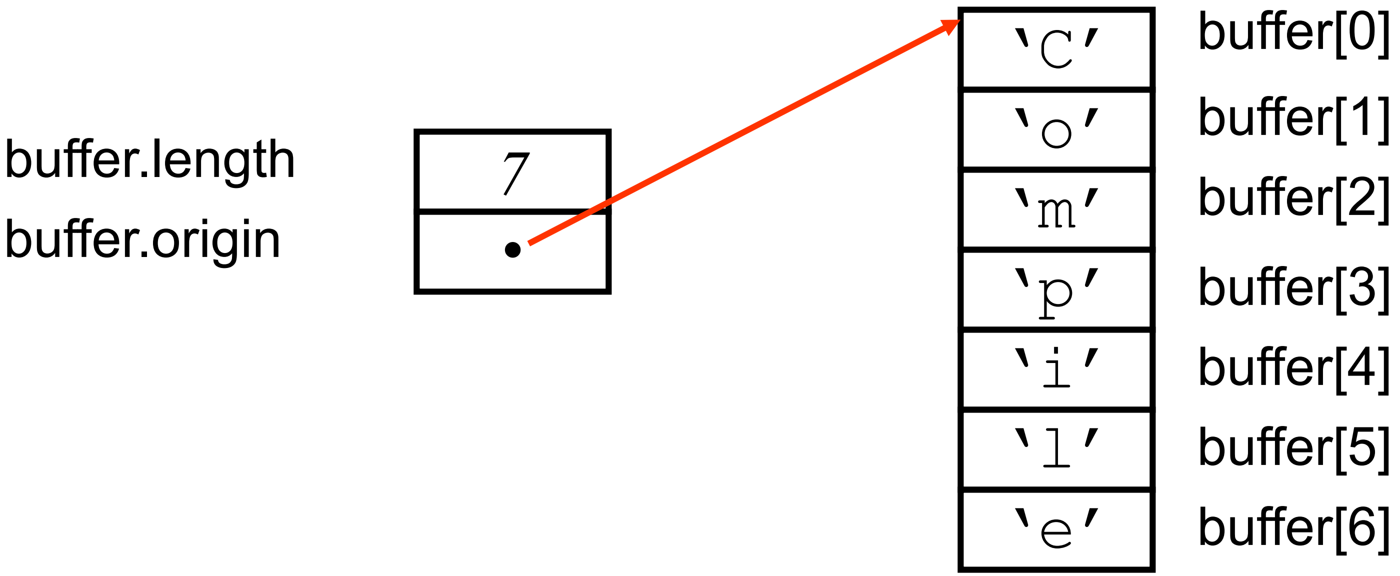
Q: How could we represent Java arrays?

Dynamic Arrays

Java Arrays

```
char[ ] buffer;

buffer = new char[7];
```

A possible representation for Java arrays

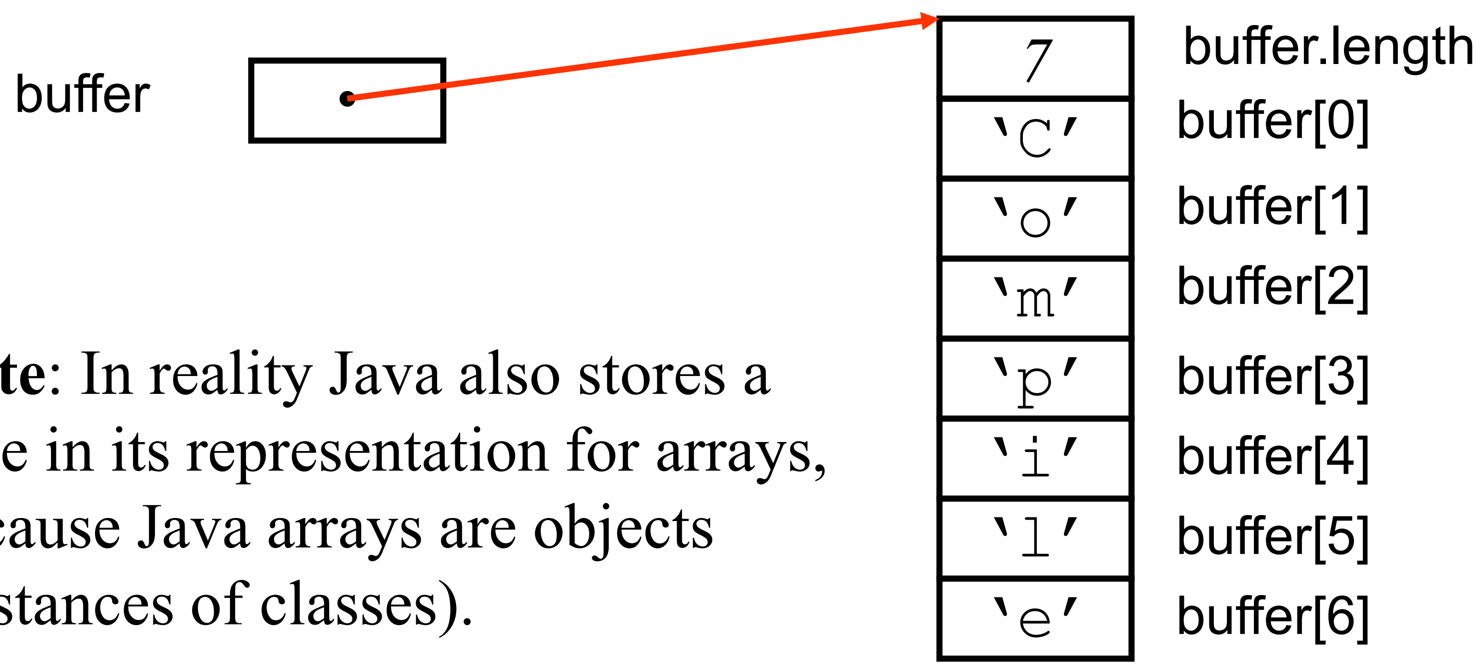


Dynamic Arrays

Java Arrays

```
char[ ] buffer;

buffer = new char[7];
```

Another possible representation for Java arrays



Note: In reality Java also stores a type in its representation for arrays, because Java arrays are objects (instances of classes).

Where to put data?

Now we have looked at how program structures are implemented in a computer memory

Next we look at where to put them

We will cover 3 methods:

1) static allocation,

2) stack allocation, and

3) heap allocation.

Static Allocation

Originally, all data were global.

Correspondingly, all memory allocation was static.

During compilation, data was simply placed at a fixed memory address for the entire execution of a program. This is called static allocation.

Examples are all assembly languages, Cobol, and Fortran.

Note: code is (still) usually allocated statically

Static Allocation (Cont.)

Static allocation can be quite wasteful of memory space. To reduce storage needs, in Fortran, the *equivalent* statement overlays variables by forcing two variables to share the same memory locations. In C,C++, *union* does this too.

Overlaying hurts program readability, as assignment to one variable changes the value of another.

In more modern languages, static allocation is used for global variables and literals (constant) that are fixed in size and accessible throughout program execution.

It is also used for static and extern variables in C/C++ and for static fields in C# and Java classes.

Stack Allocation

Recursive languages require dynamic memory allocation. Each time a recursive method is called, a new copy of local variables (frame) is pushed on a runtime stack. The number of allocations is unknown at compile-time.

A frame (or activation record) contains space for all of the local variables in the method. When the method returns, its frame is popped and the space reclaimed.

Thus, only the methods that are actually executing are allocated memory space in the runtime stack. This is called stack allocation.

```
p(int a) {
   double b;
   double c[10];
   b = c[a] * 2.51;
}
```

Figure 12.1: A Simple Subprogram

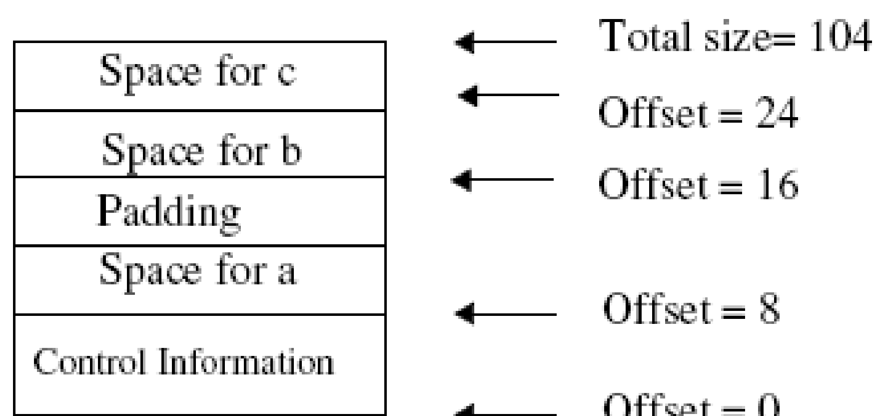


Figure 12.2: Frame for Procedure p

Stack Storage Allocation

Now we will look at allocation of local variables

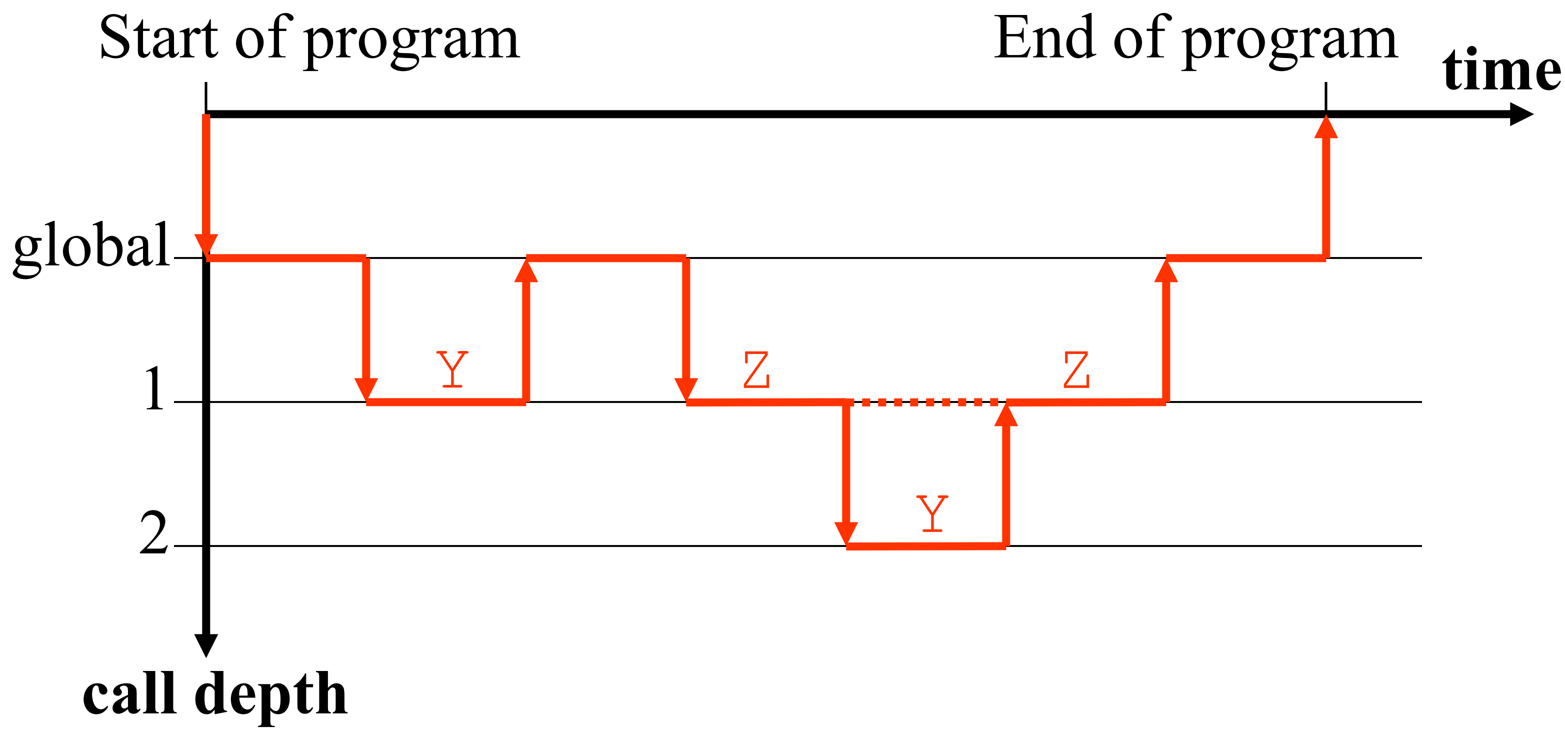
Example: When do the variables in this program "exist"

```
    void Y() {
        int d;
        ...  e;
         ... ;  }
    void Z() {
        int f;
        ...; Y(); ... }
int main(){
int[3] a;
  bool b;
  char c;
    ...; Y(); ...; Z(); }
```

- d, e: when procedure Y is active
- f: when procedure Z is active
- a, b, c: as long as the program is running

Stack Storage Allocation

A "picture" of our program running:



1) Procedure activation behaves like a stack (LIFO).

2) The local variables "live" as long as the procedure they are declared in.

1+2 => Allocation of locals on the "call stack" is a good model.

Recursion

```
int fact (int n) {
    if (n>1) return n* fact (n-1);
    else return 1;
}
```

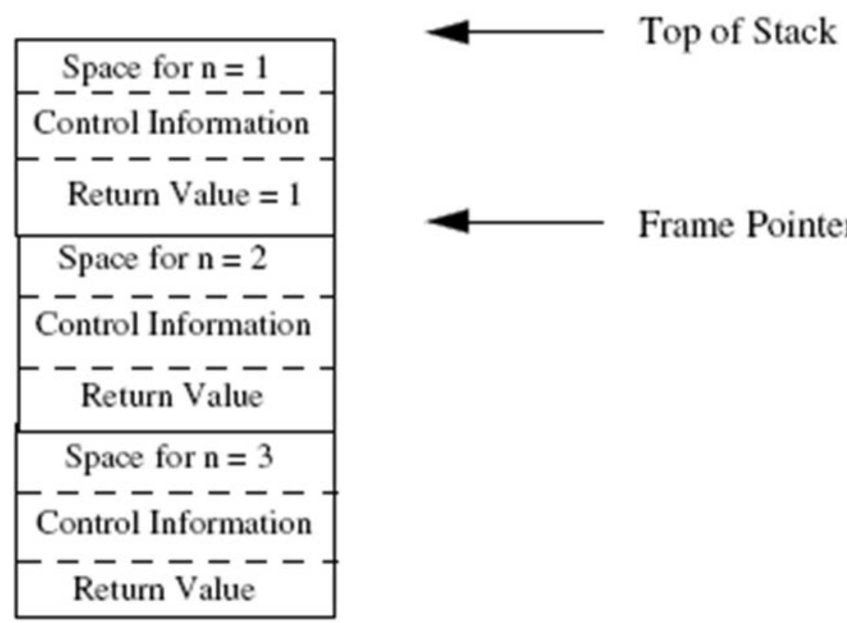
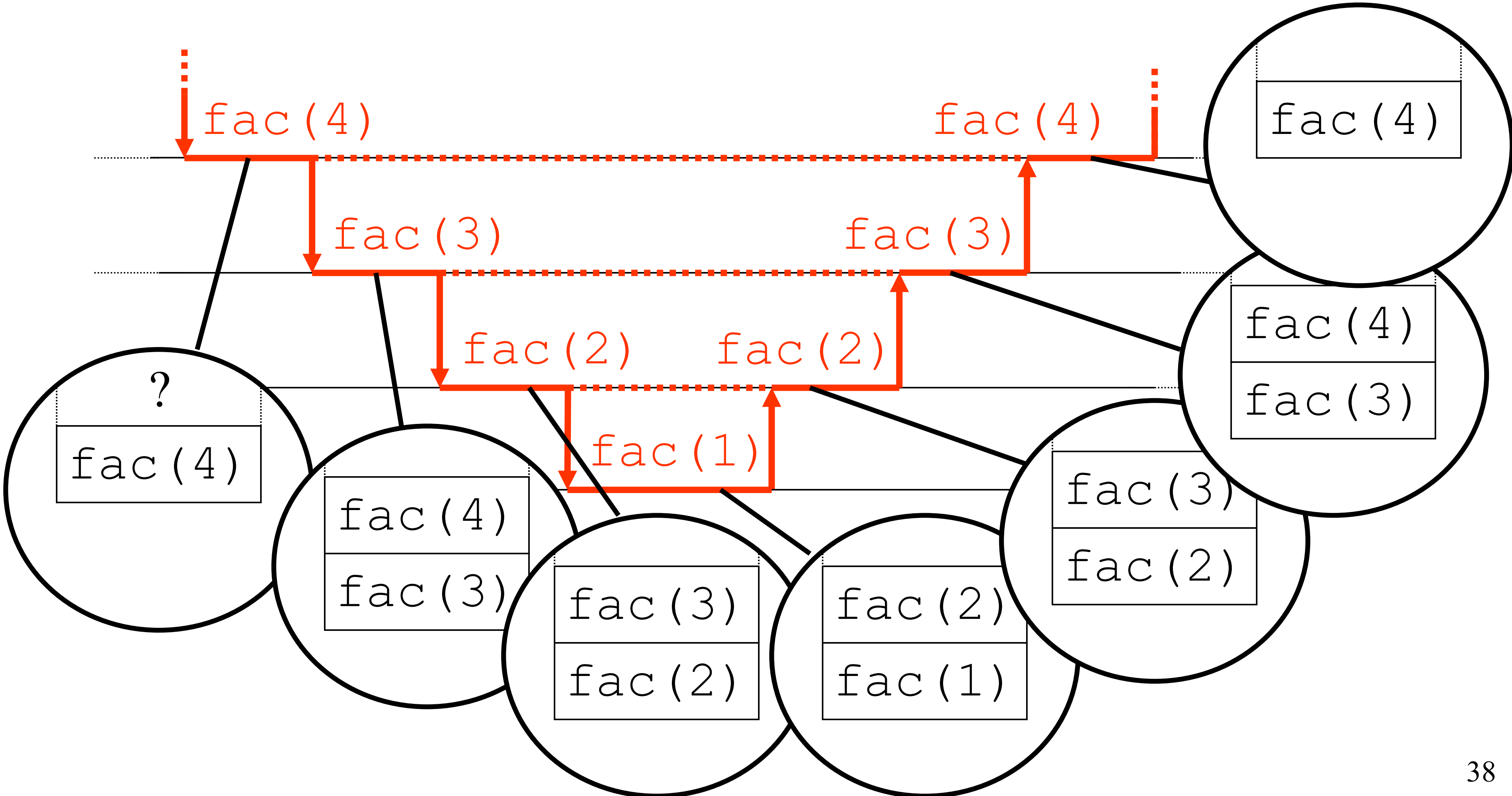


Figure 12.3: Runtime Stack for a Call of `fact(3)`

Recursion: General Idea

Why the stack allocation model works for recursion:
Like other function/procedure calls, lifetimes of local variables and parameters for recursive calls behave like a stack.



Dynamic link

Because stackframes may vary in size and because the stack may contain more than just frames (e.g., registers saved across calls), dynamic link is used to point to the preceding frame (Fig. 12.4).

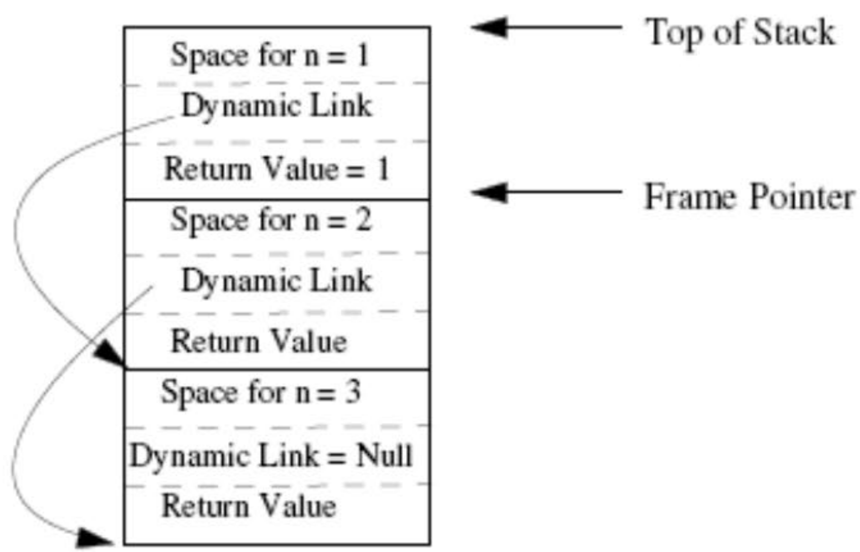


Figure 12.4: Runtime Stack for a Call of `fact(3)`with Dynamic Links

Nested functions/procedures

```
int p (int a) {
    int q (int b) { if (b <0) q (-b)  else return a+b; }
    return q (-10);
}
```

Methods cannot nest in C, Java, but in languages like Pascal, ML and Python they can. How to keep track of static block structure as above?

A static link points to the frame of the method that statically encloses the current method. (Fig. 12.6)

An alternative to using static links to access frames of enclosing methods is the use of a display. Here, we maintain a set of registers which comprise the display. (see Fig. 12,7)

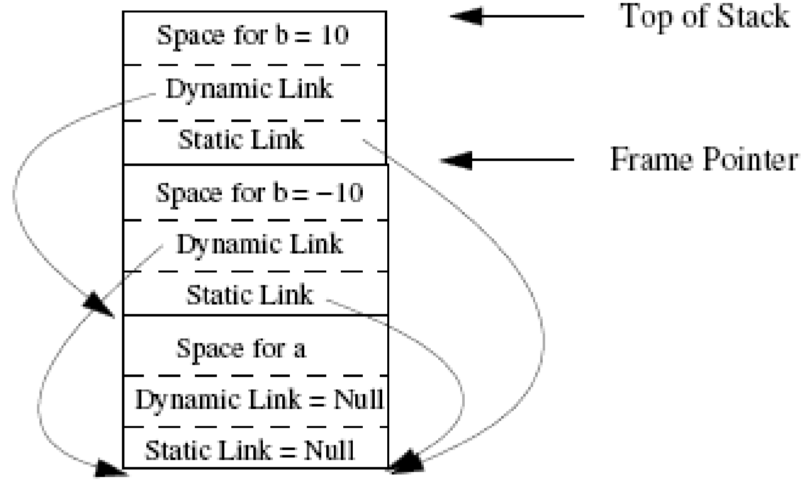


Figure 12.6: An Example of Static Links

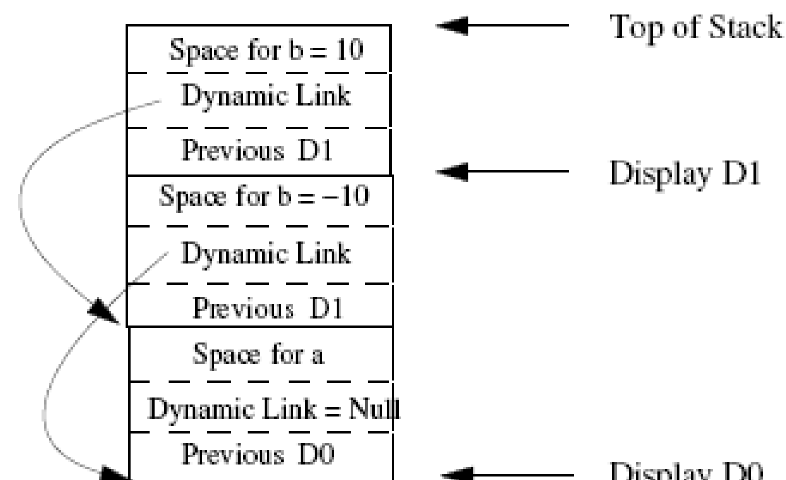


Figure 12.7: An Example of Display Registers

Blocks

```
void p (int a) {
   int b;
   if (a>0) {float c,d;  //body of block 1//}
   else     {int e[10]; //body of block 2//}
}
```

We could view such blocks as parameter-less procedures and thus use procedure-level-frames to implement blocks, but because the then and else parts of the if statement above are mutually exclusive, variables in block 1 and block 2 can overlay each other. This is called **block-level frame**, as contrasted with **procedure-level frame** allocation. (Fig. 12.8)

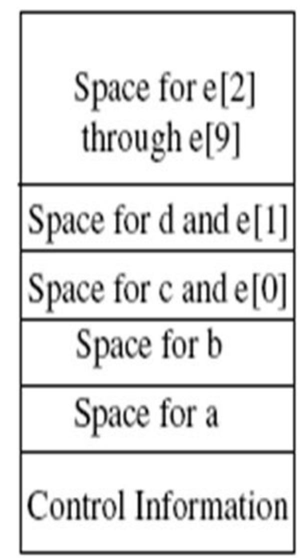


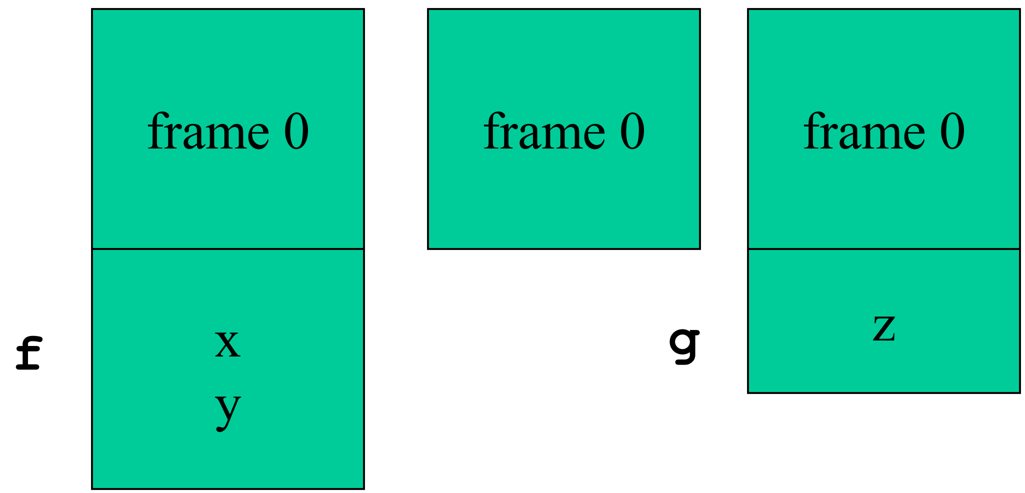
Figure 12.8: An Example of a Procedure-Level Frame

Higher-order functions

- Functions as values (first-class)
  - Pass as arguments
  - Return as values
  - Stored into data structures
- Implementation:
  - A code pointer, (i.e., a code address + an environment pointer)
  - Such a data structure is called a closure

Higher-order Nested Functions

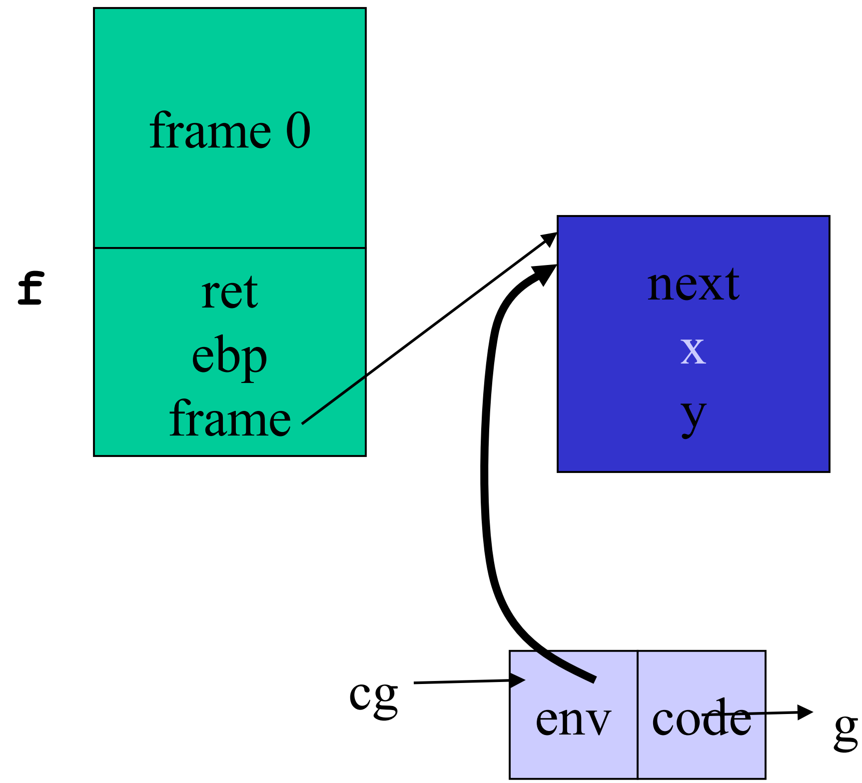
```
void->int f(){
  int x;
  int y;
  int g (){
    int z;
    return z+x;
  }
  return g;
}

h = f();  // h==g
h();      // g()
```



Function frames don't obey LIFO discipline any more. What one need to do is to keep frames live long enough! Heap-allocation!

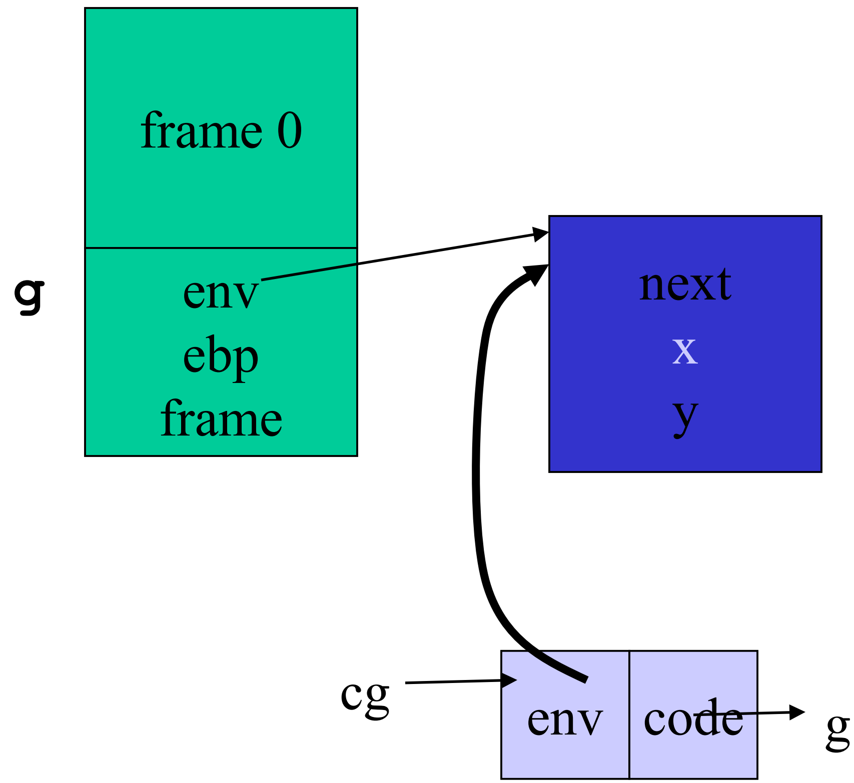
Heap-allocated Frames

```
void->int f(){
   int x;
   int y;
   int g (){
      int z;
      return z+x;
   }
   return g;
}

h = f();
// h == cg
h();
```



Heap-allocated Frames

```
void->int f(){
   int x;
   int y;
   int g (){
      int z;
      return z+x;
   }
   return g;
}
```




```
h = f();
// h == cg
h();
```

h->code(h->env);

Pause

Memory Management

When a program is started, most operating systems allocate 3 memory segments for it:

1) **code segment: read-only**

   Code (normally doesn't change during execution)

   Global variables (sometimes stored at the bottom of the stack)

2) **stack segment (data)**:

   manipulated by machine instructions.

   local variables and arguments for procedures and functions

   lifetime follows procedure activation

3) **heap segment (data)**:

   manipulated by the programmer.

   some programs may ask for and get memory allocated on arbitrary points during execution

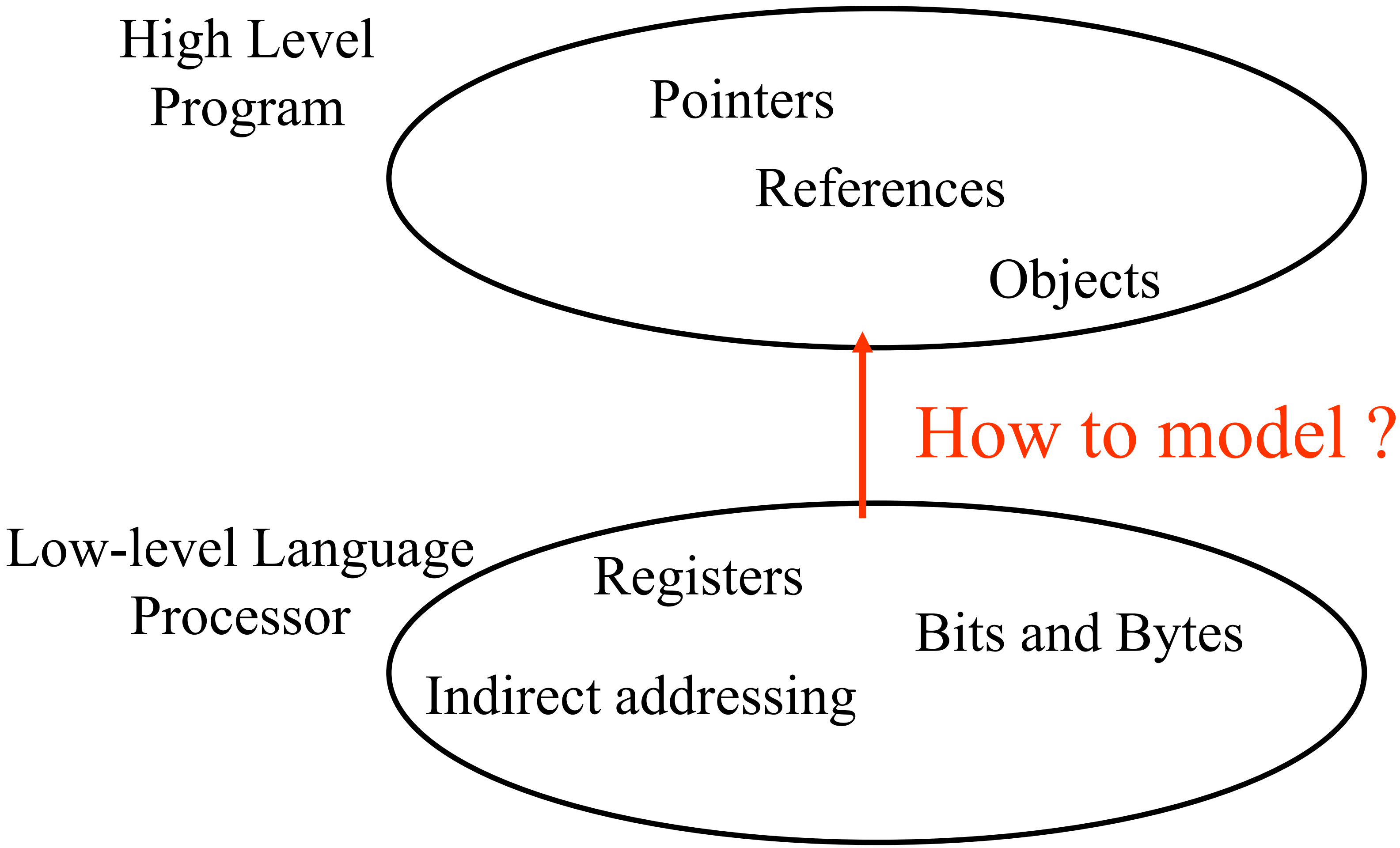
   When this memory is no longer used it should be freed

Heap Storage

- Memory allocation under explicit programmatic control
  - C malloc, C++ / Pascal / Java / C# new operation.
- Memory allocation implicit in language constructs
  - Lisp, Scheme, Haskell, SML, … most functional languages
  - Autoboxing/unboxing in Java 1.5 and C#
- Deallocation under explicit programmatic control
  - C, C++, Pascal (free, delete, dispose operations)
- Deallocation implicit
  - Java, C#, Lisp, Scheme, Haskell, SML, …

Data representation
Sometimes it goes the other way round

How to reflect low-level memory and machine data structures in terms of high-level computational structures.



How does things become garbage?

```
int *p, *q;
…
p = malloc(sizeof(int));
p = q;
```

Newly created space becomes garbage

```
for(int i=0;i<10000;i++){
    SomeClass obj= new SomeClass(i);
    System.out.println(obj);
}
```

Creates 10000 objects, which becomes garbage just after the print

Problem with explicit heap management

```
int *p, *q;
…
p = malloc(sizeof(int));
q = p;
free(p);
```

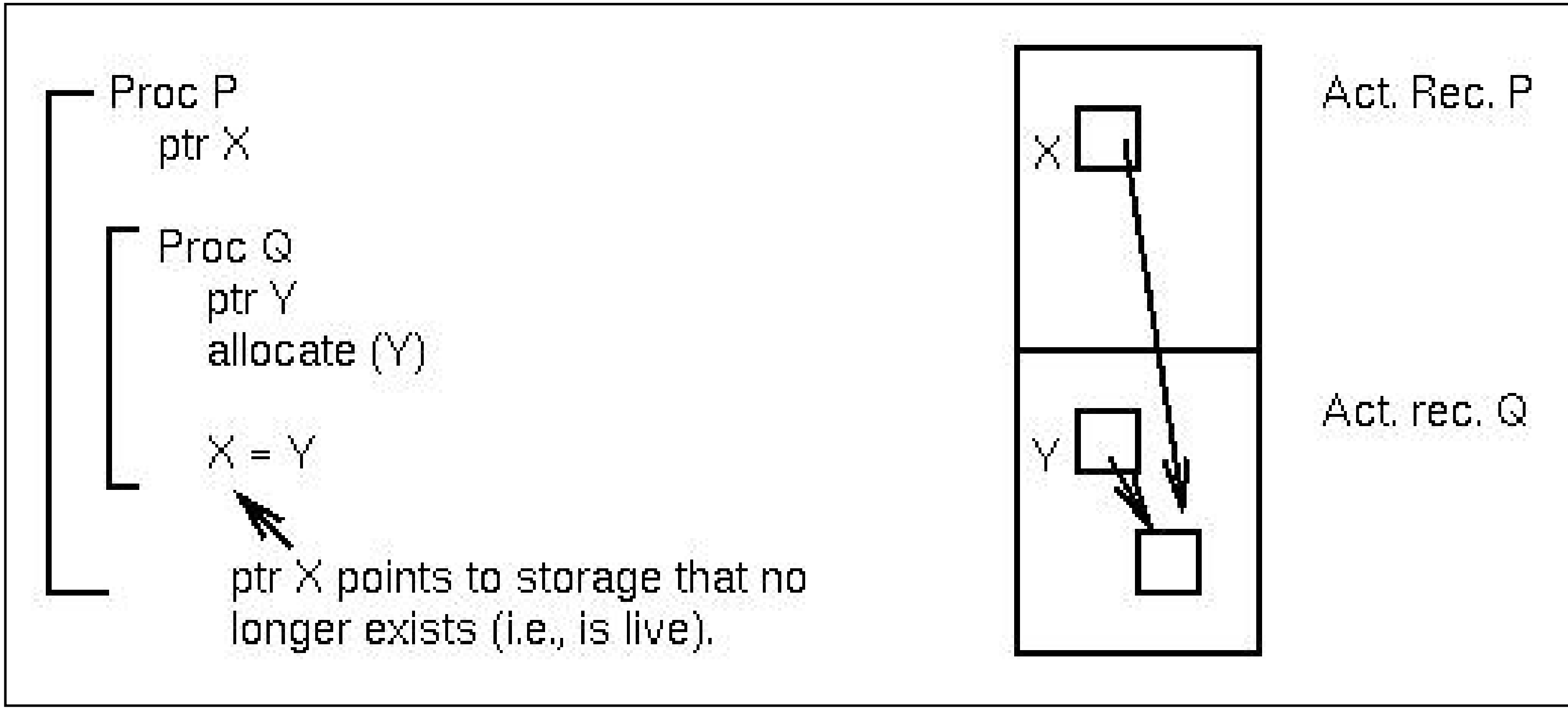
Dangling pointer in q now

```
float myArray[100];

p = myArray;
*(p+i) = …   //equivalent to myArray[i]
```

They can be hard to recognize

Stacks and dynamic allocations are incompatible

Why can't we just do dynamic allocation within the stack?



Where to put the heap?

- The heap is an area of memory which is dynamically allocated.
- Like a stack, it may grow and shrink during runtime.
- Unlike a stack it is not a LIFO => more complicated to manage
- In a typical programming language implementation we will have both heap-allocated and stack allocated memory coexisting.

Q: How do we allocate memory for both

Where to put the heap?

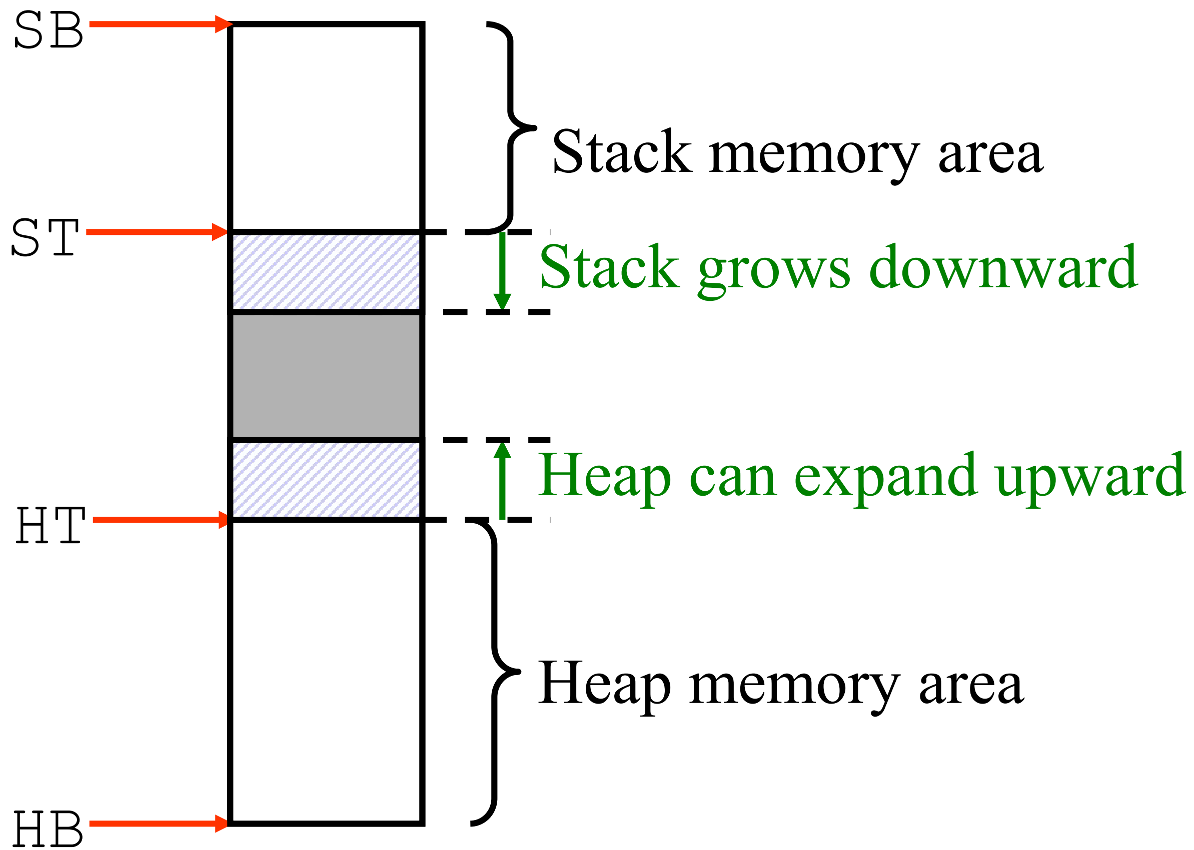
- A simple approach is to divide the available memory at the start of the program into two areas: stack and heap.
- Another question then arises
  - How do we decide what portion to allocate for stack vs. heap ?
  - Issue: if one of the areas is full, then even though we still have more memory (in the other area) we will get out-of-memory errors

Q: Isn’t there a better way?

Where to put the heap?

Q: Isn't there a better way?

A: Yes, there is an often used "trick": let both stack and heap share the same memory area, but grow towards each other from opposite ends!



Implicit memory management

- Current trend of modern programming language development: to give only implicit means of memory management to a programmer:
  - The constant increase of hardware memory justifies the policy of automatic memory management
  - The explicit memory management distracts programmer from his primary tasks: let everyone do what is required of them and nothing else!
  - The philosophy of high-level languages conforms to the implicit memory management
- Other arguments for implicit memory management:
  - Anyway, a programmer cannot control memory management for temporary variables!
  - The difficulties of combination of two memory management mechanisms: system and the programmer's
- The history repeats: in 70's people thought that the implicit memory management had finally replaced all other mechanisms

Automatic Storage Deallocation (Garbage Collection)

Everybody probably knows what a garbage collector is.

But here are two "one liners" to make you think again about what a garbage collector really is!

1) Garbage collection provides the "illusion of infinite memory"!

2) A garbage collector predicts the future!

It's a kind of magic! :-)

Let us look at how this magic is done!

Types of garbage collectors

- The "Classic" algorithms
  - Reference counting
  - Mark and sweep
- Copying garbage collection
- Generational garbage collection
- Incremental Tracing garbage collection

- **Direct Garbage Collectors:** a record is associated with each node in the heap. The record for node *N* indicates how many other nodes or roots point to *N*.

- **Indirect/Tracing Garbage Collectors:** usually invoked when a user's request for memory fails. The garbage collector visits all live nodes, and returns all other memory to the free list. If sufficient memory has been recovered from this process, the user's request for memory is satisfied.

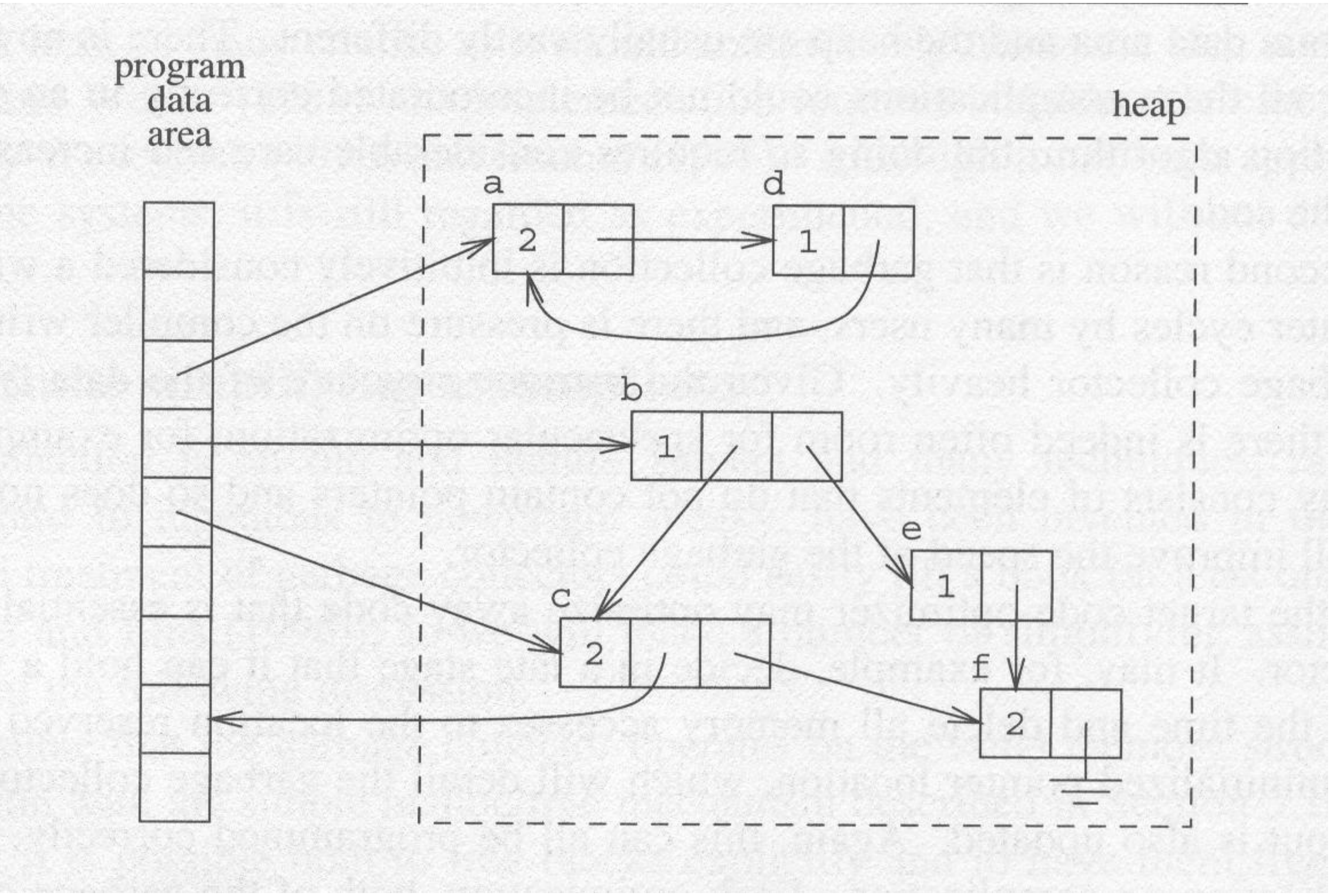
Terminology

- **Roots:** values that a program can manipulate directly (i.e. values held in registers, on the program stack, and global variables.)
- **Node/Cell/Object:** an individually allocated piece of data in the heap.
- **Children Nodes:** the list of pointers that a given node contains.
- **Live Node:** a node whose address is held in a root or is the child of a live node.
- **Garbage:** nodes that are not live, but are not free either.
- **Garbage collection:** the task of recovering (freeing) garbage nodes.
- **Mutator:** The program running alongside the garbage collection system.

Reference Counting

- Every cell has an additional field: the *reference count*. This field represents the number of pointers to that cell from roots or heap cells.
- Initially, all cells in the heap are placed in a pool of free cells, the *free list*.
- When a cell is allocated from the *free list*, its reference count is set to one.
- When a pointer is set to reference a cell, the cell's reference count is incremented by 1; if a pointer is to the cell is deleted, its reference count is decremented by 1.
- When a cell's reference count reaches 0, its pointers to its children are deleted and it is returned to the free list.

Reference Counting



Reference Counting: Advantages and Disadvantages

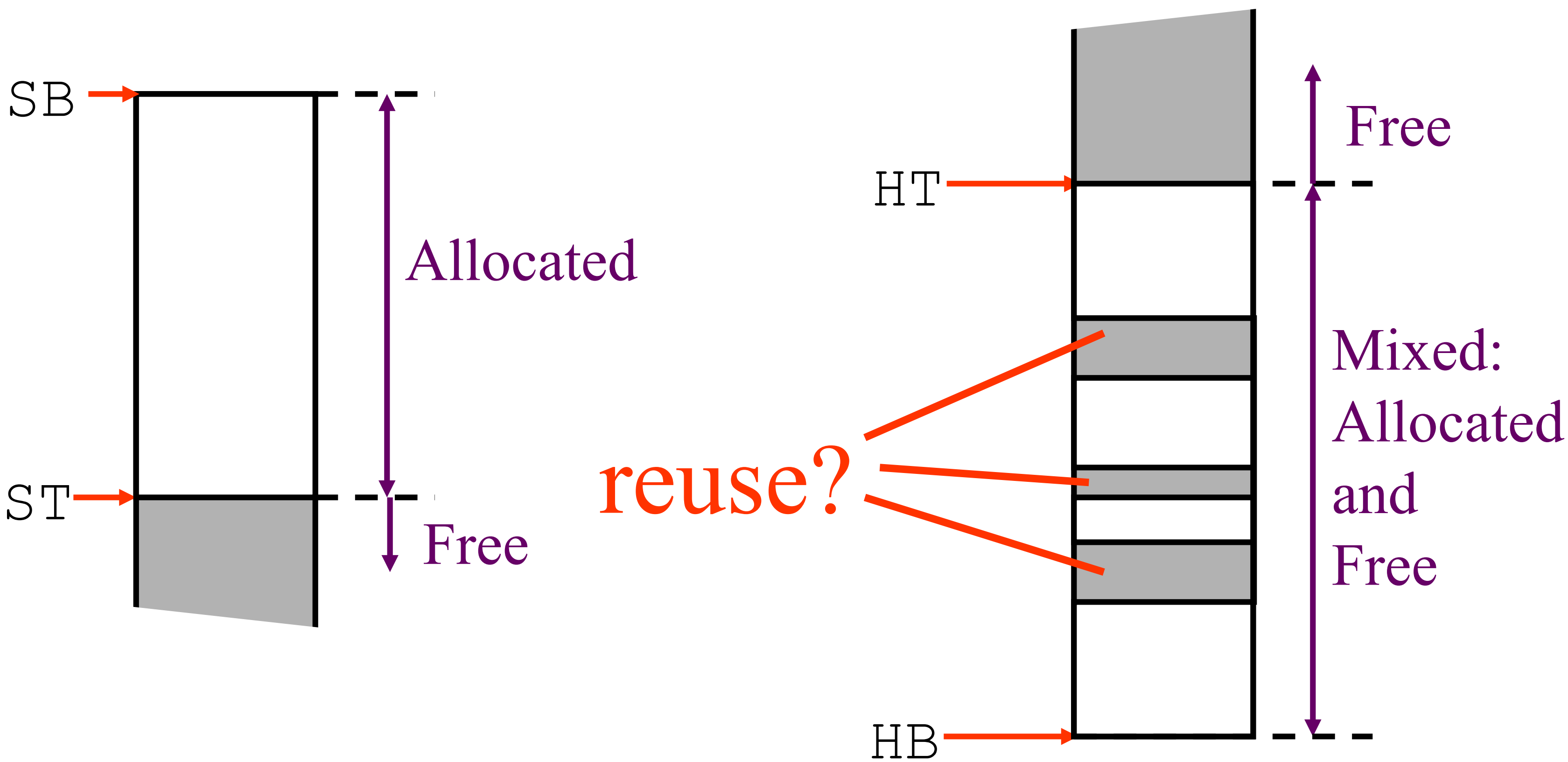
- Advantages:
  - Garbage collection overhead is distributed.
  - Locality of reference is no worse than mutator.
  - Free memory is returned to free list quickly.
- Disadvantages:
  - High time cost (every time a pointer is changed, reference counts must be updated).
    - In place of a single assignment x.f = p:

```
z = x.f
c = z.count
c = c – 1
z.count = c
If c = 0 call putOnFreeList(z)
x.f = p
c = p.count
c = c + 1
p.count = c
```

  - Storage overhead for reference counter can be high.
  - If the reference counter overflows, the object becomes permanent.
  - Unable to reclaim cyclic data structures.

How to keep track of free memory?

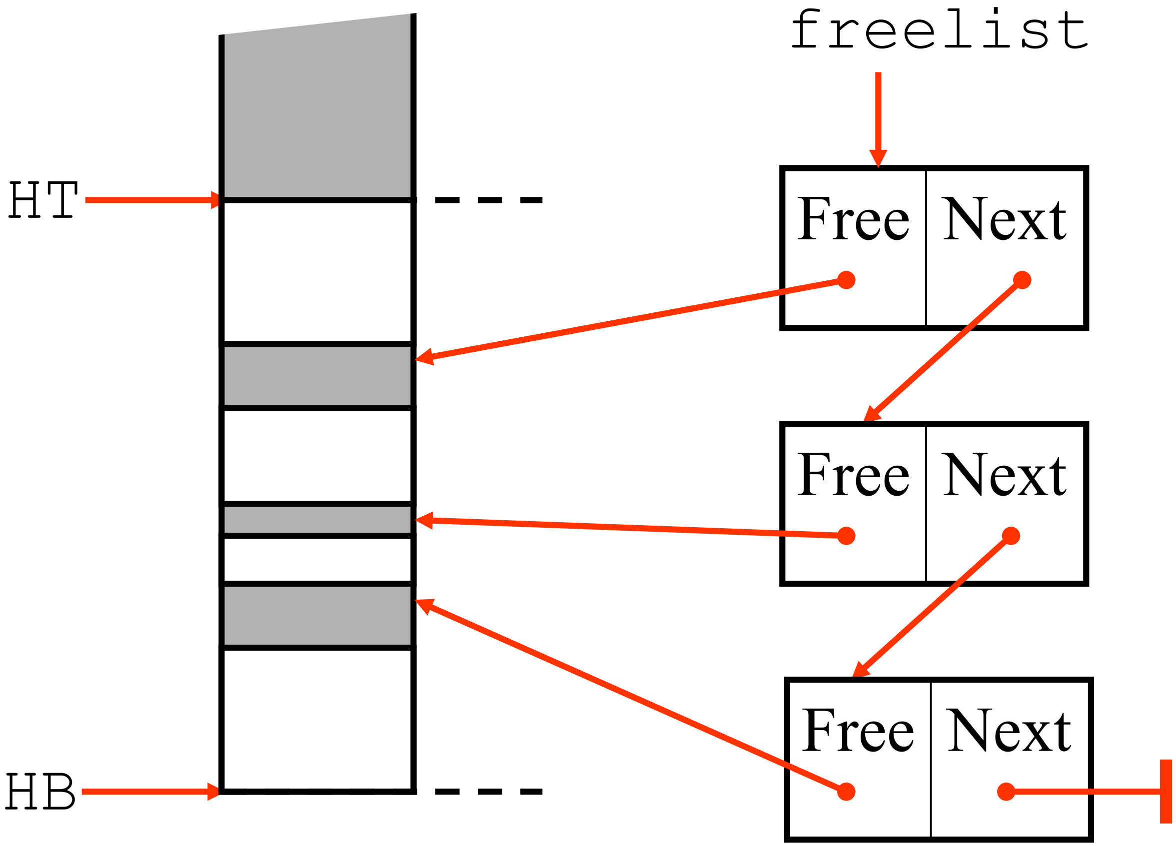
Stack is LIFO allocation => ST moves up/down everything above ST is in use/allocated. Below is free memory. This is easy! But …
Heap is not LIFO, how to manage free space in the "middle" of the heap?



How to keep track of free memory?

How to manage free space in the “middle” of the heap?

=> keep track of free blocks in a data structure: the “free list”. For example we could use a linked list pointing to free blocks.

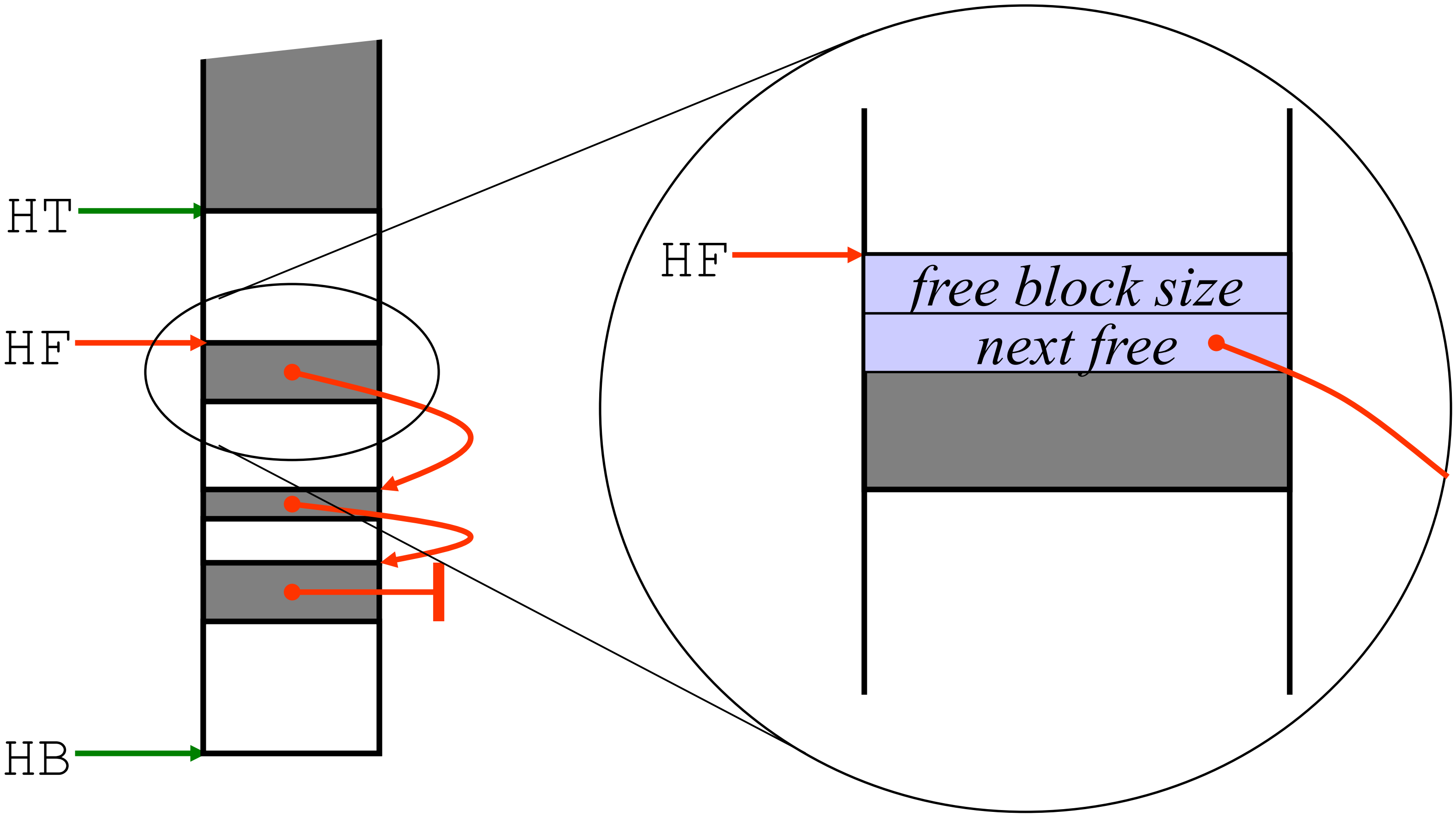


A freelist!
Good idea!

But where do we find the memory to store this data structure?

How to keep track of free memory?

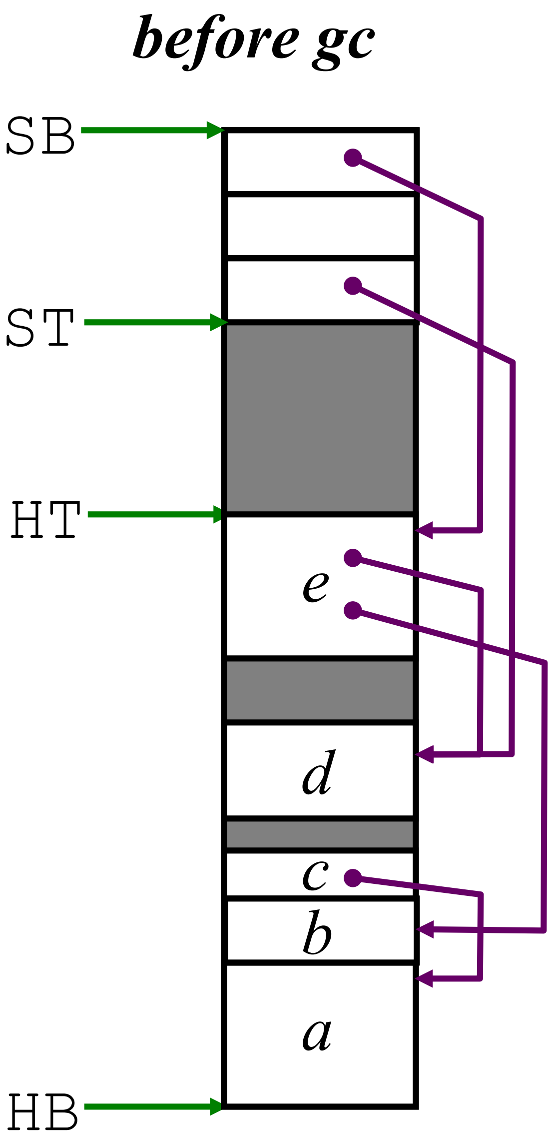
Q: Where do we find the memory to store a freelist data structure?
A: Since the free blocks are not used for anything by the program => memory manager can use them for storing the freelist itself.

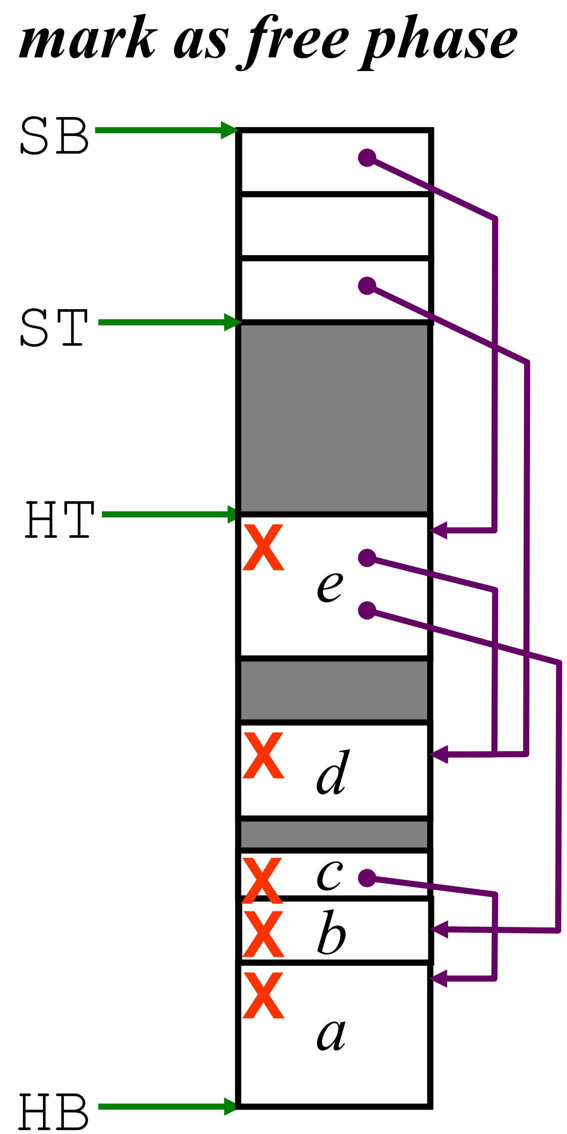


Mark-Sweep

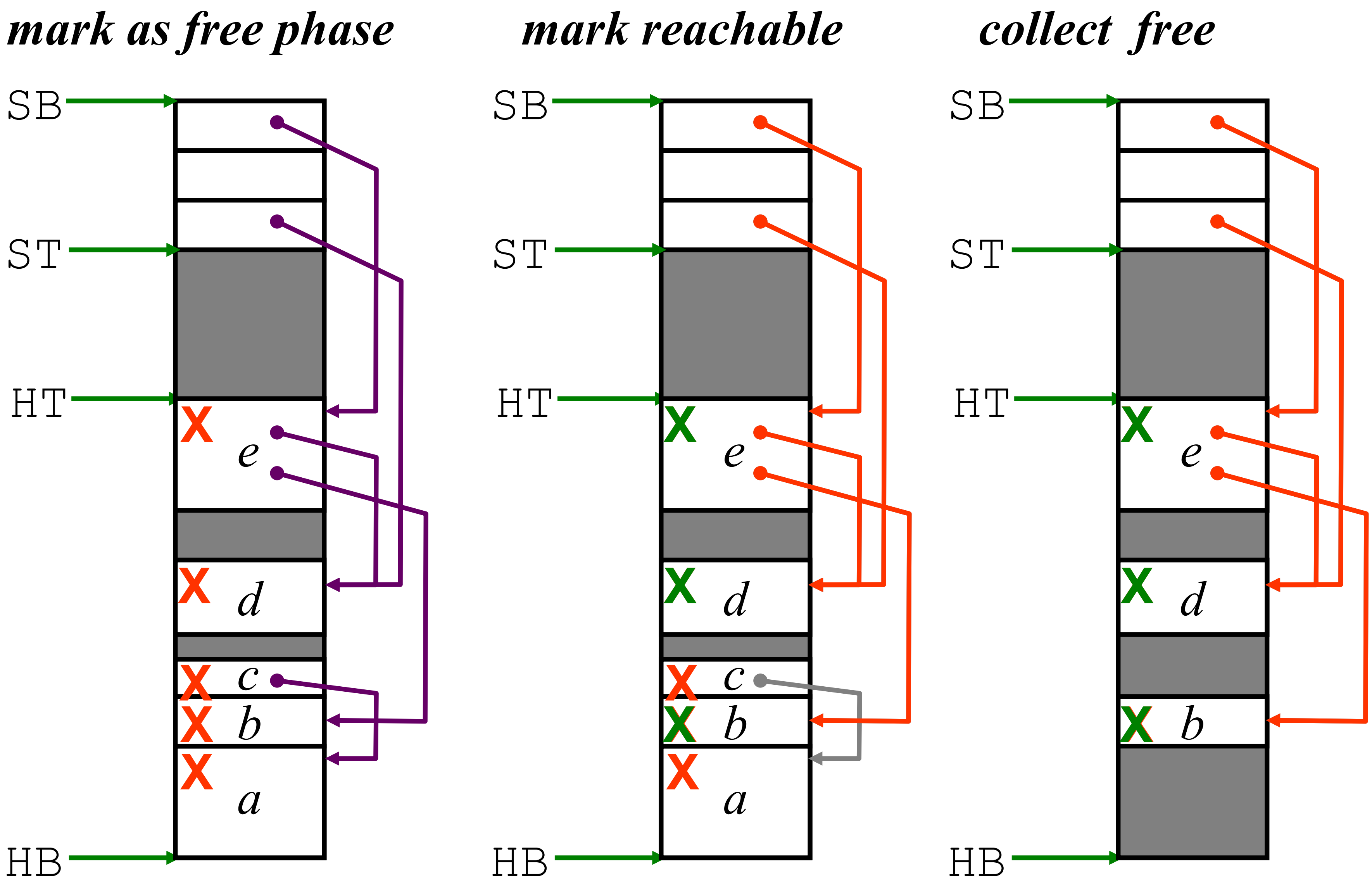
- The first tracing garbage collection algorithm
- Garbage cells are allowed to build up until heap space is exhausted (i.e. a user program requests a memory allocation, but there is insufficient free space on the heap to satisfy the request.)
- At this point, the mark-sweep algorithm is invoked, and garbage cells are returned to the free list.
- Performed in two phases:
  - **Mark:** identifies all live cells by setting a mark bit. Live cells are cells reachable from a root.
  - **Sweep:** returns garbage cells to the free list.

Mark and Sweep Garbage Collection





Mark and Sweep Garbage Collection



Mark and Sweep Garbage Collection

Algorithm pseudo code:

```
void garbageCollect() {
    mark all heap variables as free
    for each frame in the stack
        scan(frame)
    for each heapvar (still) marked as free
        add heapvar to freelist
}
void scan(region) {
    for each pointer p in region
        if p  points to region marked as free then
            mark region at p as reachable
            scan(region at p )
}
```

Q: This algorithm is recursive. What do you think about that?

Mark-Sweep:
Advantages and Disadvantages

- Advantages:
  - Cyclic data structures can be recovered.
  - Tends to be faster than reference counting.
- Disadvantages:
  - Computation must be halted while garbage collection is being performed
  - Every live cell must be visited in the mark phase, and every cell in the heap must be visited in the sweep phase.
  - Garbage collection becomes more frequent as residency of a program increases.
  - May fragment memory.

Mark-Sweep-Compact: Advantages and Disadvantages

- Advantages:
  - The contiguous free area eliminates fragmentation problem. Allocating objects of various sizes is simple.
  - The garbage space is "squeezed out", without disturbing the original ordering of objects. This improves locality.
- Disadvantages:
  - Requires several passes over the data are required. "Sliding compactors" takes two, three or more passes over the live objects.
    - One pass computes the new location
    - Subsequent passes update the pointers to refer to new locations, and actually move the objects

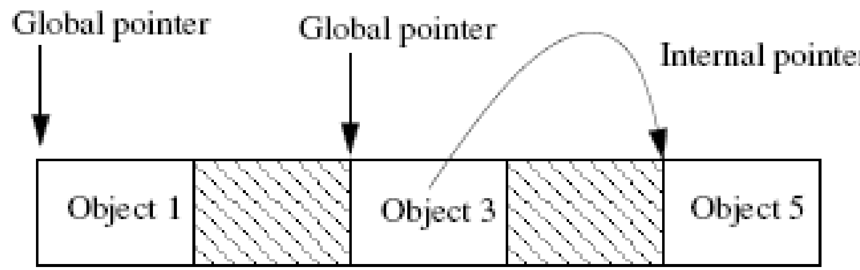


Figure 12.16: Mark-Sweep Garbage Collection

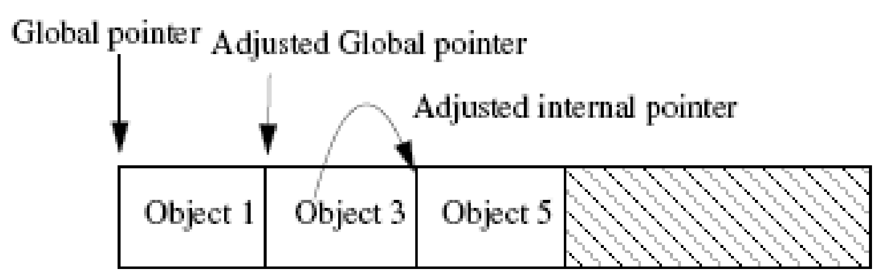


Figure 12.17: Mark-Sweep Garbage Collection with Compaction

Copying Garbage Collection (Cheney's algorithm)

- Like mark-compact, copying garbage collection, but does not really "collect" garbage.

- The heap is subdivided into two contiguous subspaces
  - (FromSpace and ToSpace).
- During normal program execution, only one of these semispaces is in use.
- When the garbage collector is called, all the live data are copied from the current semispace (FromSpace) to the other semispace (ToSpace), so that objects need only be traversed once.

- The work needed is proportional to the amount of live data (all of which must be copied).

Semispace Collector Using the Cheney Algorithm

- The heap is subdivided into two contiguous subspaces (*FromSpace* and *ToSpace*).
- During normal program execution, only one of these semispaces is in use.
- When the garbage collector is called, all the live data are copied from the current semispace (*FromSpace*) to the other semispace (*ToSpace*).

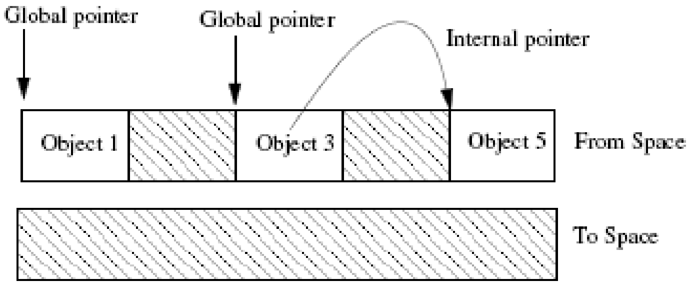


Figure 12.18: Copying Garbage Collection (a)

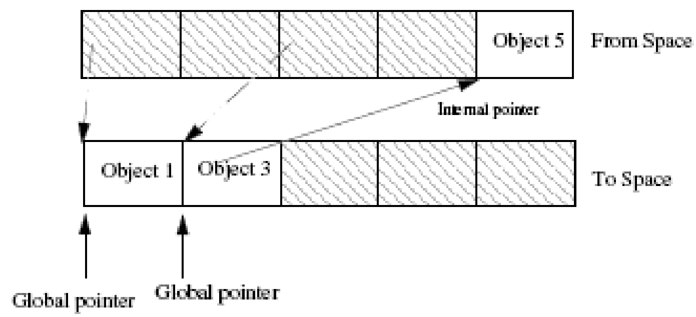


Figure 12.19: Copying Garbage Collection (b)

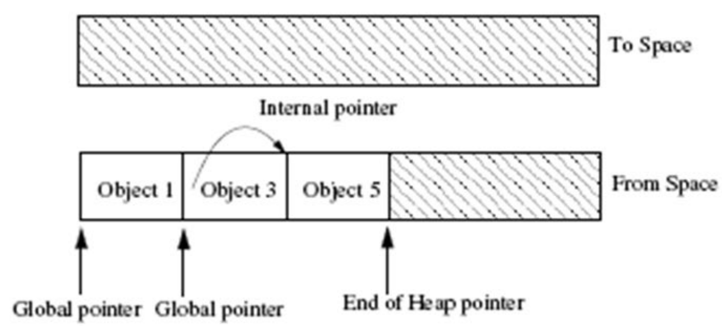


Figure 12.20: Copying Garbage Collection (c)

Copying Garbage Collection:
Advantages and Disadvantages

- Advantages:
  - Allocation is extremely cheap.
  - Excellent asymptotic complexity.
  - Fragmentation is eliminated.
  - Only one pass through the data is required.

- Disadvantages:
  - The use of two semi-spaces doubles memory requirement
  - Poor locality. Using virtual memory will cause excessive paging.

Problems with Simple Tracing Collectors

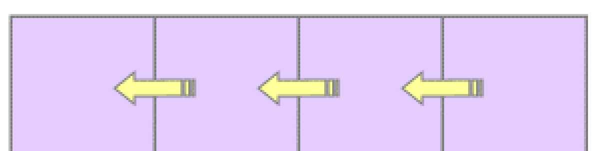
- Difficult to achieve high efficiency in a simple garbage collector, because large amounts of memory are expensive.
- If virtual memory is used, the poor locality of the allocation/reclamation cycle will cause excessive paging.
- Even as main memory becomes steadily cheaper, locality within cache memory becomes increasingly important.

Generational Garbage Collection

- Attempts to address weaknesses of simple tracing collectors such as mark-sweep and copying collectors:
  - All active data must be marked or copied.
  - For copying collectors, each page of the heap is touched every two collection cycles, even though the user program is only using half the heap, leading to poor cache behavior and page faults.
  - Long-lived objects are handled inefficiently.
- Generational garbage collection is based on the *generational hypothesis*:

Most objects die young.

- As such, concentrate garbage collection efforts on objects likely to be garbage: young objects.



Generational Garbage Collection: Multiple Generations

- Advantages:
  - Keeps youngest generation's size small.
  - Helps address mistakes made by the promotion policy by creating more intermediate generations that still get garbage collected fairly frequently.
- Disadvantages:
  - Collections for intermediate generations may be disruptive.
  - Tends to increase number of inter-generational pointers, increasing the size of the root set for younger generations.
- Performs poorly if any of the main assumptions are false:
  - That objects tend to die young.
  - That there are relatively few pointers from old objects to young ones.

Incremental Tracing Collectors

- Program (Mutator) and Garbage Collector run concurrently.
  - Can think of system as similar to two threads. One performs collection, and the other represents the regular program in execution.
- Can be used in systems with real-time requirements. For example, process control systems.
  - allow mutator to do its job without destroying collector's possibilities for keeping track of modifications of the object graph, and at the same time
  - allowing collector to do its job without interfering with mutator

Garbage Collection: Summary

Tracing: Mark Sweep, Mark Compact, Copying

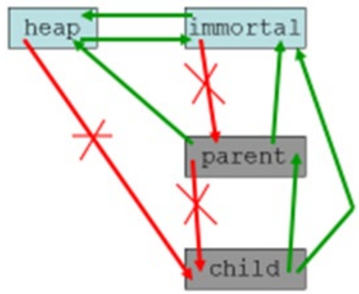
Incremental: Reference Counting, Deferred Reference Counting, Incremental, Generational

Method	Conservatism	Space	Time	Fragmentation	Locality
Mark Sweep	Major	Basic	1 traversal + heap scan	Yes	Fair
Mark Compact	Major	Basic	Many passes of heap	No	Good
Copying	Major	Two Semispaces	1 traversal	No	Poor
Reference Counting	No	Reference count field	Constant per Assignment	Yes	Very Good
Deferred Reference Counting	Only for stack variables	Reference Count Field	Constant per Assignment	Yes	Very Good
Incremental	Varies depending on algorithm	Varies	Can be Guaranteed Real-Time	Varies	Varies
Generational	Variable	Segregated Areas	Varies with number of live objects in new generation	Yes (Non-Copying) No (Copying)	Good

Different choices for different reasons

- JVM
  - Sun Classic: Mark, Sweep and Compact
  - SUN HotSpot: Generational (two generation + Eden)
    - -Xincgc an incremental collector that breaks that old-object region into smaller chunks and GCs them individually
    - -Xconcgc Concurrent GC allows other threads to keep running in parallel with the GC
  - BEA jRockit JVM: concurrent, even on another processor
  - IBM: Improved Concurrent Mark, Sweep and Compact with a notion of weak references
  - Real-Time Java
    - Scoped LTMemory, VTMemory, RawMemory
- .Net CLR
  - Managed and unmanaged memory (memory blob)
  - PC version: Self-tuning Generation Garbage Collector
  - .Net CF: Mark, Sweep and Compact

RTSJ Scoped Memory



- Scopes have fixed lifetimes
- Lifetime starts here:
  - `scopedMemArea.enter() { … }`
- Lifetime ends:
- All calls to new inside a scope, create an object inside of that scope
- When the scope's lifetime ends, all objects within are destroyed
- Scopes may be nested

Region Based memory management

- Compiler (especially Type inference) automatically detects scopes or regions
- May require programmer to annotate types
- May sometimes have worse behaviour than GC and heap

Summary of Storage Allocation

- Data Representation
  - Non-confusion and uniqueness
  - Direct vs. indirect
- Data Allocation
  - Static
  - Stack
    - Frames, dynamic and static links/display regs, closures
  - Heap
    - Manual vs. automatic
    - Garbage Collection
      - Different algorithms have pros and cons

Languages and Compilers (SProg og Oversættere)

Lecture 18
Low Level Code Generation

Bent Thomsen

Department of Computer Science

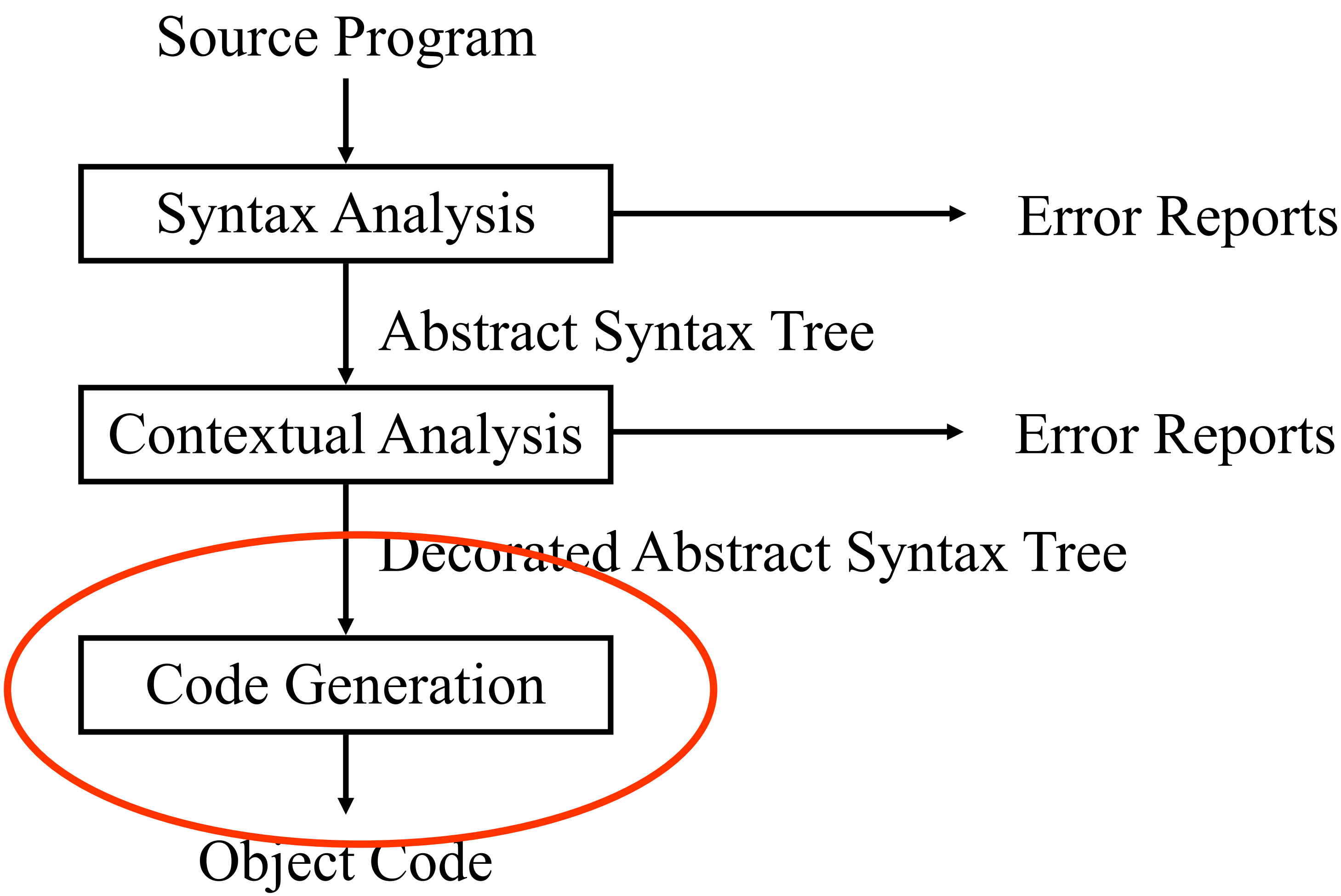
Aalborg University

Learning goals

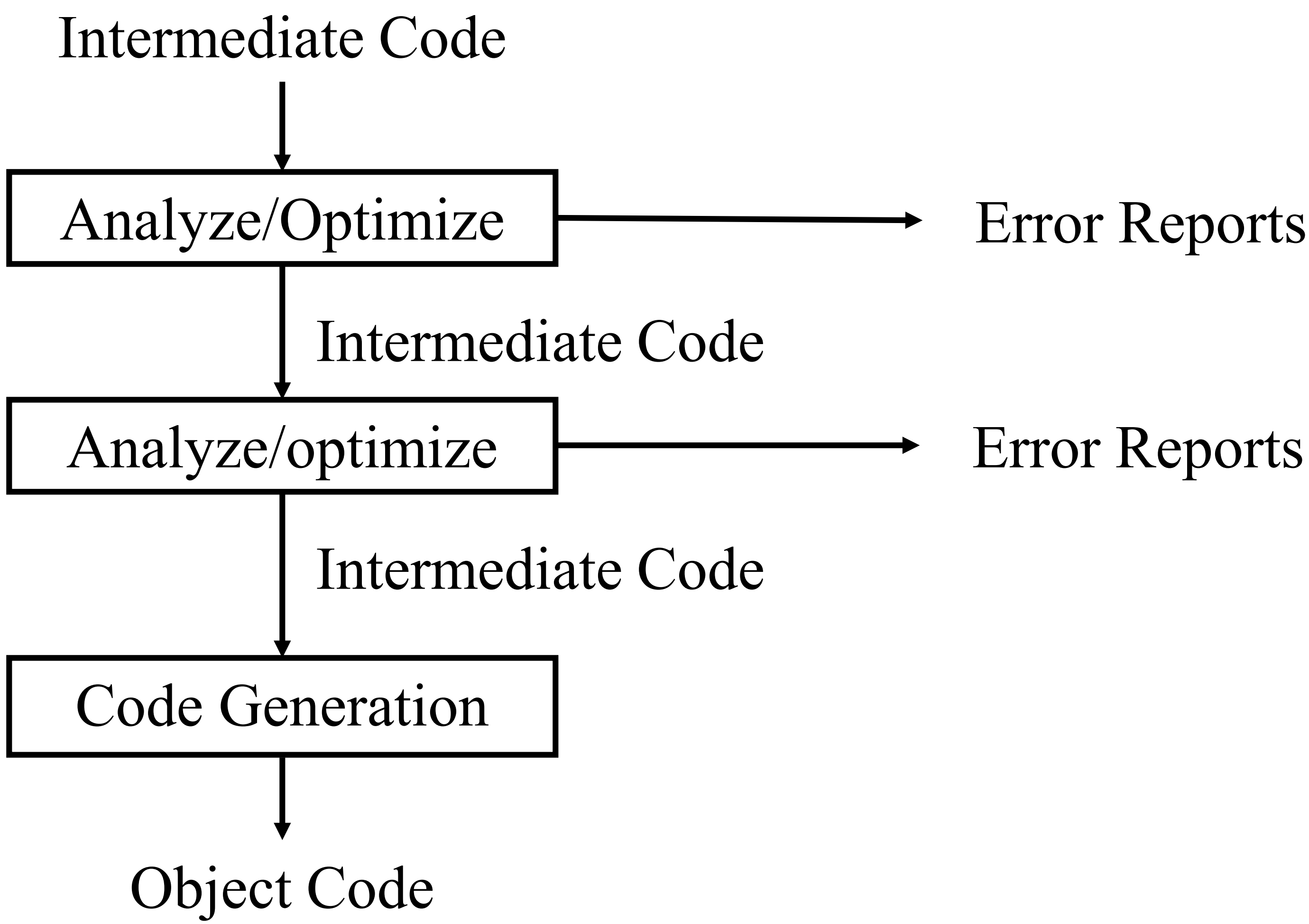
- Understand issues such as
  – code selection
  – storage allocation
  – register allocation
  – code scheduling

  for low level code generation.
- Understand different approaches to low level code generation:
  – Code generation from AST via visitor
  – Code generation by tree-rewrite and pattern matching
  – Code generation from IR

The "Phases" of a Compiler



The "Phases" of a Compiler



Intermediate Representations

- Abstract Syntax Tree
  - Convenient for semantic analysis phases
  - We can generate code directly from the AST, but...
  - What about multiple target architectures?
    - Remember n * m vs. n + m
- Intermediate Representation
  - "Neutral" architecture
  - Easy to translate to native code
  - Can abstracts away complicated runtime issues
    - Stack Frame Management
    - Memory Management
    - Register Allocation

Issues in Code Generation

- Code Selection:

  Deciding which sequence of target machine instructions will be used to implement each phrase in the source language.

- Storage Allocation

  Deciding the storage address for each variable in the source program. (static allocation, stack allocation etc.)

- Register Allocation (for register-based machines)

  How to use registers efficiently to store intermediate results.

- Code Scheduling

  The order in which the generated instructions are executed

Code generation from AST summary

- Idea from Brown & Watt
- Create code templates inductively
  - There may be special case templates generating equivalent, but more efficient code
  - Keep in mind what goes on at compile time
    - AST traversal order
  - Keep in mind what goes on at run time
    - Control flow order
- Use visitors (or composit or functional) pattern to walk the AST recursively emitting code as you go along
- Low level VM, called Triangle VM, with direct addressing and storage allocation

Developing a Code Generator "Visitor"

Phrase Class	visitor method	Behavior of the visitor method
Program	visitProgram	generate code as specified by *run*[P]
Command	visit…Command	generate code as specified by *execute*[C]
Expression	visit…Expression	generate code as specified by *evaluate*[E]
V-name	visit…Vname	Return "entity description" for the visited variable or constant name.
Declaration	visit…Declaration	generate code as specified by *elaborate*[D]
Type-Den	visit…TypeDen	return the size of the type

Example from Brown&Watt chapter 7, p. 260- 280, translating miniTriangle to TAM, a stack based VM with explicit addressing and storage allocation

Developing a Code Generator "Visitor"

evaluate [IL] =

LOADL *v* where *v* is the integer value of IL

```
/* Expressions */
public Object visitIntegerExpression (
            IntegerExpression expr,Object arg) {
   short v = valuation(expr.IL.spelling);
   emit(Instruction.LOADLop, 0, 0, v);
   return null;
}

public short valuation(String s) {
   ... convert string to integer value ...
}
```

Developing a Code Generator "Visitor"

evaluate [`E1 O E2`] =
  evaluate [`E1`]
  evaluate [`E2`]
  `CALL` *p*   where *p* is the address of routine for `O`

```
public Object visitBinaryExpression (
           BinaryExpression expr,Object arg) {
  expr.E1.visit(this,arg);
  expr.E2.visit(this,arg);
  short p = address for expr.O operation
  emit(Instruction.CALLop,
    Instruction.SBr,
    Instruction.PBr, p);
  return null;
}
```

Remaining expression visitors are developed in a similar way.

Developing a Code Generator "Visitor"

execute [V := E] =
  evaluate [E]
  assign [V]

```
/* Generating code for commands */

public Object visitAssignCommand(
                AssignCommand com,Object arg) {
   com.E.visit(this,arg);
   RuntimeEntity entity =
      (RuntimeEntity) com.V.visit(this,null);
   short d = entity.address;
   emit(Instruction.STOREop,Com.V.size,d);
   return null;
}
```

Developing a Code Generator "Visitor"

execute [C1 ; C2] =
  execute[C1]
  execute[C2]

```
public Object visitSequentialCommand(
                SequentialCommand com,Object arg) {
   com.C1.visit(this,arg);
   com.C2.visit(this,arg);
   return null;
}
```

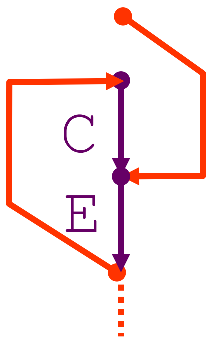
- IfCommand and WhileCommand: complications with jumps
- LetCommand is more complex: memory allocation and addresses

Control Structures

We have yet to discuss generation for IfCommand and WhileCommand

execute [**while** E **do** C] =

```
     JUMP h
  g: execute [C]
  h: evaluate[E]
     JUMPIF(1) g
```



A complication is the generation of the correct addresses for the jump instructions.

We can determine the address of the instructions by incrementing a counter while emitting instructions.

Backwards jumps are easy but forward jumps are harder.
Q: why?

Control Structures

Backwards jumps are easy:

The "address" of the target has already been generated and is known

Forward jumps are harder:

When the jump is generated the target is not yet generated so its address is not (yet) known.

There is a solution which is known as **backpatching**

1) Emit jump with "dummy" address (e.g. simply 0).
2) Remember the address where the jump instruction occurred.
3) When the target label is reached, go back and patch the jump instruction.

Backpatching Example

```
public Object WhileCommand (
              WhileCommand com,Object arg) {
   short j = nextInstrAddr;
   emit(Instruction.JUMPop, 0,
        Instruction.CBr,0);
   short g = nextInstrAddr;
   com.C.visit(this,arg);
   short h = nextInstrAddr;
   code[j].d = h;
   com.E.visit(this,arg);
   emit(Instruction.JUMPIFop, 1,
        Instruction.CBr,g);
   return null;
}
```

dummy address

backpatch

execute [**while** E **do** C] =
      JUMP *h*
   g: *execute* [C]
   h: *evaluate*[E]
      JUMPIF(1) *g*

Static Storage Allocation: In the Code Generator

```
public Object visit...Command(
    ...Command com, Object arg) {
    short gs = shortValueOf(arg);
    generate code as specified by execute[com]
    return null;
}
public Object visit...Expression(
    ...Expression expr, Object arg) {
    short gs = shortValueOf(arg);
    generate code as specified by evaluate[expr]
    return new Short(size of expr result);
}
public Object visit...Declaration(
    ...Declaration dec, Object arg) {
    short gs = shortValueOf(arg);
    generate code as specified by elaborate[dec]
    return new Short(amount of extra allocated by dec);
}
```

Routines

We call the assembly language equivalent of procedures "routines".

What are routines? Unlike procedures/functions in higher level languages. They are not directly supported by language constructs. Instead they are modeled in terms of how to use the low-level machine to "emulate" procedures.

What behavior needs to be "emulated"?

- Calling a routine and returning to the caller after completion.
- Passing arguments to a called routine
- Returning a result from a routine
- Local and non-local variables.

Code Generation for Procedures and Functions

We extend Mini Triangle with procedures:

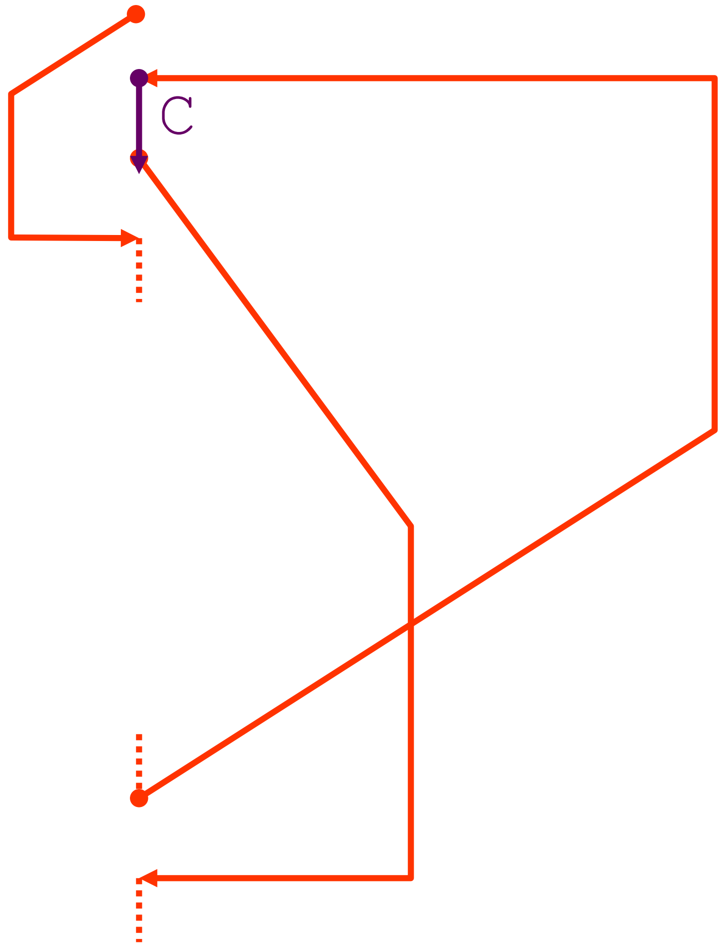
```
Declaration
   ::= ...
     | proc Identifier ( ) ~ Command
Command
   ::= ...
     | Identifier ( )
```

First , we will only consider global procedures (with no arguments).

Code Template: Global Procedure

elaborate [**proc** I () ~ C] =

```
    JUMP g
e:  execute [C]
    RETURN(0) 0
g:
```



execute [I ()] =

```
    CALL(SB) e
```

Routines

- Transferring control to and from routine:
  Most low-level processors have `CALL` and `RETURN` for transferring control from caller to callee and back.
- Transmitting arguments and return values:
  Caller and callee must agree on a method to transfer argument and return values.
  => This is called the "routine protocol"

!

> There are many possible ways to pass argument and return values.
> => A routine protocol is like a **"contract" between the caller and the callee**.

The routine protocol is often dictated by the operating system.

Routine Protocol Examples

The routine protocol depends on the machine architecture (e.g. stack machine versus register machine).

Example 1: A possible routine protocol for a RM
- Passing of arguments:
  first argument in R1, second argument in R2, etc.
- Passing of return value:
  return the result (if any) in R0

Note: this example is simplistic:
- What if more arguments than registers?
- What if the representation of an argument is larger than can be stored in a register.

For RM protocols, the protocol usually also specifies who (caller or callee) is responsible for saving contents of registers.

Routine Protocol Examples

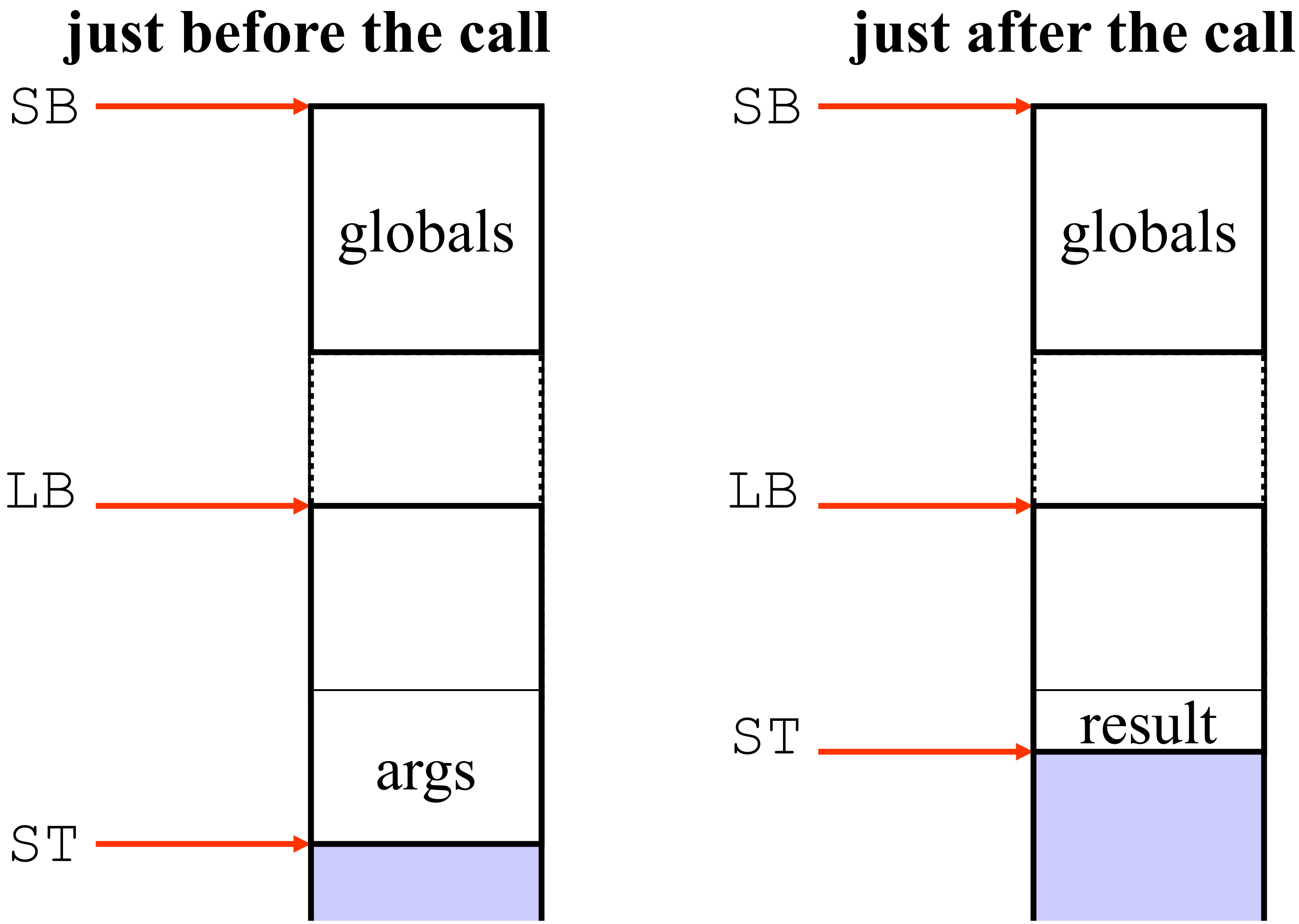
Example 2: A possible routine protocol for a stack machine

- Passing of arguments:
  pass arguments on the top of the stack.
- Passing of return value:
  leave the return value on the stack top, in place of the arguments.

Note: this protocol puts no boundary on the number of arguments and the size of the arguments.

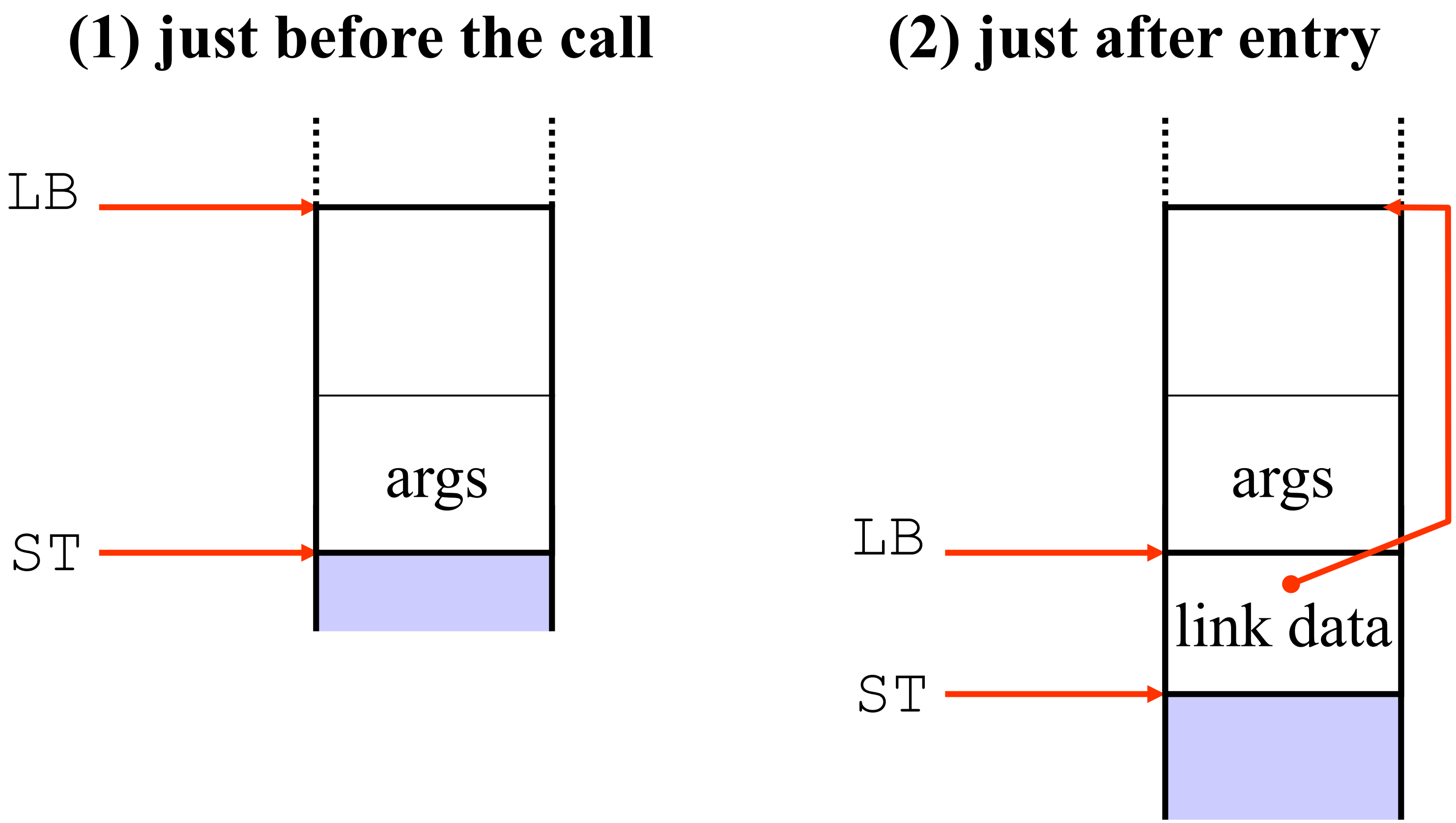
Most micro-processors, have registers as well as a stack. Such "mixed" machines also often use a protocol like this one.

Routine Protocol



What happens in between?

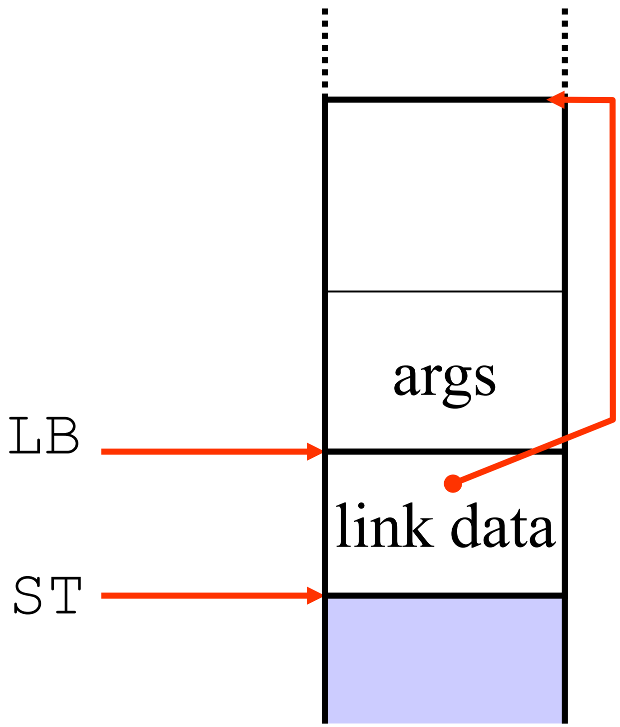
Routine Protocol



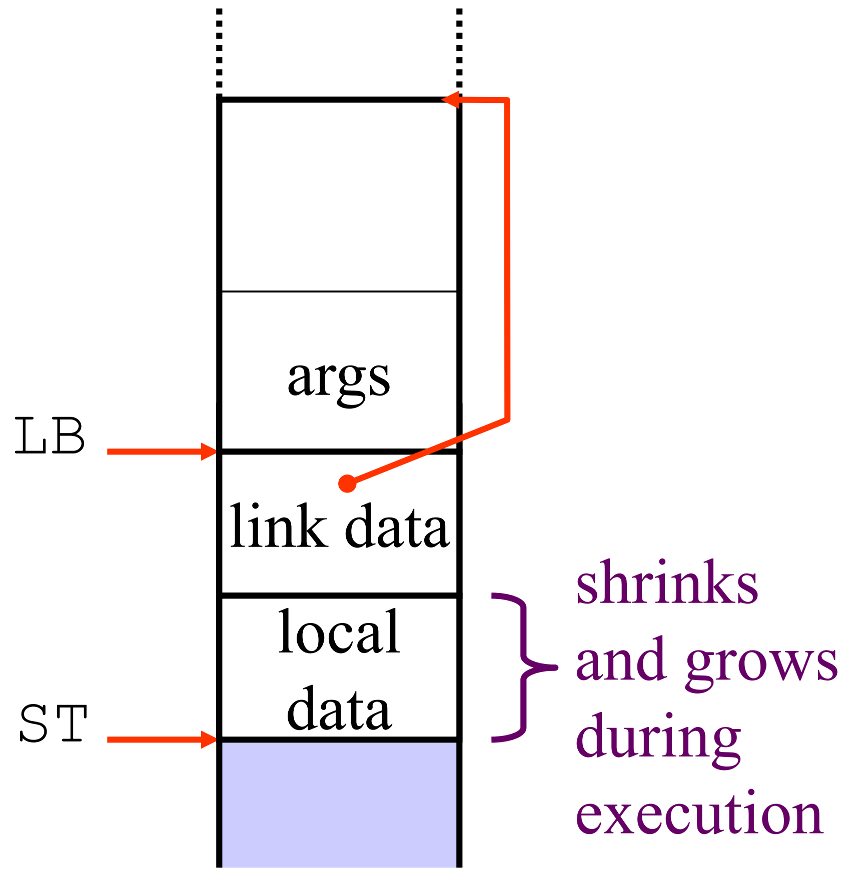
note: Going from (1) -> (2) in JVM is the execution of a single `CALL` instruction.

Routine Protocol

(2) just after entry

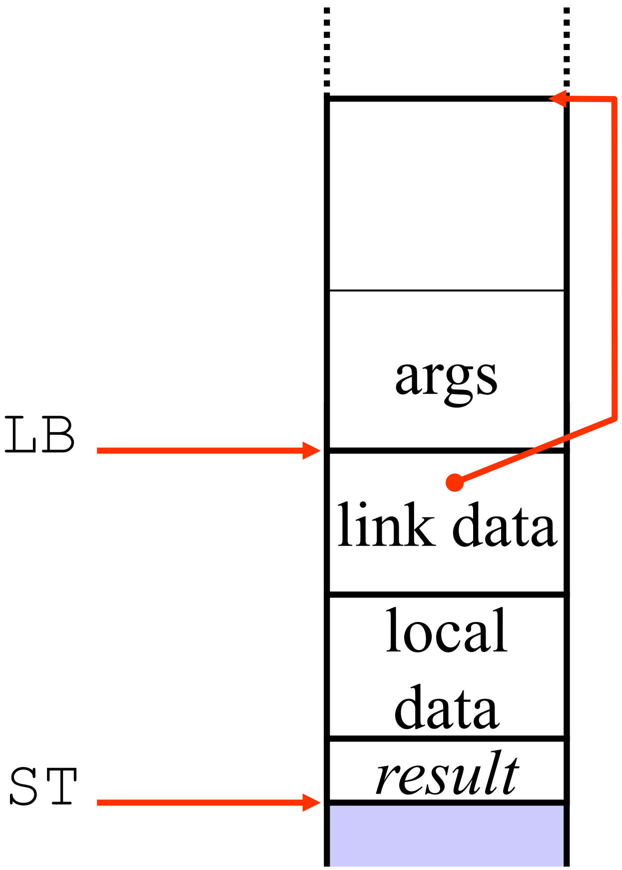


(3.1) during execution of routine

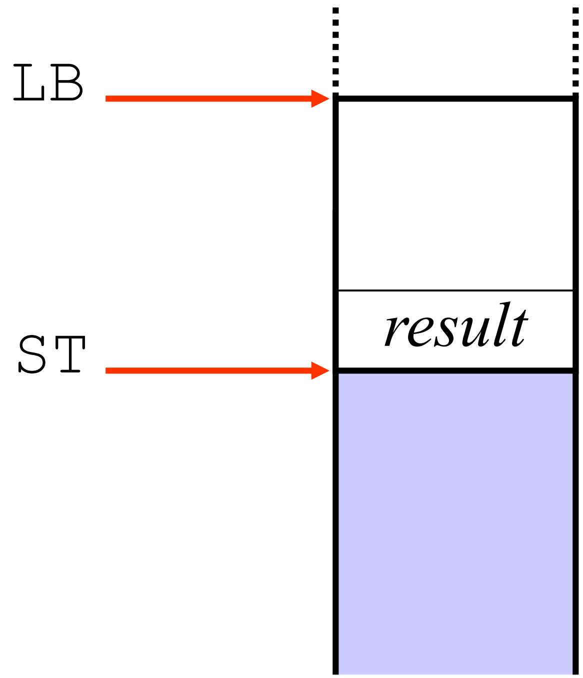


Routine Protocol

(3.2) just before return



(4) just after return



note: Going from (3.2) -> (4) in JVM is the execution of a single `RETURN` instruction.

Procedures and Functions: Parameters

We extend Mini Triangle with ...

```
Declaration
   ::= ...
     | proc Identifier (Formal) : TypeDenoter ~
               Expression
Expression
   ::= ...
     | Identifier (Actual)
Formal
   ::= Identifier : TypeDenoter
     | var Identifier : TypeDenoter
Actual
   ::= Expression
     | var VName
```

Code Templates Parameters

elaborate [**proc** I (FP) ~ C] =
    JUMP *g*
  e: *execute* [C]
    RETURN(0) *d*   where *d* is the size of FP
  g:

execute [I (AP)] =
    passArgument [AP]
    CALL(*r*) *e*   Where (l,e) = address of routine bound to I,
          Cl = current routine level
passArgument [E] =   r = display-register(cl,l)
    evaluate [E]
passArgument [**var** V] =
    fetchAddress [V]

Arguments: by value or by reference

Value parameters:
At the call site the argument is an expression, the evaluation of that expression leaves some value on the stack. The value is passed to the procedure/function.
A typical instruction for putting a value parameter on the stack:
`LOADL 6`

Var parameters:
Instead of passing a value on the stack, the address of a memory location is pushed. This implies a restriction that only “variable-like” things can be passed to a var parameter. In Triangle there is an explicit keyword **var** at the call-site, to signal passing a var parameter. In Pascal and C++ the reference is created implicitly (but the same restrictions apply).
Typical instructions: `LOADA 5[LB]` `LOADA 10[SB]`

Summary

- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- Production compilers do different things
  - Emphasis is on keeping values (esp. current stack frame) in registers
  - Intermediate results are laid out in the AR, not pushed and popped from the stack

Pause

Code generation for the MIPS Architecture

MIPS is implementation of a RISC architecture

- MIPS32 ISA
  - Designed for use with high-level programming languages
    - small set of instructions and addressing modes, easy for compilers
      - fixed instruction width (32-bits),
      - minimize control complexity, allow for more registers
      - 32 general purpose registers (32 bits each)
  - Arithmetic operations use registers for operands and results
    - Must use load and store instructions to use operands and results in memory
  - Load-store machine
    - large register set (32 word sized regs)
    - minimize main memory access
- MIPS has a nice simulator called SPIM
- MIPS (sometimes called RISC-I) is inspiration for the RISC-V processor

MIPS organization

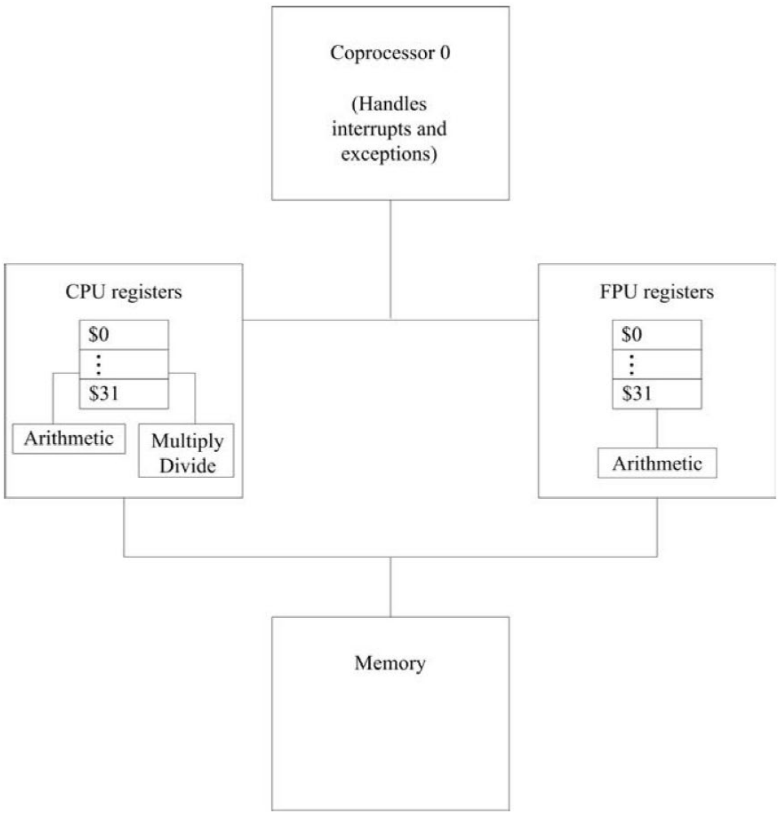


FIGURE 6.2 The MIPS computer organization.

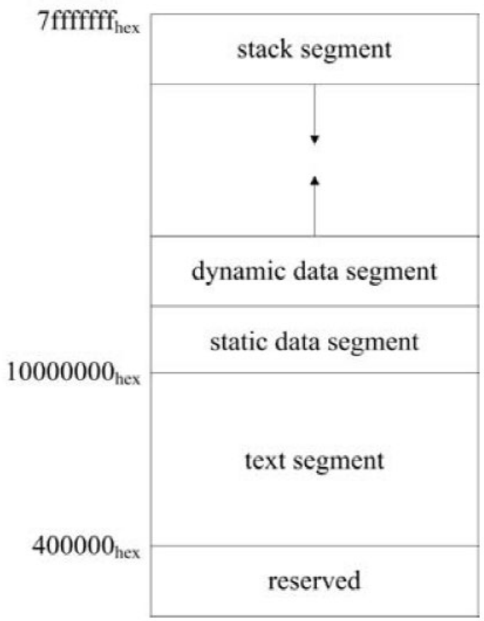


FIGURE 6.3 SPIM memory organization.

Source: Introduction to Compiler Construction in a Java World: B. Campbell et. Al.

Register conventions

register conventions and mnemonics

Number	Name	Use
0	`$zero`	hardwired 0 value
1	`$at`	used by assembler (pseudo-instructions)
2-3	`$v0-1`	subroutine return value
4-7	`$a0-3`	arguments: subroutine parameter value
8-15	`$t0-7`	temp: can be used by subroutine without saving
16-23	`$s0-7`	saved: must be saved and restored by subroutine
24-25	`$t8-9`	temp
26-27	`$k0-1`	kernel: interrupt/trap handler
28	`$gp`	global pointer (static or extern variables)
29	`$sp`	stack pointer
30	`$fp`	frame pointer
31	`$ra`	return address for subroutine
	`Hi, Lo`	used in multiplication (provide 64 bits for result)

hidden registers

PC, the program counter, which stores the current address of the instruction being executed

IR, which stores the instruction being executed

MIPS Instructions

- MIPS instructions fall into 5 classes:
  - Arithmetic/logical/shift/comparison (R-type)
  - Load/store (I-type)
  - Control instructions (branch and jump) (J-type)
  - Other (exception, register movement to/from GP registers, etc.)

- Three instruction encoding formats:
  - R-type (6-bit opcode, 5-bit rs, 5-bit rt, 5-bit rd, 5-bit shamt, 6-bit function code)

31-26	25-21	20-16	15-11	10-6	5-0
opcode	*rs*	*rt*	*rd*	shamt	function

  - I-type (6-bit opcode, 5-bit rs, 5-bit rt, 16-bit immediate)

31-26	25-21	20-16	15-0
opcode	*rs*	*rt*	*imm*

  - J-type (6-bit opcode, 26-bit pseudo-direct address)

31-26	25-0
opcode	*pseudodirect jump address*

A Sample of MIPS Instructions

- lw reg_1 offset(reg_2)
  - Load 32-bit word from address reg_2 + offset into reg_1
- add reg_1 reg_2 reg_3
  - $reg_1 \leftarrow reg_2 + reg_3$
- sw reg_1 offset(reg_2)
  - Store 32-bit word in reg_1 at address reg_2 + offset
- addiu reg_1 reg_2 imm
  - $reg_1 \leftarrow reg_2 + imm$
  - "u" means overflow is not checked
- li reg imm
  - $reg \leftarrow imm$

MIPS Addressing Modes

- MIPS addresses register operands using 5-bit field
  - Example: **`ADD $2, $3, $4`**
- Immediate addressing
  - Operand is help as constant (literal) in instruction word
  - Example: **`ADDI $2, $3, 64`**

- MIPS addresses load/store locations
  - base register + 16-bit signed offset (byte addressed)
    - Example: **`LW $2, 128($3)`**

  - 16-bit direct address (base register is 0)
    - Example: **`LW $2, 4092($0)`**

  - indirect (offset is 0)
    - Example: **`LW $2, 0($4)`**

MIPS Addressing Modes

- MIPS addresses jump targets as register content or 26-bit "pseudo-direct" address
- Example: JR $31, J 128

- MIPS addresses branch targets as signed instruction offset
  - relative to next instruction ("PC relative")
  - in units of instructions (words)
  - held in 16-bit offset in I-type
  - Example: **`BEQ $2, $3, 12`**

A small language example

- A language with integers and integer operations

$$P \rightarrow D; P \mid D$$

$$D \rightarrow \text{def id(ARGS)} = E;$$

$$\text{ARGS} \rightarrow \text{id, ARGS} \mid \text{id}$$

$$E \rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4$$

$$\mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)$$

- The first function definition f is the "main" routine
- Running the program on input i means computing f(i)

Code Generation Strategy

- For each expression e we generate MIPS code that:
  - Computes the value of e in $a0
  - Preserves $sp and the contents of the stack

- We define a code generation function cgen[e] whose result is the code generated for e

Code Generation for Sub and Constants

- The code to evaluate a constant simply copies it into the accumulator:

- cgen[i] = li $a0 i

- Note that this also preserves the stack, as required

Code Generation for Add and SUB

cgen[e1 + e2] =

```
cgen[e1]
sw $a0 0($sp)
addiu $sp $sp -4
cgen[e2]
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

Cgen[e_1 - e_2] =

```
cgen[e1]
sw $a0 0($sp)
addiu $sp $sp -4
cgen[e2]
lw $t1 4($sp)
sub $a0 $t1 $a0
addiu $sp $sp 4
```

Code Generation for Conditional

- We need flow control instructions
- Instruction: beq reg_1 reg_2 label
  - Branch to label if $reg_1 = reg_2$
- Instruction: b label
  - Unconditional jump to label

Code Generation for Conditional

```
Cgen[if e1 = e2 then e3 else e4] =
  cgen[e1]
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen[e2]
  lw $t1 4($sp)
  addiu $sp $sp 4
  beq $a0 $t1 true_branch
  false_branch:
    cgen[e4]
    b end_if
  true_branch:
    cgen[e3]
  end_if:
```

The Activation Record

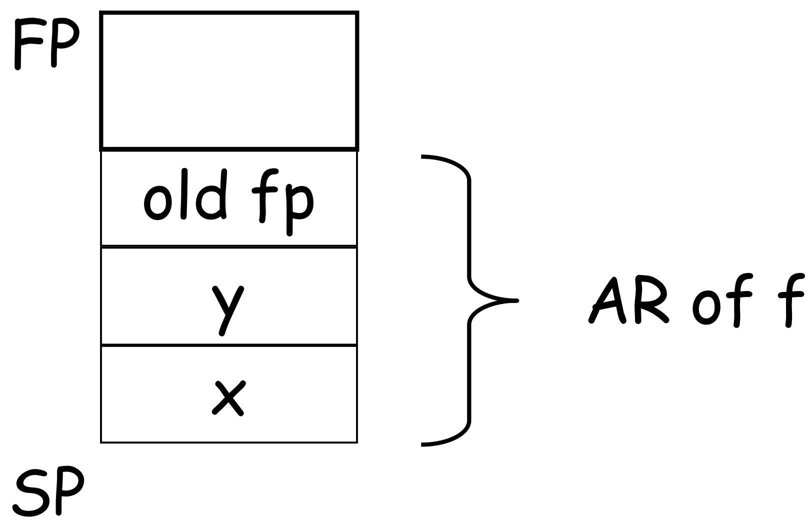
- Code for function calls and function definitions depends on the layout of the activation record
- A very simple AR suffices for this language:
  - The result is always in the accumulator
    - No need to store the result in the AR
  - The activation record holds actual parameters
    - For $f(x_1,\ldots,x_n)$ push $x_n,\ldots,x_1$ on the stack
    - These are the only variables in this language

The Activation Record (Cont.)

- The stack discipline guarantees that on function exit $sp is the same as it was on function entry
  - No need for a control link/static link
- We need the return address
- It’s handy to have a pointer to the current activation
  - This pointer lives in register $fp (frame pointer)
  - Reason for frame pointer will be clear shortly

The Activation Record

- For this language, an AR with the caller's frame pointer (dynamic link), the actual parameters, and the return address suffices
- Picture: Consider a call to f(x,y), The AR will be:



Code Generation for Function Call

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation
- New instruction: jal label
  - Jump to label, save address of next instruction in $ra
  - On other architectures the return address is stored on the stack by the "call" instruction

Code Generation for Function Call (Cont.)

```
Cgen[f(e1,…,en)] =
    sw $fp 0($sp)
    addiu $sp $sp -4
    cgen[en]
    sw $a0 0($sp)
    addiu $sp $sp -4
    …
    cgen[e1]
    sw $a0 0($sp)
    addiu $sp $sp -4
    jal f_entry
```

- The caller saves its value of the frame pointer
- Then it saves the actual parameters in reverse order
- The caller saves the return address in register $ra
- The AR so far is 4*n+4 bytes long

Code Generation for Function Definition

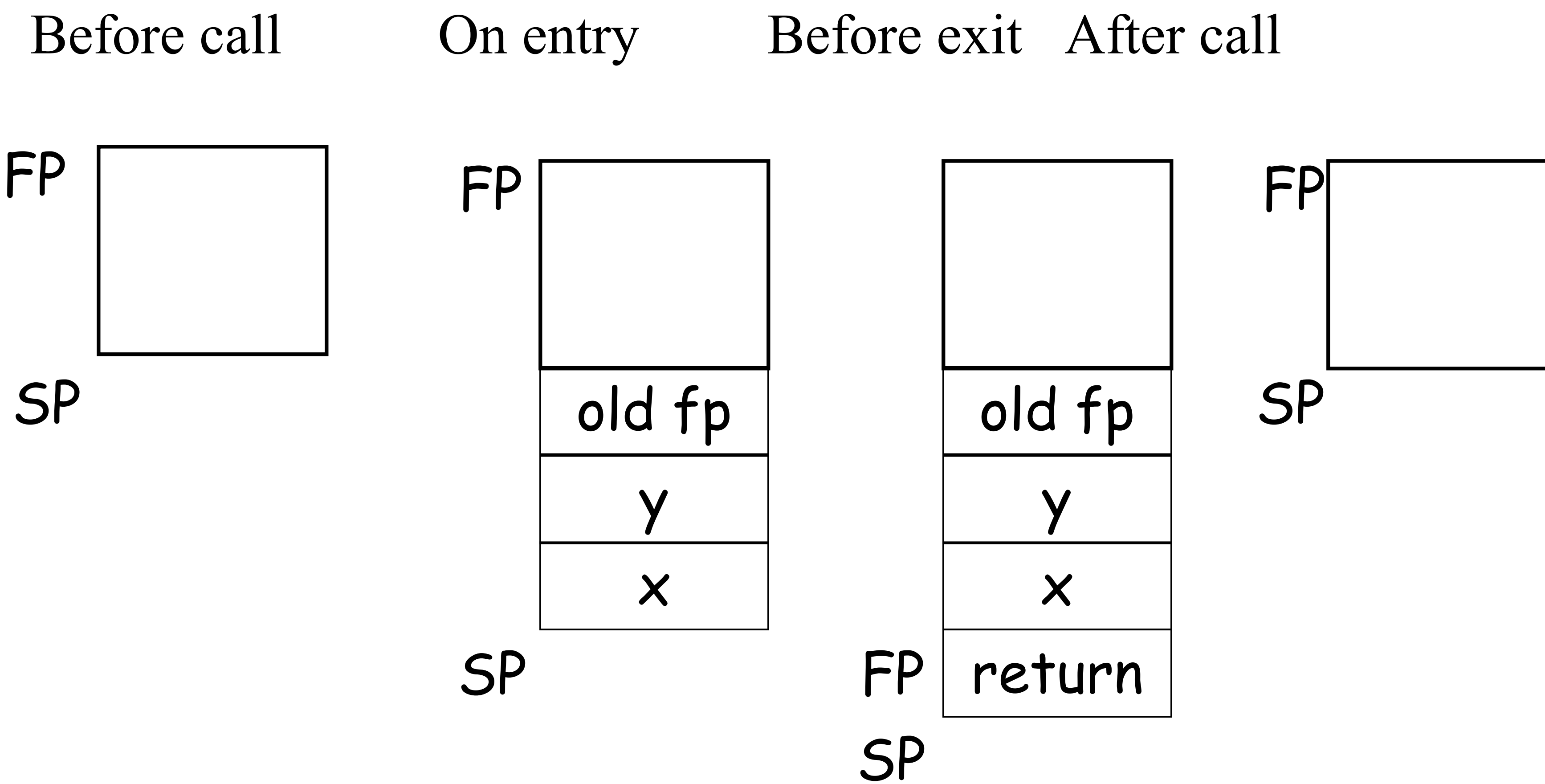
- Instruction: jr reg
  - Jump to address in register reg

Cgen[def f(x_1,…,x_n) = e] =

```
move $fp $sp
sw $ra 0($sp)
addiu $sp $sp -4
cgen[e]
lw $ra 4($sp)
addiu $sp $sp z
lw $fp 0($sp)
jr $ra
```

- Note: The frame pointer points to the top, not bottom of the frame
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- $z = 4*n + 8$

Calling Sequence. Example for f(x,y).



Code Generation for Variables

- Variable references are the last construct
- The “variables” of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller

- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $sp

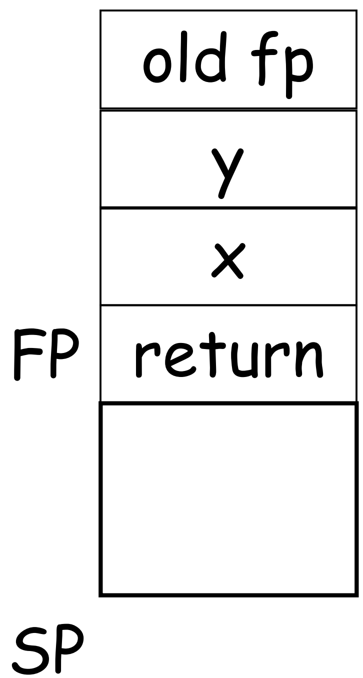
Code Generation for Variables (Cont.)

- Solution: use a frame pointer
  - Always points to the return address on the stack
  - Since it does not move it can be used to find the variables
- Let x_i be the i^{th} ($i = 1,\ldots,n$) formal parameter of the function for which code is being generated

cgen[x_i] = lw \$a0 z(\$fp)   (z = 4*i)

Code Generation for Variables (Cont.)

- Example: For a function def f(x,y) = e the activation and frame pointer are set up as follows:



- X is at fp + 4
- Y is at fp + 8

fac(n) = if (n = 1) then 1 else (n*fac(n-1))

```
move $fp $sp                        #copy fp to top of stack
sw $ra 0($sp)                       #save ra on top of stack
addiu $sp $sp -4                              #adjust tos
    lw $a0 4($fp)                   #/load n
      sw $a0 0($sp)                 # save n on tos
      addiu $sp $sp -4
      li $a0 1                      #load 1
      lw $t1 4($sp)                 #load n into t1
      addiu $sp $sp 4
      beq $a0 $t1 true_branch       #branch if 1 = n
     false_branch:
           lw $a0 4($fp)            #load n
           sw $a0 0($sp)
           addiu $sp $sp -4
           lw $a0 4($fp)            #load n
             sw $a0 0($sp)
             addiu $sp $sp -4
             li $a0 1               #load 1
             lw $t1 4($sp)
             sub $a0 $t1 $a0                  #n-1
             addiu $sp $sp 4
             sw $a0 0($sp)
             addiu $sp $sp -4
             jal f_entry            #call fac
        lw $t1 4($sp)
        mul $a0 $t1 $a0             #n*fac(n-1)
        addiu $sp $sp 4
       b end_if
      true_branch:
       li $a0 1                     #load 1
      end_if:
lw $ra 4($sp)
addiu $sp $sp 4                     #remove n from toc
lw $fp 0($sp)
jr $ra                              #return from fac
```

Pause

Instruction selection by patternmatching

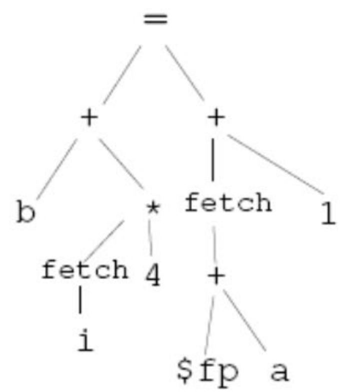


Figure 13.26: Low-Level IR Representation of b[i]=a+1

Translate AST to tree rep.
with leaves corresponding
to registers, memeory locations or litterals
and internal nodes to fetch
and basic operations

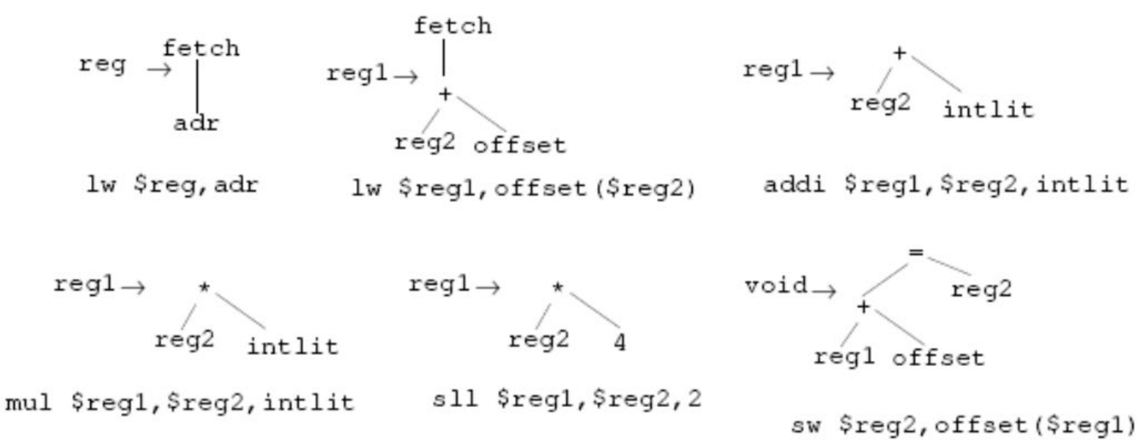


Figure 13.27: IR Tree Patterns for Various MIPS Instructions

Instruction selection is now a question of pattern matching similar to bottom up parsing

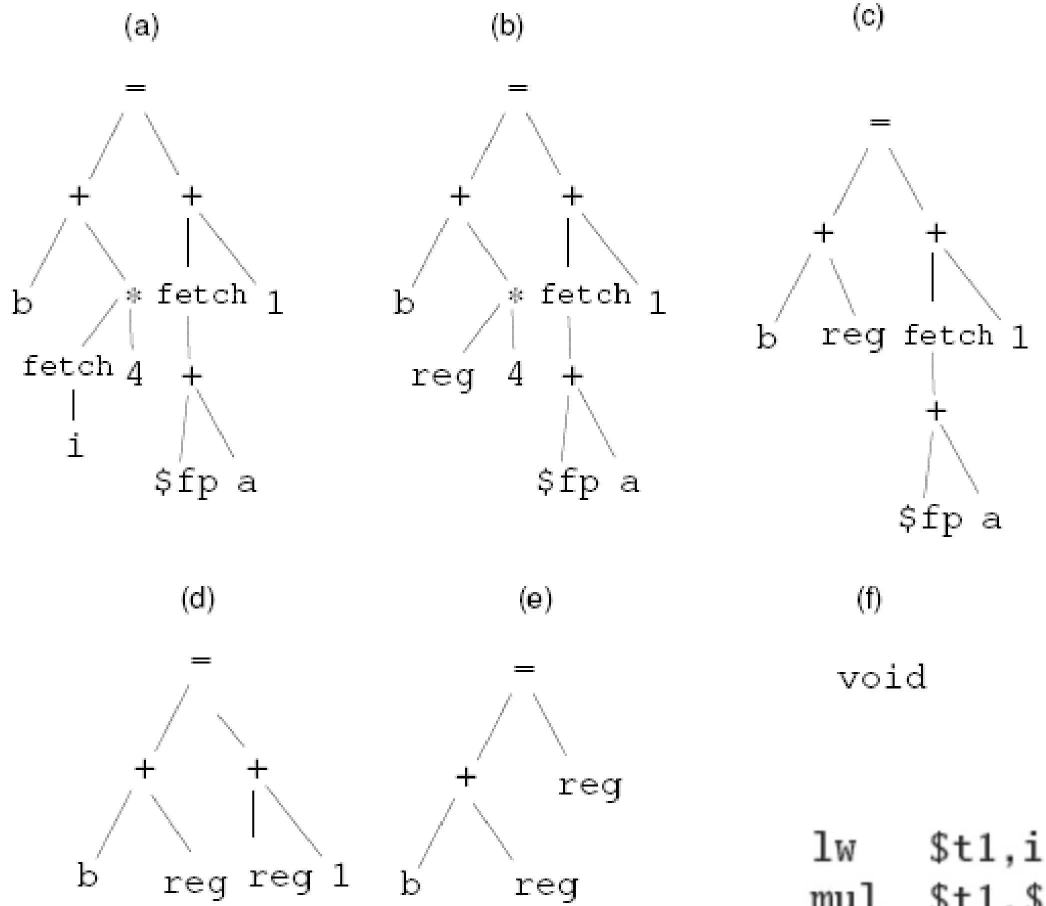
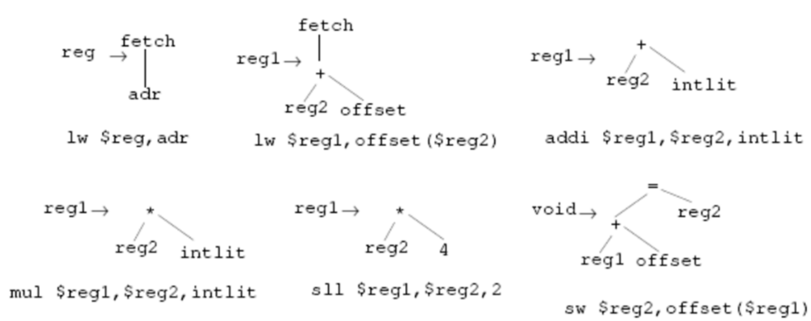


Figure 13.28: Instruction Selection Using Patterns




```
lw   $t1,i
mul  $t1,$t1,4
lw   $t2,a($fp)
addi $t2,$t2,1
sw   $t2,b($t1)
```

Figure 13.29: MIPS code for b[i]=a+1

Code generation from IR

- JBC to Machine code is used by AOT (Ahead-of-Time) Java compilers like gcj and FijiVM
- JBC to Machine code is used by all JIT VMs
  - Some JIT VM compile JBC on class loading
  - Others start interpretation and then compile HOT methods and store the compiled code in a method cashe
  - Others record sequences of JBC and discover ”often used sequences” and then compiles these – so called trace based JIT (e.g. Mozilla’s TraceMonkey)
- We look at JBC to MIPS

```
iload 2       ; Push int b onto stack
iload 3       ; Push int c onto stack
iadd          ; Add top two stack values
iload 4       ; Push int d onto stack
isub          ; Subtract top two stack values
istore 1      ; Store top stack value into a
```

Figure 13.1: Bytecodes for `a = b + c - d;`

```
lw     $t0,16($fp)      # Load b, at 16+$fp, into $t0
lw     $t1,20($fp)      # Load c, at 20+$fp, into $t1
add    $t2,$t0,$t1      # Add $t0 and $t1 into $t2
lw     $t3,24($fp)      # Load d, at 24+$fp, into $t3
sub    $t4,$t2,$t3      # Subtract $t3 from $t2 into $t4
sw     $t4,12($fp)      # Store result into a, at 12+$fp
```

Figure 13.2: MIPS code for `a = b + c - d;`

```
bltz    $index,badIndex         # Branch to badIndex if $index<0
lw      $temp,SIZE($array)      # Load size of array into $temp
slt     $temp,$index,$temp      # $temp = $index < size of array
beqz    $temp,badIndex          # Branch to badIndex if
                                # $index >= size of array
sll     $temp,$index,2          # multiply $index by 4 (size of
                                # an int) using a left shift
add     $temp,$temp,$array      # Compute $array + 4*$index
lw      $val,OFFSET($temp)      # Load word at
                                # $array + 4*$index + OFFSET
```

Figure 13.3: MIPS code for `iaload` bytecode

```
bltz    $index,badIndex         # Branch to badIndex if $index<0
lw      $temp,SIZE($array)      # Load size of array into $temp
slt     $temp,$index,$temp      # $temp = $index < size of array
beqz    $temp,badIndex          # Branch to badIndex if
                                # $index >= size of array
sll     $temp,$index,2          # multiply $index by 4 (size of
                                # an int) using a left shift
add     $temp,$temp,$array      # Compute $array + 4*$index
sw      $val,OFFSET($temp)      # Load $val into word at
                                #  $array + 4*$index + OFFSET
```

Figure 13.4: MIPS code for `iastore` bytecode

```
move    $a0,$t0          # Copy $t0 to parm register 1
li      $a1,2            # Load 2 into parm register 2
sw      $t0,32($fp)      # Store $t0 across call
jal     f                # Call function f
                         # Function value is in $v0
lw      $t0,32($fp)      # Restore $t0
sw      $v0,a            # Store function value in a
```

Figure 13.5: MIPS code for the function call `a = f(i,2);`

```
subi   $sp,$sp,frameSz     # Push frame on stack
sw     $ra,0($sp)          # Save return address in frame
sw     $fp,4($sp)          # Save old frame pointer in frame
move   $fp,$sp             # Set $fp to access new frame
# Save callee-save registers (if any) here
# Body of method is here
# Restore callee-save registers (if any) here
lw     $ra,0($fp)          # Reload return address register
lw     $fp,4($fp)          # Reload old frame pointer
addi   $sp,$sp,frameSz     # Pop frame from stack
jr     $ra                 # Jump to return address
```

Figure 13.6: MIPS prologue and epilogue code

Stringsum example

```
public static String stringSum(int limit){
        int sum = 0;
        for (int i = 1; i <= limit; i++)
                sum += i;
        return Integer.toSting(sum);
}
```

```
    iconst_0      ; Push 0
    istore_1      ; Store into variable #1 (sum)
    iconst_1      ; Push 1
    istore_2      ; Store into variable #2 (i)
    goto L2       ; Go to end of loop test
L1: iload_1       ; Push var #1 (sum) onto stack
    iload_2       ; Push var #2 (i) onto stack
    iadd          ; Add sum + i
    istore_1      ; Store sum + i into var #1 (sum)
    iinc 2 1      ; Increment var #2 (i) by 1
L2: iload_2       ; Push var #2 (i)
    iload_0       ; Push var #0 (limit)
    if_icmple L1 ; Goto L1 if i <= limit
    iload_1       ; Push var #1 (sum) onto stack
                  ; Call toString:
    invokestatic
          java/lang/Integer/toString(I)Ljava/lang/String;
    areturn       ; Return String reference to caller
```

Figure 13.7: Bytecodes for method `stringSum`

```
        subi    $sp,$sp,20      # Push frame on stack
        sw      $ra,0($sp)      # Save return address
        sw      $fp,4($sp)      # Save old frame pointer
        move    $fp,$sp         # Set $fp to access new frame
        sw      $a0,8($fp)      # Store limit in frame
        sw      $0,12($fp)      # Store 0 ($0) into sum
        li      $t0,1           # Load 1 into $t0
        sw      $t0,16($fp)     # Store 1 into i
        j       L2              # Go to end of loop test
L1:     lw      $t1,12($fp)     # Load sum into $t1
        lw      $t2,16($fp)     # Load i into $t2
        add     $t3,$t1,$t2     # Add sum + i into $t3
        sw      $t3,12($fp)     # Store sum + i into sum
        lw      $t4,16($fp)     # Load i into $t2
        addi    $t4,$t4,1       # Increment $t4 by 1
        sw      $t4,16($fp)     # Store $t4 into i
L2:     lw      $t5,16($fp)     # Load i into $t5
        lw      $t6,8($fp)      # Load limit into $t6
        sle     $t7,$t5,$t6     # set $t7 = i <= limit
        bnez    $t7,L1          # Goto L1 if i <= limit
        lw      $t8,12($fp)     # Load sum into $t8
        move    $a0,$t8         # Copy $t8 to parm register
        jal     String_toString_int_   # Call toString
                                # String ref now is in $v0
        lw      $ra,0($fp)      # Reload return address
        lw      $fp,4($fp)      # Reload old frame pointer
        addi    $sp,$sp,20      # Pop frame from stack
        jr      $ra             # Jump to return address
```

Figure 13.8: MIPS code for method `stringSum`

Register Allocation

- A compiler generating code for a register machine needs to pay attentention to register allocation as this is a limited ressource
- In routine protocol
  - Allocate arg1 in R1, arg2 in R2 .. Result in R0
  - But what if there are more args than regs?
- In evaluation of expressions
  - On MIPS all calculations take place in regs
  - Reduce traffic between memory and regs

```
procedure registerNeeds(T)
    if T.kind = Identifier or T.kind = IntegerLiteral
    then  T.regCount ← 1
    else
        call registerNeeds(T.leftChild)
        call registerNeeds(T.rightChild)
        if T.leftChild.regCount = T.rightChild.regCount
        then  T.regCount ← T.rightChild.regCount + 1
        else
            T.regCount ← max(T.leftChild.regCount, T.rightChild.regCount)
end
```

Figure 13.9: An Algorithm to Label Expression Trees with Register Needs

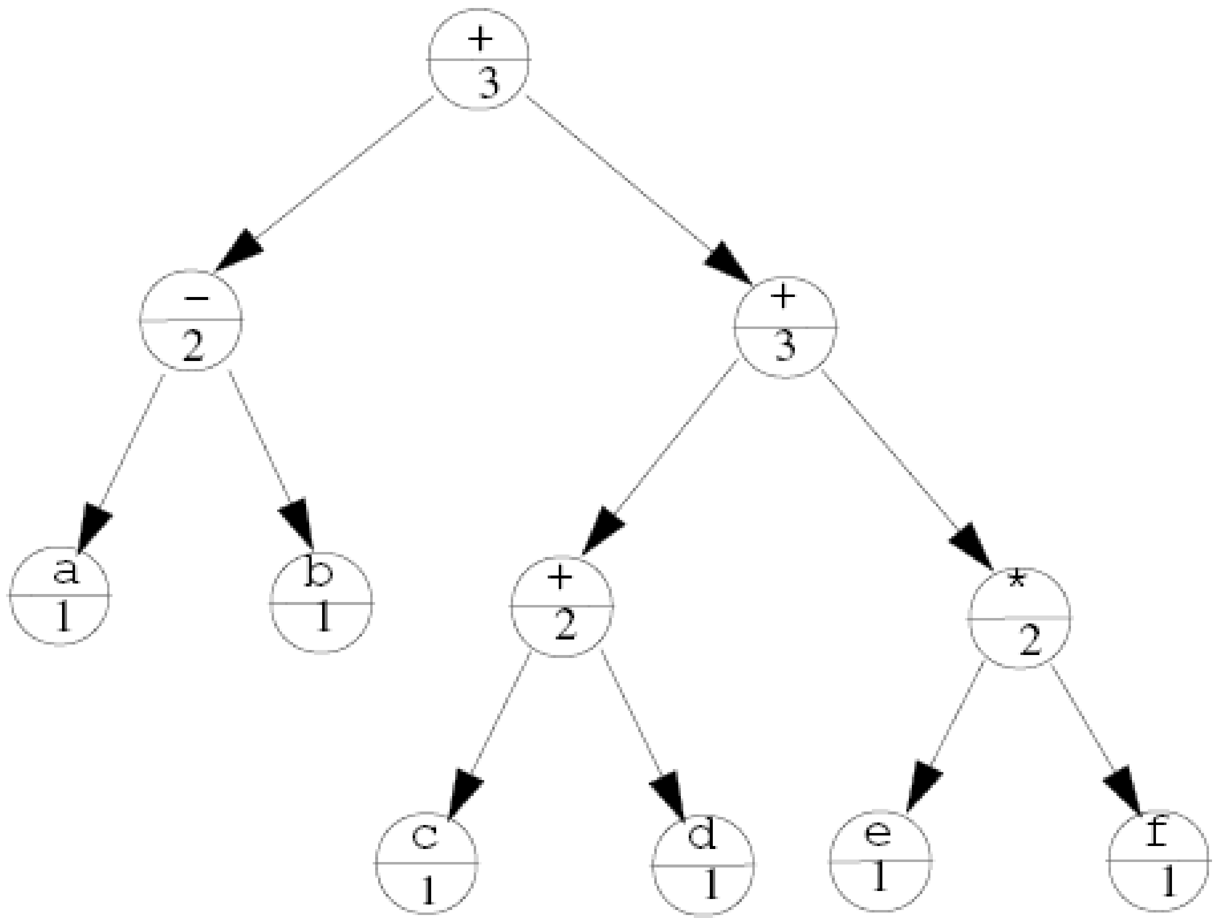


Figure 13.10: Expression Tree for (a-b) + ((c+d)+(e*f)) with Register Needs.

```
lw   $10, c          # Load c into register 10
lw   $11, d          # Load d into register 11
add  $10, $10, $11   # Compute c + d into register 10
lw   $11, e          # Load e into register 11
lw   $12, f          # Load f into register 12
mul  $11, $11, $12   # Compute e * f into register 11
add  $10, $10, $11   # Compute (c + d) + (e * f) into reg 10
lw   $11, a          # Load a into register 11
lw   $12, b          # Load b into register 12
sub  $11, $11, $12   # Compute a - b into register 11
add  $10, $11, $10   # Compute (a-b)+((c+d)+(e*f)) into reg 10
```

Figure 13.11: MIPS code for `(a-b) + ((c+d)+(e*f))`

```
procedure treeCG(T, regList)
    r1 ← head(regList)
    r2 ← head(tail(regList))
    if T.kind = Identifier
    then
        /*  Load a variable.                                    */
        call generate(lw, r1, T.IdentifierName)
    else
        if T.kind = IntegerLiteral
        then
            /*  Load a literal.                                 */
            call generate(li, r1, T.IntegerValue)
        else
            /*  T.kind must be a binary operator.               */
            left ← T.leftChild
            right ← T.rightChild
            if left.regCount ≥ length(regList) and right.regCount ≥ length(regList)
            then
                /*  Must spill a register into memory.          */
                call treeCG(left, regList)
                /*  Get memory location.                        */
                temp ← getTemp( )
                call generate(sw, r1, temp)
                call treeCG(right, regList)
                call generate(lw, r2, temp)
                /*  Free memory location.                       */
                call freeTemp(temp)
                call generate(T.operation, r1, r2, r1)
            else
                /*  There are enough registers; no spilling is needed.  */
                if left.regCount ≥ right.regCount
                then
                    call treeCG(left, regList)
                    call treeCG(right, tail(regList))
                    call generate(T.operation, r1, r1, r2)
                else
                    call treeCG(right, regList)
                    call treeCG(left, tail(regList))
                    call generate(T.operation, r1, r2, r1)
end
```

Figure 13.12: An Algorithm to Generate Optimal Code from Expression Trees

Optimizing register allocations

- TreeCG generates code such that result(s) end up in targeted registers
- However TreeCG does not exploit communicative operators
  - exp1 op exp2 = exp2 op exp1
  - Also difficult due to overflow or exceptions
- Exploiting associativity can reduce reg needs
  - (a+b)+(c+d) needs 3 regs
  - a+b+c+d needs only 2 regs

Register Allocation

- Expression level register allocation
- Procedure level register allocation
  - Interference graphs
  - Graph coloring
- Intra-procedural register allocation
  - 10%-28% speed-up

Code scheduling

- Modern computers are pipelined
  - Instructions are processed in stages
  - Instructions take different time to execute
  - If result from previous instruction is needed but not yet ready then we have a **stalled pipeline**
  - Delayed load
    - Load from memory takes 2, 10 or 100 cycles
  - Also FP instructions takes time

```
1. lw  $10,a            6.  add  $10,$10,$12
2. lw  $11,b            7.  mul  $11,$11,$10
3. mul $11,$10,$11      8.  mul  $12,$10,$12
4. lw  $10,c            9.  add  $12,$11,$12
5. lw  $12,d            10. sw   $12,a
```

Figure 13.21: MIPS code for a=((a*b)*(c+d))+(d*(c+d))

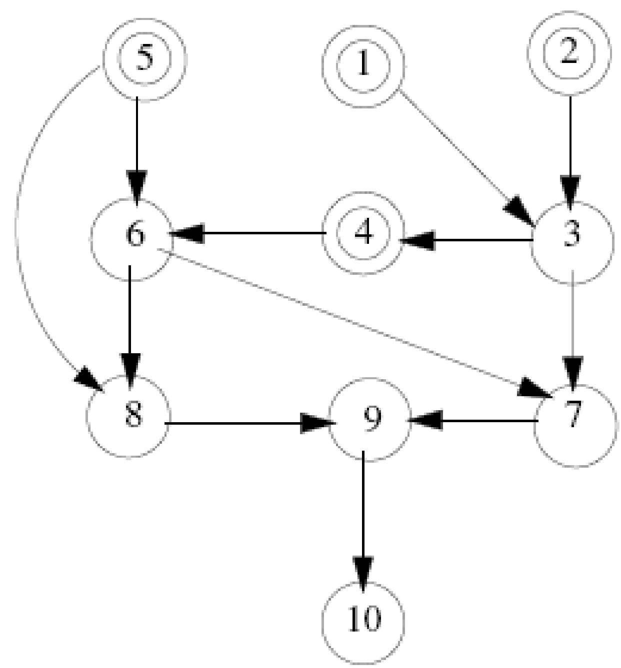


Figure 13.22: Dependency DAG for a=((a*b)*(c+d))+(d*(c+d))

procedure SCHEDULEDAG(*dependencyDAG*)
    candidates ← ROOTS(*dependencyDAG*)
    while *candidates* ≠ ∅ **do**
        call SELECT(*candidates*, "Is not stalled by last instruction generated")
        call SELECT(*candidates*, "Can stall some successor")
        call SELECT(*candidates*, "Exposes the most new roots if generated")
        call SELECT(*candidates*, "Has the longest path to a leaf")
        inst ← Any *node* ∈ *candidates*
        Schedule *inst* as next instruction to be executed
        dependencyDAG ← *dependencyDAG* − {*inst*}
        candidates ← ROOTS(*dependencyDAG*)
end

Figure 13.23: An Algorithm to Schedule Code from a Dependency DAG

```
1. lw  $10,a           6. add $10,$10,$12
2. lw  $11,b           7. mul $11,$11,$10
3. lw  $12,d           8. mul $12,$10,$12
4. mul $11,$10,$11     9. add $12,$11,$12
5. lw  $10,c           10 sw  $12,a
```

Figure 13.24: Scheduled MIPS code for `a=((a*b)*(c+d))+(d*(c+d))`

Reg allocation and Code Scheluling

- Reg allocations algorithms try to minimize the number of regs used
- May conflict with pipeline architecture
  - Using more regs than strictly necessary may avoid pipeline stalls
- Solution
  - Integrated register allocator and code scheduler

```
1. lw  $10,a          6.  add  $10,$13,$12
2. lw  $11,b          7.  mul  $11,$11,$10
3. lw  $12,d          8.  mul  $12,$10,$12
4. lw  $13,c          9.  add  $12,$11,$12
5. mul $11,$10,$11    10. sw   $12,a
```

Figure 13.25: Delay-free MIPS code for a=((a*b)*(c+d))+(d*(c+d))

Modern Hardware and code generation

- Speculative execution
- Prefetch instructions
  - Load data into cache
- Dynamic scheduling
- Out of order architectures

- Should the HW, Compiler or the programmer do the job?

Register variable in C

- Ex: `register float a = 0 ;`

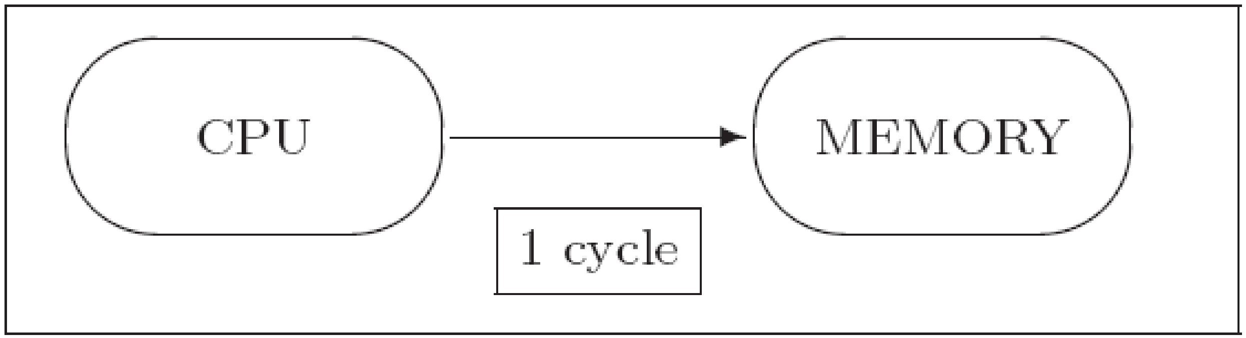
- register provides a hint to the compiler that you think a variable will be frequently used
- compiler is free to ignore register hint
- if ignored, the variable is equivalent to an auto variable with the exception that you may not take the address of a register (since, if put in a register, the variable will not have an address)
- rarely used, since any modern compiler will do a better job of optimization than most programmers

Java Memory Model

- Abstract memory model
  - Local stack for each thread
    - But stacks may need to be implemented via registers and memory
  - Shared variables can be problematic on some implementations
    - Serial to concurrent
      - Code for serial execution may not work in concurrent system
    - Concurrent to serial
      - Code with synchronization may be inefficient in serial programs (10-20% unnecessary overhead)

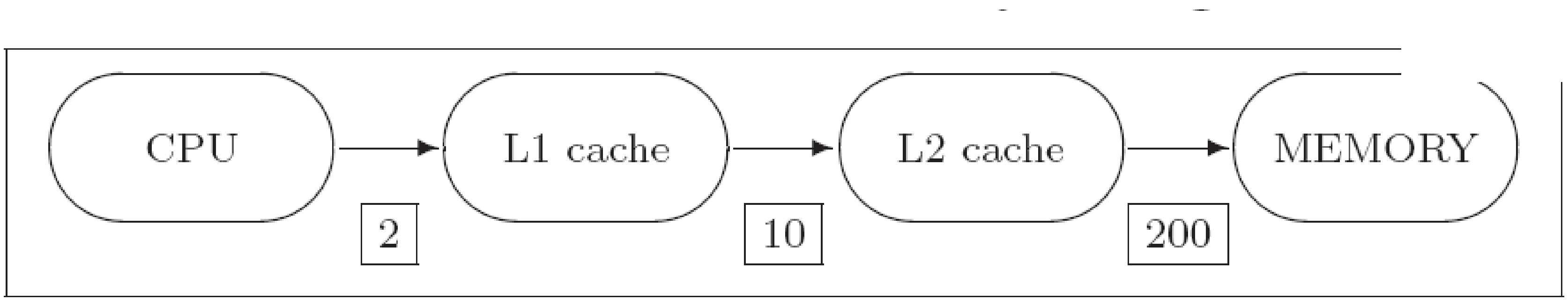
  - Java 1.5 has expanded the definition of the memory model
    - Volatile keyword
      - The value of a volatile variable will **never be cached thread-locally**: all reads and writes will go straight to "main memory"

A programmer's view of memory



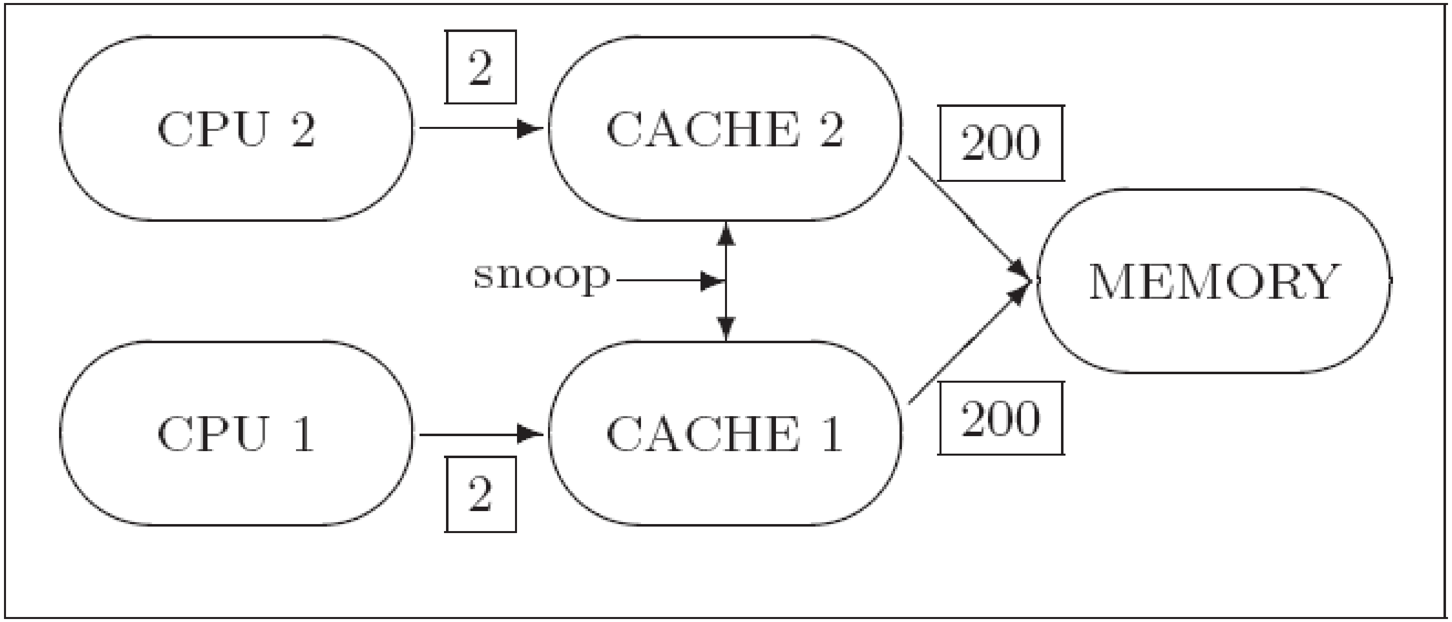
This model was pretty accurate in 1985.
Processors (386, ARM, MIPS, SPARC) all ran at 1–10MHz clock speed and could access external memory in 1 cycle; and most instructions took 1 cycle.
Indeed the C language was as expressively time-accurate as a language could be: almost all C operators took one or two cycles.
But this model is no longer accurate!

A modern view of memory timings



So what happened?
On-chip computation (clock-speed) sped up
faster (1985–2005) than off-chip communication (with memory) as feature sizes shrank.
The gap was filled by spending transistor budget on caches which (statistically) filled the mismatch until 2005 or so.
Techniques like caches, deep pipelining with bypasses, and superscalar instruction issue burned power to preserve our illusions.
2005 or so was crunch point as faster, hotter, single-CPU Pentiums were scrapped. These techniques had delayed the inevitable.

The Current Mainstream Processor



Will scale to 2, 4 maybe 8 processors.
But ultimately shared memory becomes the bottleneck (1024 processors?!?).

Conclusions

- Low level code genrations requires attentions to lots of details:
  - Instruction sequence selection
  - Register allocation
  - Instruction scheduling
  - Storage allocation
    - Memory hierachies
  - (multi-core placement)
- Sometimes Implications for language design
  - E.g. high level memory models

What can you do in your projects now?

- You should by now have lexer, parser and AST in place
  - Write pretty printer to test front end
    - Use all the programs you wrote when designing your syntax
- You should have static semantic analyzer in place.
  - Write recursive interpreter to test programs
  - And generate ideas for formal semantics
- Code generation:
  - Write C, Java, python … code generator
  - Write JBC or CIL code generator
  - Write MIPS, AVR or x86

Languages and Compilers
(SProg og Oversættere)

Lecture 19

Abstract Data Types
and
Object Oriented Features

Bent Thomsen

Department of Computer Science

Aalborg University

With acknowledgement to John Mitchell, Elsa Gunter, David Watt and Amiram Yehudai whose slides this lecture is based on.

Learning goals

- To understand the concept of abstract data types
- Understand implementations of abstract data types
- Understand concepts of Object Oriented programming:
  - Classes and objects
  - Inheritance
  - Dynamic dispatch
- Understand how classes and objects can be implemented
- Understand issues in modularity of large programs

Tennent's Language Design principles

- The Principle of Abstraction
  - All major syntactic categories should have abstractions defined over them. For example, functions are abstractions over expressions
- The Principle of Correspondence
  - Declarations ≈ Parameters
- The Principle of Data Type Completeness
  - All data types should be first class without arbitrary restriction on their use

–Originally defined by R.D.Tennent

The Concept of Abstraction

- The concept of abstraction is fundamental in programming (and computer science)
- Tennents principle of abstraction
  - is based on identifying all of the semantically-meaningful syntactic categories of the language and then designing a coherent set of abstraction facilities for each of these.
- Nearly all programming languages support process (or command) abstraction with subprograms (procedures)
- Many programming languages support expression abstraction with functions
- Nearly all programming languages designed since 1980 have supported data abstraction:
  - Abstract data types
  - Objects
  - Modules

What have we seen so far?

- Structured data
  - Arrays
  - Records or structs
  - Lists
- Visibility of variables and subprograms
  - Scope rules
- Why is this not enough?

Information Hiding

- Consider the C code:

```
 typedef struct RationalType {
    int numerator;
    int denominator;
} Rational

Rational mk_rat (int n,int d) { ...}
Rational add_rat (Rational x, Rational y) {
... }
```

- Can use **`mk_rat`**, **`add_rat`** without knowing the details of **`RationalType`**

Need for Abstract Types

- Problem: abstraction not enforced
  - User can create Rationals without using **`mk_rat`**
  - User can access and alter numerator and denominator directly without using provided functions

- With abstraction we also need information hiding

Abstract Types - Example

- Suppose we need sets of integers
- Decision:
  - implement as lists of int
- Problem:
  - lists have order and repetition, sets don't
- Solution:
  - use only lists of int ordered from smallest to largest with no repetition (data invariant)

Abstract Type – SML code Example

```
type intset = int list
val empty_set = []:intset
fun insert {elt, set = [] } = [elt]
      |  insert {elt, set = x :: xs} =
         if elt < x then elt :: x :: xs
         else if elt = x then x :: xs
         else x :: (insert {elt = elt, set = xs})

fun union ([],ys) = ys
    | union (x::xs,ys) =
       union(xs,insert{elt=x,set = ys})

fun intersect ([],ys) = []
   | intersect (xs,[]) = []
   | intersect (x::xs,y::ys) =
        if x <y then intersect(xs, y::ys)
        else if y < x then intersect(x::xs,ys)
        else x :: (intersect(xs,ys))
```

```
fun elt_of {elt, set = []} = false
   | elt_of {elt, set = x::xs} =
      (elt = x) orelse
      (elt > x andalso
       elt_of{elt = elt, set = xs})
```

Abstract Type – Example

- Notice that all these definitions maintain the data invariant for the representation of sets, and depend on it

- Are we happy now?
- NO!
- As is, user can create any pair of lists of int and apply union to them; the result is meaningless

Solution: abstract datatypes

```
abstype intset = Set of int list with
val empty_set = Set []
local
fun ins {elt, set = [] } = [elt]
   |  ins {elt, set = x :: xs} =
        if elt < x then elt :: x :: xs
        else if elt = x then x :: xs
        else x :: (ins {elt = elt, set =
         xs})
fun un ([],ys) = ys
   | un (x::xs,ys) =
      un (xs,ins{elt=x,set = ys})
in
     fun insert {elt, set = Set s}=
          Set(ins{elt = elt, set  = s})
     fun union (Set xs, Set ys) =
         Set(un (xs, ys))
end
```

```
local
fun inter ([],ys) = []
  | inter (xs,[]) = []
  | inter (x::xs,y::ys) =
      if x <y then inter(xs, y::ys)
      else if y < x then inter(x::xs,ys)
      else x :: (inter(xs,ys))

in
     fun intersect(Set xs, Set ys) =
       Set(inter(xs,ys))
end
fun elt_of {elt, set = Set []} = false
  | elt_of {elt, set = Set (x::xs)} =
     (elt = x) orelse
     (elt > x andalso
      elt_of{elt = elt, set = Set xs})
fun set_to_list (Set xs) = xs
end (* abstype *)
```

Abstract Type – Example

- Creates a new type (not equal to **int list**)
  - Remember type equivalence – structure vs. name
- Exports
  - type **intset,**
  - Constant **empty_set**
  - Operations: **insert, union, elt_of,** and **set_to_list**; act as primitive
  - Note: Unfortunately in SML we cannot use pattern matching or list functions on intset; won't type check
  - Lack of orthogonality in the design of abstype for SML – does not fulfill Tennent's principle of data type completion

Abstract Type – Example

- Implementation: just use **int list**, except for type checking

- Data constructor **Set** only visible inside the asbtype declaration; type intset visible outside

- Function **set_to_list** used only at compile time

- Data abstraction allows us to prove data invariant holds for all objects of type intset

Abstract Types

- A type is abstract if the user can only see:
  - the type
  - constants of that type (by name)
  - operations for interacting with objects of that type that have been explicitly exported
- Primitive types are built-in abstract types

  e.g. **int** type in Java
  - The representation is hidden
  - Operations are all built-in
  - User programs can define objects of **int** type
- User-defined abstract data types must have the same characteristics as built-in abstract data types

User Defined Abstract Types

- Syntactic construct to provide encapsulation of abstract type implementation
- Inside, implementation visible to constants and subprograms
- Outside, only type name, constants and operations visible, not implementation
- No runtime overhead as all the above can be checked statically

Advantages of Data Abstraction

- Advantage of Inside condition:
  - Program organization, modifiability (everything associated with a data structure is together)
  - Separate compilation may be possible
- Advantage of Outside condition:
  - Reliability--by hiding the data representations, user code cannot directly access objects of the type. User code cannot depend on the representation, allowing the representation to be changed without affecting user code.

Limitation of Abstract data types

Queue

```
abstype q
with
  mk_Queue : unit -> q
  is_empty : q -> bool
  insert   : q * elem -> q
  remove   : q -> elem
is  …
in
  program
end
```

Priority Queue

```
abstype pq
with
  mk_Queue : unit -> pq
  is_empty : pq -> bool
  insert   : pq * elem -> pq
  remove   : pq -> elem
is …
in
  program
end
```

But cannot intermix pq’s and q’s

Abstract Data Types

- Guarantee invariants of data structure
  - only functions of the data type have access to the internal representation of data
- Limited “reuse”
  - Cannot apply queue code to pqueue, except by explicit parameterization, even though signatures identical
  - Cannot form list of points and colored points

- Data abstraction is important – how can we make it extensible?
- Remember subtyping from Lecture 13 ?

Subtyping for Product Types

The rule is:

if $A <: T$ and $B <: U$ then $A \times B <: T \times U$

This rule, and corresponding rules for other structured types, can be worked out by following the principle:

$T <: U$ means that whenever a value of type U is expected, it is safe to use a value of type T instead.

What can we do with a value *v* of type $T \times U$?

• use *fst(v)* , which is a value of type T

• use *snd(v)* , which is a value of type U

If *w* is a value of type $A \times B$ then *fst(w)* has type A and can be used instead of *fst(v)*. Similarly *snd(w)* can be used instead of *snd(v)*. Therefore *w* can be used where *v* is expected.

Objects and subtyping

- Objects can be thought of as (extendible) records of fields and methods.
- That is why Square <: Shape and Circle <: Shape in

```
abstract class Shape {
  abstract float area( ); }
```

```
class Square extends Shape {
  float side;
  float area( ) {return (side * side); } }
```

```
class Circle extends Shape {
  float radius;
  float area( ) {return ( PI * radius * radius); } }
```

Objects

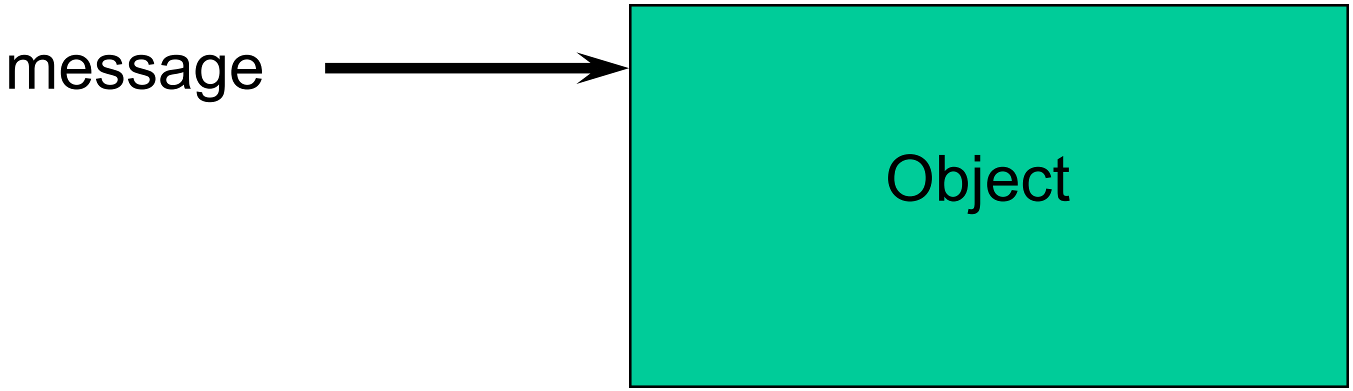
- An object consists of
  - hidden data
    - instance variables, also called member data
    - hidden functions also possible
  - public operations
    - methods or member functions
    - can also have public variables in some languages
- Object-oriented program:
  - Send messages to objects:
    - o → m (a) or o.m(a)

hidden data	
msg_1	$method_1$
. . .	. . .
msg_n	$method_n$

Objects can be extended by cloning or subclassing

Encapsulation

- Builder of a concept has detailed view
- User of a concept has "abstract" view
- Encapsulation is the mechanism for separating these two views
- The message concept facilitate loose coupling



Object-oriented programming

- Metaphor usefully ambiguous
  - Database, window, file, integer – all are objects
  - sequential or concurrent computation
  - distributed, sync. or async. Communication
- Programming methodology
  - organize concepts into objects and classes
  - build extensible systems
- Language concepts
  - encapsulate data and functions into objects
  - subtyping allows extensions of data types
  - inheritance allows reuse of implementation
  - dynamic lookup facilitate loose coupling

Dynamic Lookup – dynamic dispatch

- In object-oriented programming,

  object → message (arguments)

  object.method(arguments)

  code depends on object and message

  – Add two numbers x → add (y) or x.add(y)

    different add if x is integer or complex

- In conventional programming,

  operation (operands)

  meaning of operation is always the same

  – Conventional programming add (x, y)

    function add has fixed meaning

Dynamic Dispatch Example

```
class point {
    int c;
    int getColor() { return(c); }
    int distance() { return(0); }
}
class cartesianPoint extends point{
    int x, y;
    int distance() { return(x*x + y*y); }
}
class polarPoint extends point {
    int r, t;
    int distance() { return(r*r); }
    int angle() { return(t); }
}
```

Dynamic Dispatch Example

```
if (x == 0) {
    p = new point();
} else if (x < 0) {
    p = new cartesianPoint();
} else if (x > 0) {
    p = new polarPoint();
}
y = p.distance();
```

Which distance method is invoked?

- Invoked Method Depends on Type of Receiver!
  - if p is a point
    - return(0)
  - if p is a cartesianPoint
    - return(x*x + y*y)
  - if p is a polarPoint
    - return(r*r)

Dynamic dispatch

- If methods are overridden, and if the PL allows a variable of a particular class to refer to an object of a subclass, then method calls entail **dynamic dispatch**.

- Consider the Java method call O.M(E_1, …, E_n):
  - The compiler infers the type of *O*, say class *C*.
  - The compiler checks that class *C* is equipped with a method named *M*, of the appropriate type.
  - Nevertheless, it might turn out (at run-time) that the target object is actually of class *S*, a subclass of *C*.
  - If method *M* is overridden by any subclass of *C*, a run-time tag test is needed to determine the actual class of the target object, and hence which of the methods named *M* is to be called.

Overloading vs. Dynamic Dispatch

- Dynamic Dispatch
  - Add two numbers x.add (y)

    different add if x is integer, complex, ie. depends on the run-time type of x

- Overloading
  - add (x, y) function add has fixed meaning
  - int-add if x and y are ints, i.e. add (int x, int y)
  - real-add if x and y are reals i.e. add (float x, float y)

Important distinction:

Overloading is resolved at compile time,

Dynamic lookup at run time.

Comparison

- Traditional approach to encapsulation is through abstract data types
- Advantage
  - Separate interface from implementation
- Disadvantage
  - All ADTs are independent and at the same level
  - Not extensible in the way that OOP is
  - Not reusable in the way OOP is

Subtyping and Inheritance

- Interface
  - The external view of an object
- Subtyping
  - Relation between interfaces
- Implementation
  - The internal representation of an object
- Inheritance
  - Relation between implementations

Object Interfaces

- Interface
  - The messages understood by an object
- Example: point
  - x-coord : returns x-coordinate of a point
  - y-coord : returns y-coordinate of a point
  - move : method for changing location
- The interface of an object is its *type*.

Subtyping

- If interface A contains all of interface B, then A objects can also be used as B objects.

Point
- x-coord
- y-coord
- move

Colored_point
- x-coord
- y-coord
- color
- move
- change_color

- Colored_point interface contains Point
  - Colored_point is a *subtype* of Point

Inheritance

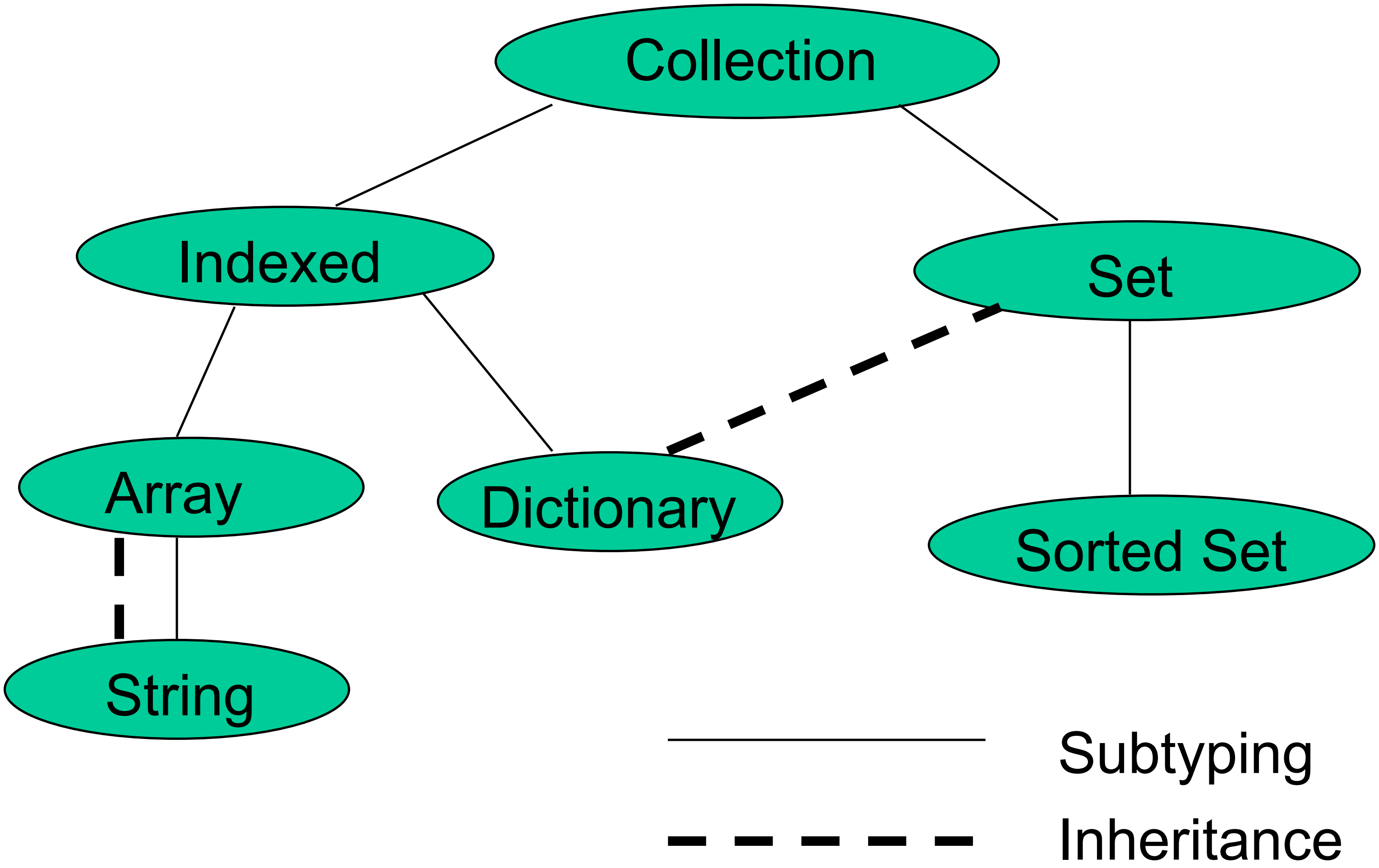
- Implementation mechanism
- New objects may be defined by reusing implementations of other objects

Example

```
class Point
    private
        float x, y
    public
        point move (float dx, float dy);
class Colored_point
    private
        float x, y; color c
    public
        point move(float dx, float dy);
       point change_color(color newc);
```

- **Subtyping**
  - Colored points can be used in place of points
  - Property used by client program
- **Inheritance**
  - Colored points can be implemented by reusing point implementation
  - Property used by implementor of classes

Subtyping differs from inheritance



Inheritance

- Implementation mechanism
- New objects may be defined by reusing implementations of other objects

- Note in Java and C# inheritance also implies a subtype relation !

- In C++ you can have inheritance without subtyping by extending a class private:
  - class Derived: private Base { … };
  -

Tennent’s Language Design principles and OOP

- The Principle of Abstraction
  - All major syntactic categories should have abstractions defined over them. For example, functions are abstractions over expressions

- We have seen abstractions over expressions, i.e. functions
- We have seen abstractions over commands, i.e. procedures
- What about abstractions over declarations?
  - Well Tennent, in 1981 saw that
  - Declabs Name(params) begin D end
  - Is exactly the notion of a class in the simula language !
  - “but this is not a widespread language construct”
  - Well not in 1981 ☺

Varieties of OO languages

- class-based languages
  - behaviour of object determined by its class
- object-based
  - objects defined directly
- multi-methods
  - operation depends on all operands

History

- Simula 1960's
  - Object concept used in simulation
- Smalltalk 1970's
  - Object-oriented design, systems
- C++ 1980's
  - Adapted Simula ideas to C
- Java 1990's
  - Distributed programming, internet
- C# 2000's
  - Combine the efficiency of C/C++ with the safety of Java
- Scala,F#, Swift, RUST - combine FP and OOP 2010's

Runtime Organization for OO Languages

How to represent/implement object oriented constructs such as **objects**, **classes, methods**, **instance variables** and **method invocation**

Some definitions for these concepts:

- An **object** is a group of instance variables to which a group of instance methods is attached.
- An **instance variable** is a named component of a particular object.
- An **instance method** is a named operation attached to a particular object and able to access that objects instance variables
- An **object class** (or just **class)** is a family of objects with similar instance variables and identical methods.

Runtime Organization for OO Languages

Objects are a lot like records, and instance variables are a lot like fields.
=> The representation of objects is similar to that of a record.

Methods are a lot like procedures.
=> Implementation of methods is similar to routines.

But… there are differences:

Objects have methods as well as instance variables, records only have fields (except in C#).

The methods have to somehow know what object they are associated with (so that methods can access the object's instance variables)

Example

A simple Java object (no inheritance)

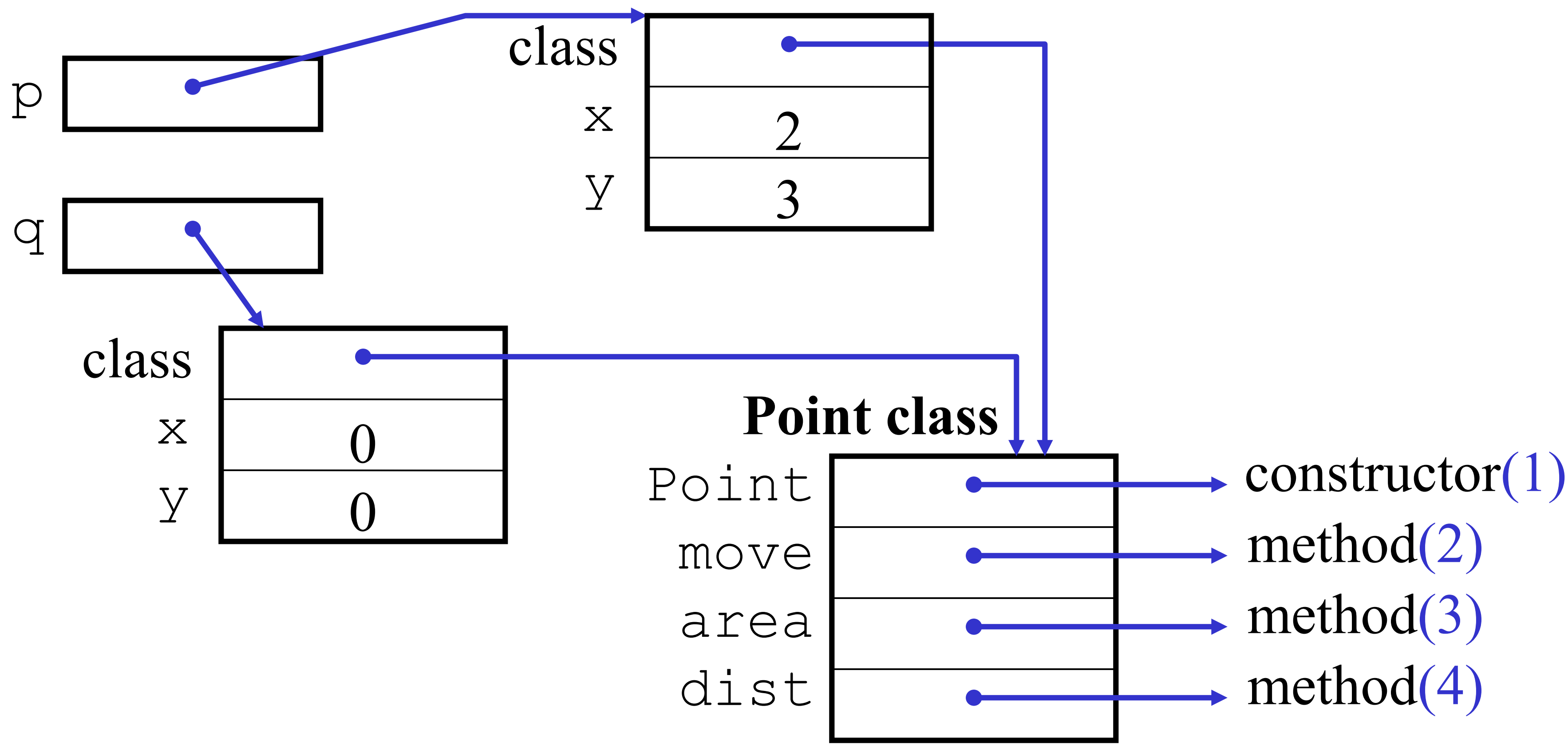
```
class Point {
  int x,y;
(1)public Point(int x, int y) {
     this.x=x; this.y=y;
  }

(2)public void move(int dx, int dy) {
     x=x+dx; y=y+dy;
  }

(3)public float area() { ...}
(4)public float dist(Point other) { ... }
}
```

Example

Representation of a simple Java object (no inheritance)

```
Point p = new Point(2,3);
Point q = new Point(0,0);
```

new allocates an object in the heap



Example

Points and other "shapes" (Inheritance)

```
abstract class Shape {
   int x,y; // "origin" of the shape
(S1) public Shape(int x, int y) {
       this.x=x; this.y=y;
   }

(S2) public void move(int dx, int dy) {
       x=x+dx; y=y+dy;
   }

   public abstract float area();
(S3) public float dist(Shape other) { ... }
}
```

Example

Points and other "shapes" (Inheritance)

```
class Point extends Shape {

(P1) public Point(int x, int y) {
        super(x,y);
     }

(P2) public float area() { return 0.0; }
}
```

Example

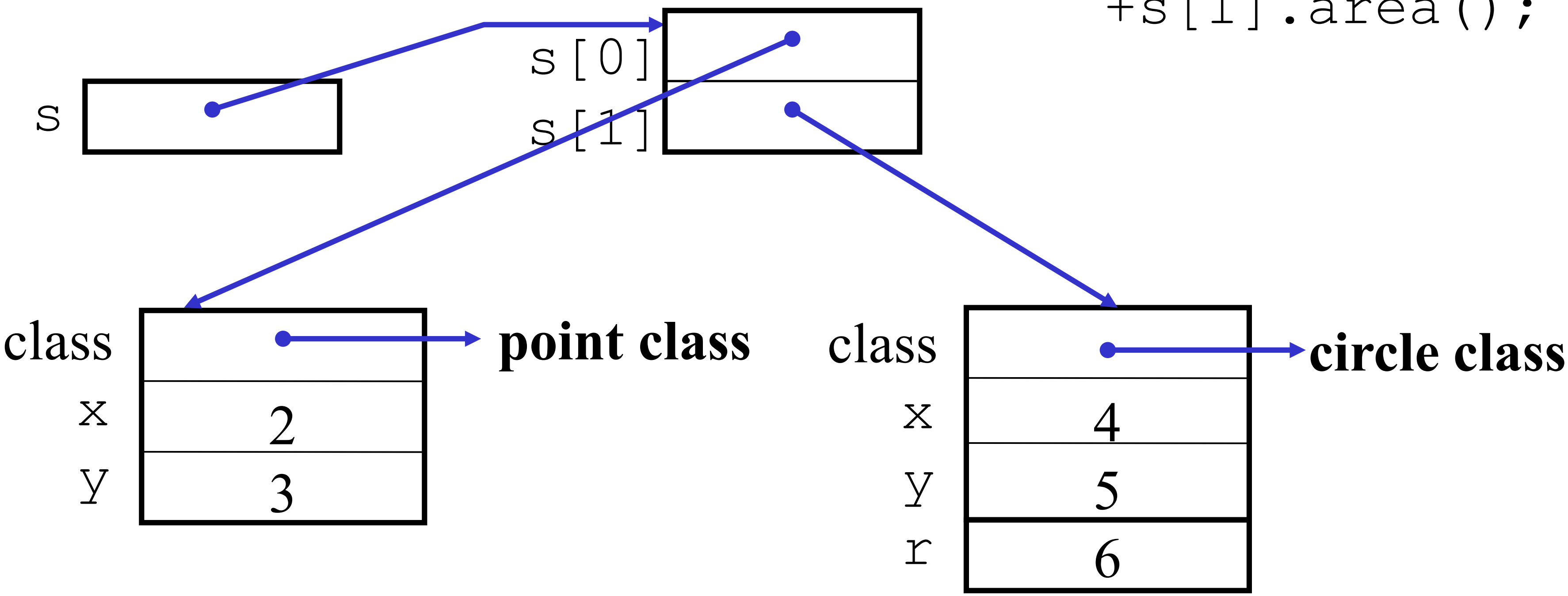
Points and other "shapes" (Inheritance)

```
class Circle extends Shape {
   int r;
(C1)public Circle(int x,int y,int r) {
      super(x,y); this.r = r;
   }

(C2)public int radius() { return r; }

(C3)public float area() {
      return 3.14 * r * r;
   }
}
```

Representation of Points and other “shapes” (Inheritance)

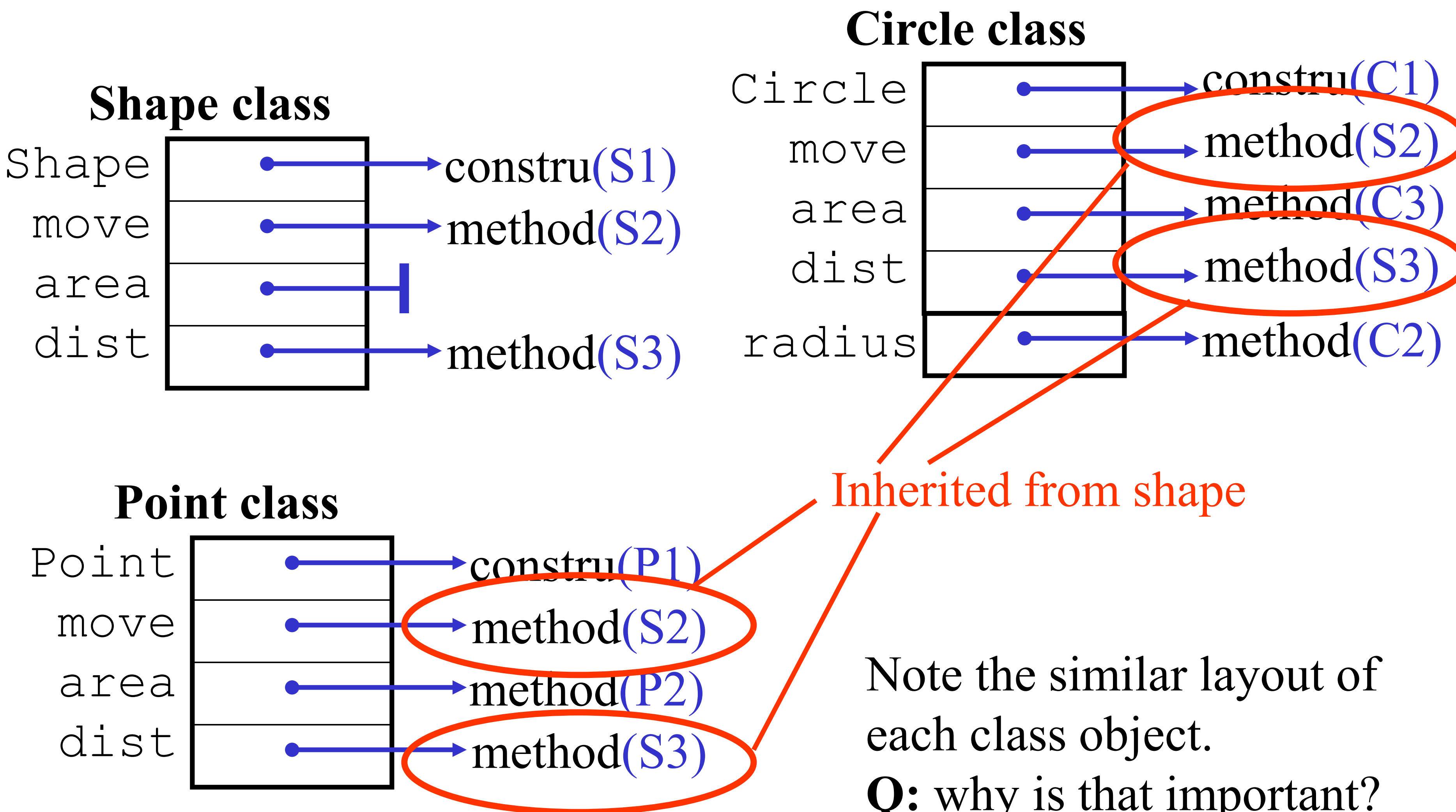
```
Shape[] s = new Shape[2];
s[0] = new Point(2,3);
s[1] = new Circle(4,5,6);
```

```
s[0].x = ...;
s[1].y = ...;
float areas =
   s[0].area()
  +s[1].area();
```



Note the similar layout between point and circle objects!

Representation of Points and other "shapes" (Inheritance)

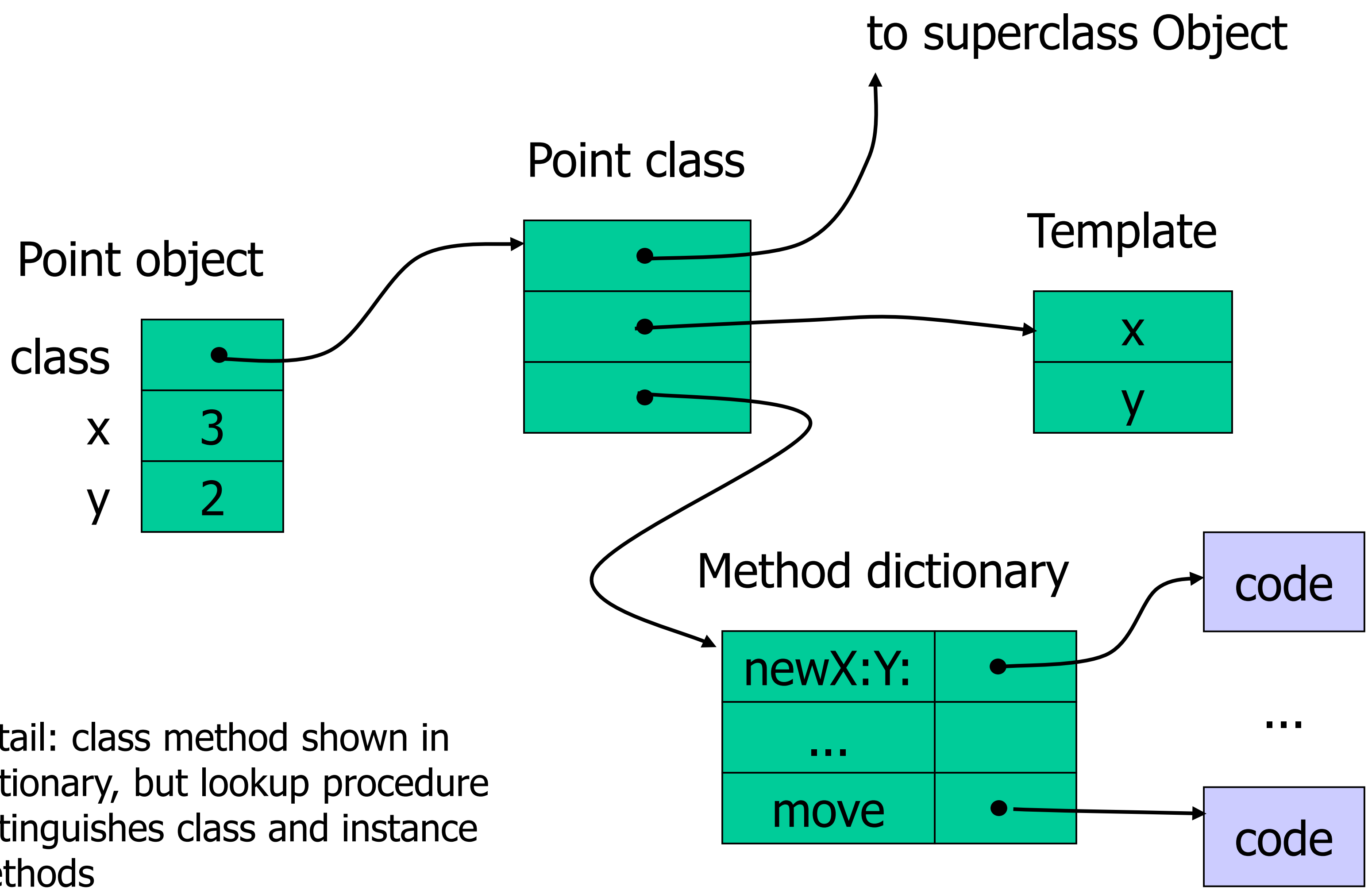


Note the similar layout of each class object.

Q: why is that important?

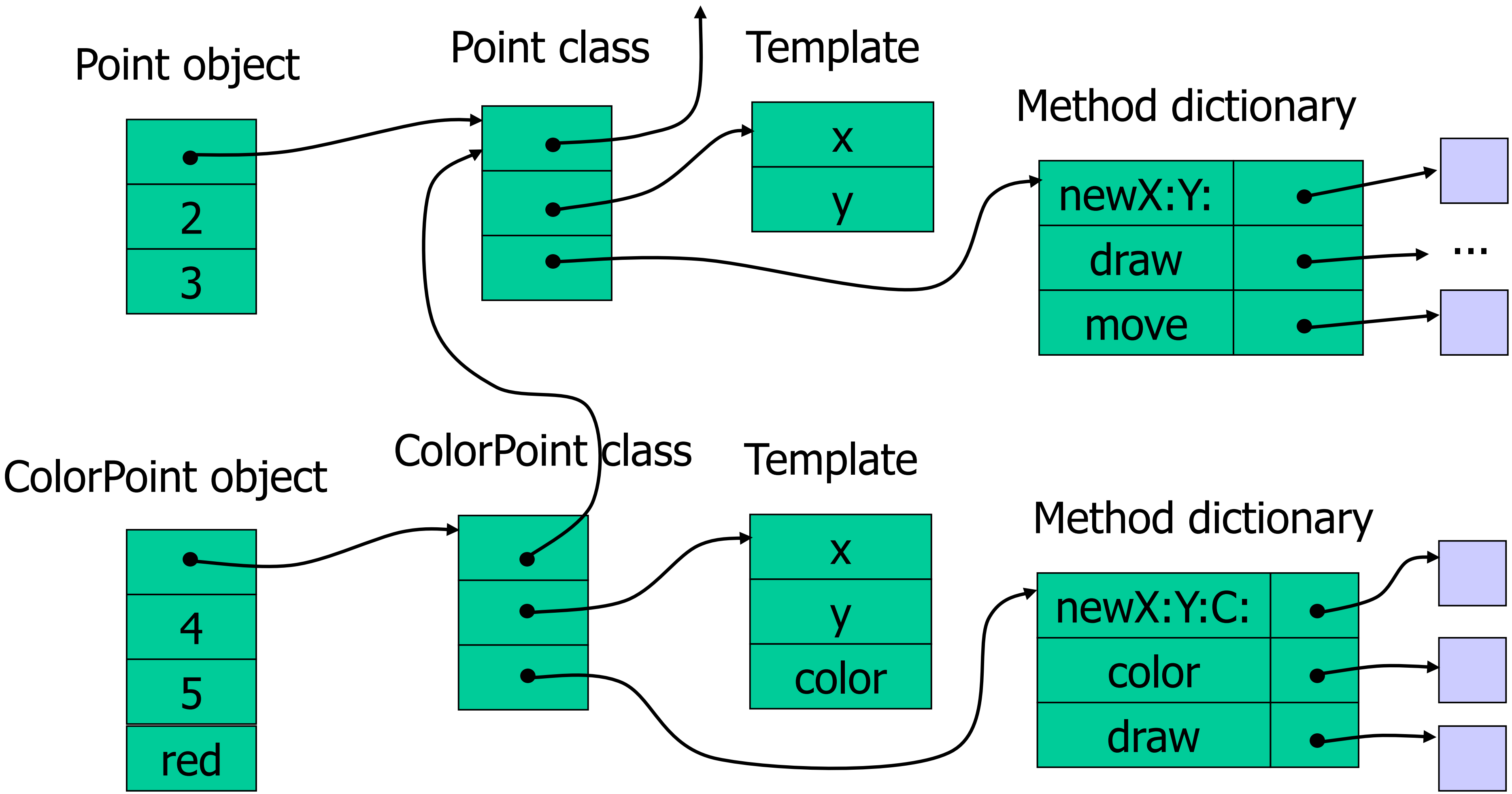
Q: why don't we need a pointer to the super class in a class object?

Alternative Run-time representation of point



Detail: class method shown in dictionary, but lookup procedure distinguishes class and instance methods

Alternative Run-time representation



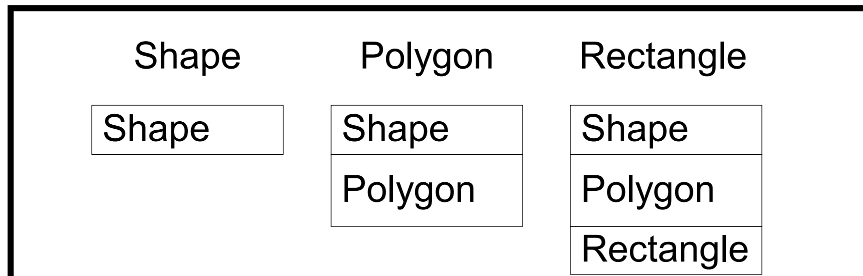
This is a schematic diagram meant to illustrate the main idea. Actual implementations may differ.

Multiple Inheritance

- In the case of single inheritance, each class may have one direct predecessor; multiple inheritance allows a class to have several direct predecessors.
- In this case the simple ways of accessing attributes and binding method-calls (shown previously) don't work.
- The problem: if class C inherits class A and class B the objects of class C cannot begin with attributes inherited from A and at the same time begin with attributes inherited from B.
- In addition to these implementation problems multiple inheritance also introduces problems at the language (conceptual) level.

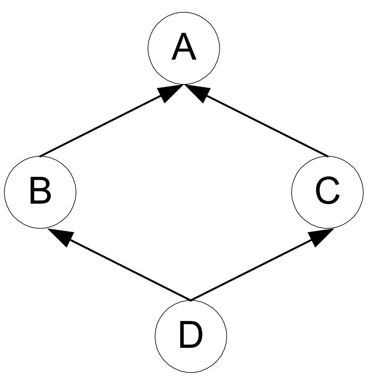
Object Layout

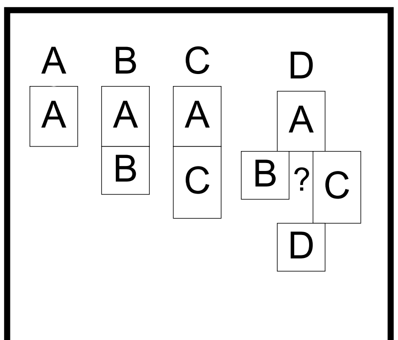
- The memory layout of the **object's fields**
- How to access a field if the dynamic type is unknown?
  - Layout of a type must be "compatible" with that of its supertypes
  - Easy for Single Inheritance hierarchies
    - The new fields are added at the end of the layout



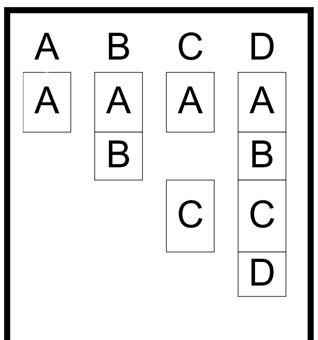
Layout in SI

- Hard for MI hierarchies

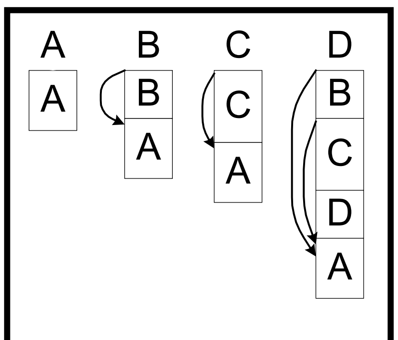




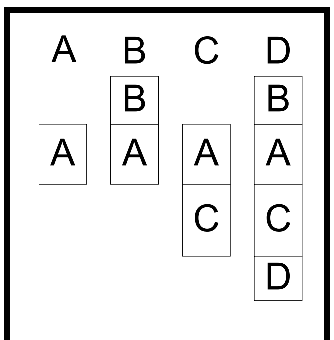
The difficulty in MI



Leave holes



C++ layout



BiDirectional layout

Dynamic (late) Binding

- Consider the method call:
  - x.f(a,b) where is f defined?
    in the class (type)of x? Or in a predecessor?
- If multiple inheritance is supported then the entire predecessor graph must be searched:
  - This costs a large overhead in dynamic typed languages like Smalltalk (normally these languages don't support multiple inheritance)
  - In static typed languages like Java, Eiffel, C++ the compiler is able to analyse the class-hierarchy (or more precise: the graph) for x and create a display-array containing addresses for all methods of an object (including inherited methods)
  - According to Meyer the overhead of this compared to static binding is at most 30%, and overhead decreases with complexity of the method
- If multi-methods are supported a forest like data structure has to be searched

Traits

- Some feel that single inheritance is too limiting
- Interface specification helps by forcing class to implement specified methods, but can lead to code duplication
- A trait is a collection of *pure* methods
- Can be thought of as an interface with implementation
- Classes “use” traits
- Traits can be used to supply the same methods to multiple classes in the inheritance hierarchy

Simple Example Using Traits

```
trait Similarity {
  def isSimilar(x: Any): Boolean
  def isNotSimilar(x: Any): Boolean = !isSimilar(x)
}
```

- This trait consists of two methods isSimilar and isNotSimilar
  - isSimilar is abstract
  - isNotSimilar is concrete but written in terms of isSimilar
- Classes that integrate this trait only have to provide a concrete implementation for isSimilar, isNotSimilar gets inherited directly from the trait

Simple Example Using Traits

```
class Point(xc: Int, yc: Int) extends
Similarity {
var x: Int = xc
var y: Int = yc
  def isSimilar(obj: Any) =
      obj.isInstanceOf[Point] &&
      obj.asInstanceOf[Point].x == x
}
```

Using Traits

- *Class = Superclass + State + Traits + Glue*
- A class provides it's own state
- It also provides "glue", which is the code that hooks the traits in
- Traits can satisfy each other's requirements for accessors
- A class is *complete* if all of the trait's requirements are met

- Languages with traits:
  - SmallTalk/Squeak/Pharo
  - Fortress
  - Scala
  - Swift
  - Kotlin
  - (Java8 – default methods on interfaces)

Implementation of Object Oriented Languages

- Implementation of Object Oriented Languages differs only slightly from implementations of block structured imperative languages
- Some additional work to do for the contextual analysis
  - Access control, e.g. private, public, protected directives
  - Subtyping can be tricky to implement correctly
- The main difference is that methods usually have to be looked up dynamically, thus adding a bit of run-time overhead
  - For efficiency some languages introduce modifiers like:
    - **final** (Java) or **virtual/override** (C#)
  - Multiple inheritance poses a bigger problem
  - Multi methods pose an even bigger problem

Larger Encapsulation Constructs

- Original motivation:
  - Large programs have two special needs:
    1. Some means of organization, other than simply division into subprograms
    2. Some means of partial compilation (compilation units that are smaller than the whole program)
- Obvious solution: a grouping of subprograms that are logically related into a unit that can be separately compiled (compilation units)
  - These are called encapsulations (or packages or modules)
  - Classes are too small (unless they allow true inner classes)

Encapsulation Constructs

- Why are Classes are too small and what is true inner classes?
- Originally mainstream OOP languages like C++ and Java had a flat namespace for classes
- But what if classes can be declared within classes?
- Java 1.1 introduced the notion of inner/nested classes:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

- Distinction between inner and nested classes
  - An inner class refer to an instance of the outer class in Java
  - A nested class is declared as static in Java
  - C# and C++ have static nested classes
    - remember in C# members are static unless declared to be overridable and inner classes cannot be declared overridable

Encapsulation Constructs

- Static nested classes introduce a form of namespace hierachy:

```
OuterClass.StaticNestedClass nestedObject =
    new OuterClass.StaticNestedClass();
```

Static nested classes only have access to static members and methods!

- Inner classes needs an instance of the outer class:

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject =
                        outerObject.new InnerClass();
```

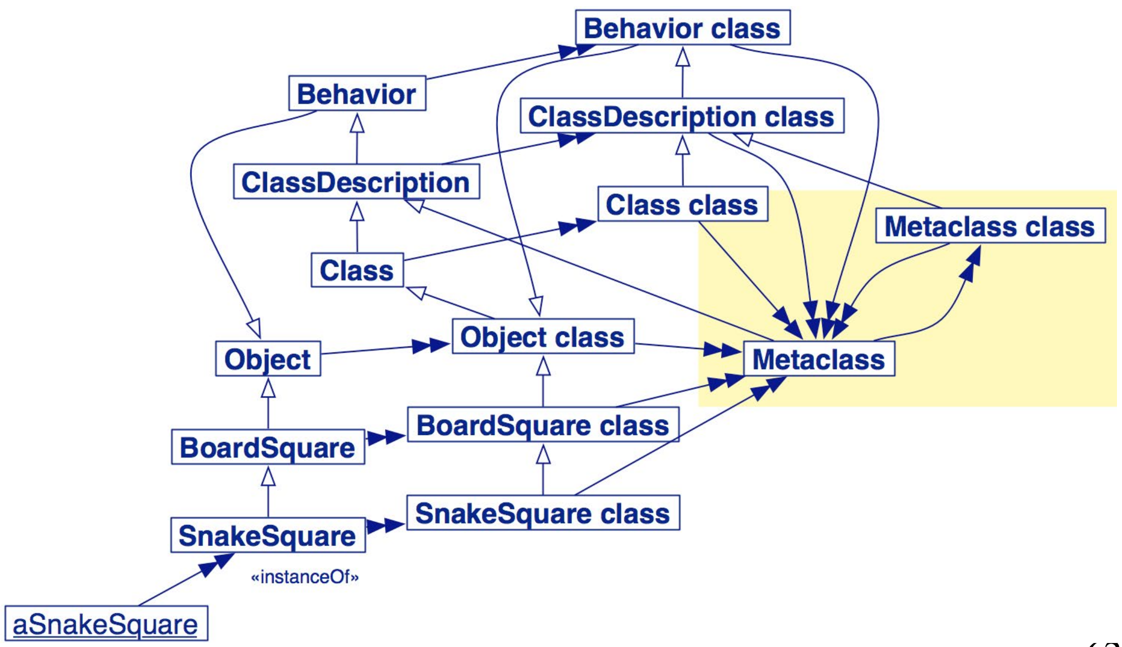
Inner classes have access to members and methods of the instance of the outer class

Note Java also allow class definitions in methods, but

Encapsulation Constructs

- However, many restrictions on inner classes in Java
  - A method can declare a local class,
    - but only access to variables declared as final – a restriction put to ensure a closure is not needed.
  - Classes cannot be treated as objects in Java.
- Other languages treat classes as first class objects
  - E.g. SmallTalk:

    Every object has a class and every Class is an object



Naming Encapsulations

- Large programs define many global names
- So we need a way to divide names into logical groupings
- A naming encapsulation is used to create a new scope for names
- C++ Namespaces
  - Can place each library in its own namespace and qualify names used outside with the namespace
- C# also includes namespaces
- In Java namespaces are called packages

Naming Encapsulations

- Java Packages
  - Packages can contain more than one class definition; classes in a package are partial friends
  - Clients of a package can use fully qualified name or use the `import` declaration
- Ada Packages
  - Packages are defined in hierarchies which correspond to file hierarchies
  - Visibility from a program unit is gained with the `with` clause
- SML Modules
  - Called **structure**; interface called **signature**
  - Interface specifies what is exported
  - Interface and structure may have different names
  - If structure has no signature, everything exported
  - Modules may be parameterized (**functors)**
  - Module system quite expressive

Modules

- Language construct for grouping related types, data structures, and operations
- Typically allows at least some encapsulation
  - Can be used to provide abstract types
- Provides scope for variable and subprogram names
- Typically includes interface stating which modules it depends upon and what types and operations it exports
- Compilation unit for separate compilation

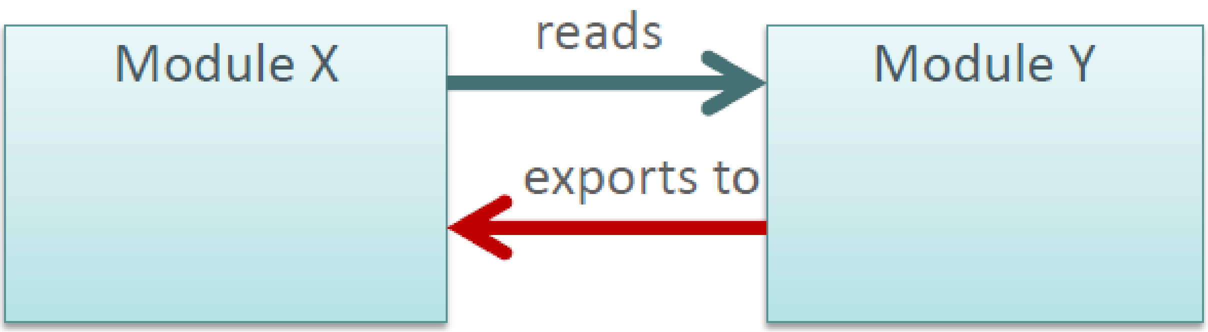
Encapsulation Constructs

- Encapsulation in C
  - Files containing one or more subprograms can be independently compiled
  - The interface is placed in a header file (.h)
  - Problem: the linker does not check types between a header and associated implementation
- Encapsulation in C++
  - Similar to C
  - Addition of friend functions that have access to private members of the friend class

Encapsulation Constructs

- Ada Package
  - Can include any number of data and subprogram declarations
  - Two parts: specification and body
  - Can be compiled separately
- C# Assembly
  - Collection of files that appears to be a single dynamic link library or executable
  - Larger construct than class; used by all .NET programming languages
- Java Module System (JSR 277/JSR376)
  - New deployment and distribution format
  - New language constructs:
    - module, import/export, provides/requires

Java 9 module system




```
module X {
  requires Y;
}
```

```
module Y {
  exports Q;
}
```

Issues for modules

- The target language usually has one name space
  - Generate unique names for modules
  - Some assemblers support local names per file
  - Use special characters which are invalid in the programming language to guarantee uniqueness
    - This is what Java does since the JVM has no nested classes

- Generate code for initialization
  - Modules may use items from other modules
  - Init before used
  - Init only once
  - Circular dependencies
    - How to initialize C once if module A uses module B and C, and B uses C
      - Compute a total order and init before use
      - Use special compile-time flag

Summary

- Abstract Data Types
  - Encapsulation
  - Invariants may be preserved
- Objects
  - Reuse
  - Subtyping
  - Inheritance
  - Dynamic dispatch
- Modules
  - Grouping (related) entities
  - Namespace management
  - Separate compilation

“I invented the term *Object-Oriented* and I can tell you I did not have C++ in mind.”

Alan Kay

Inventor of Smalltalk

Languages and Compilers (SProg og Oversættere)

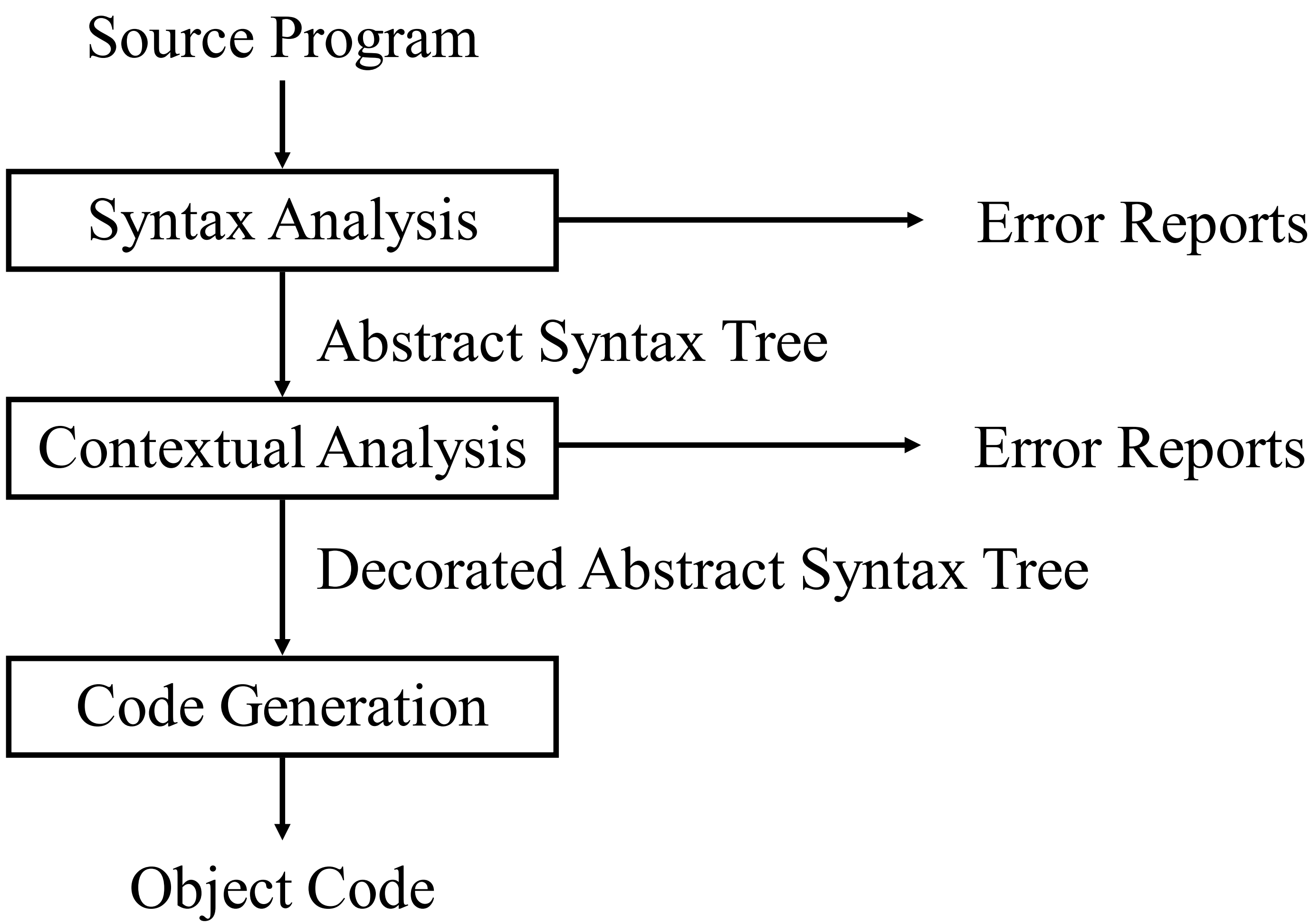
Lecture 20

Compiler Optimizations

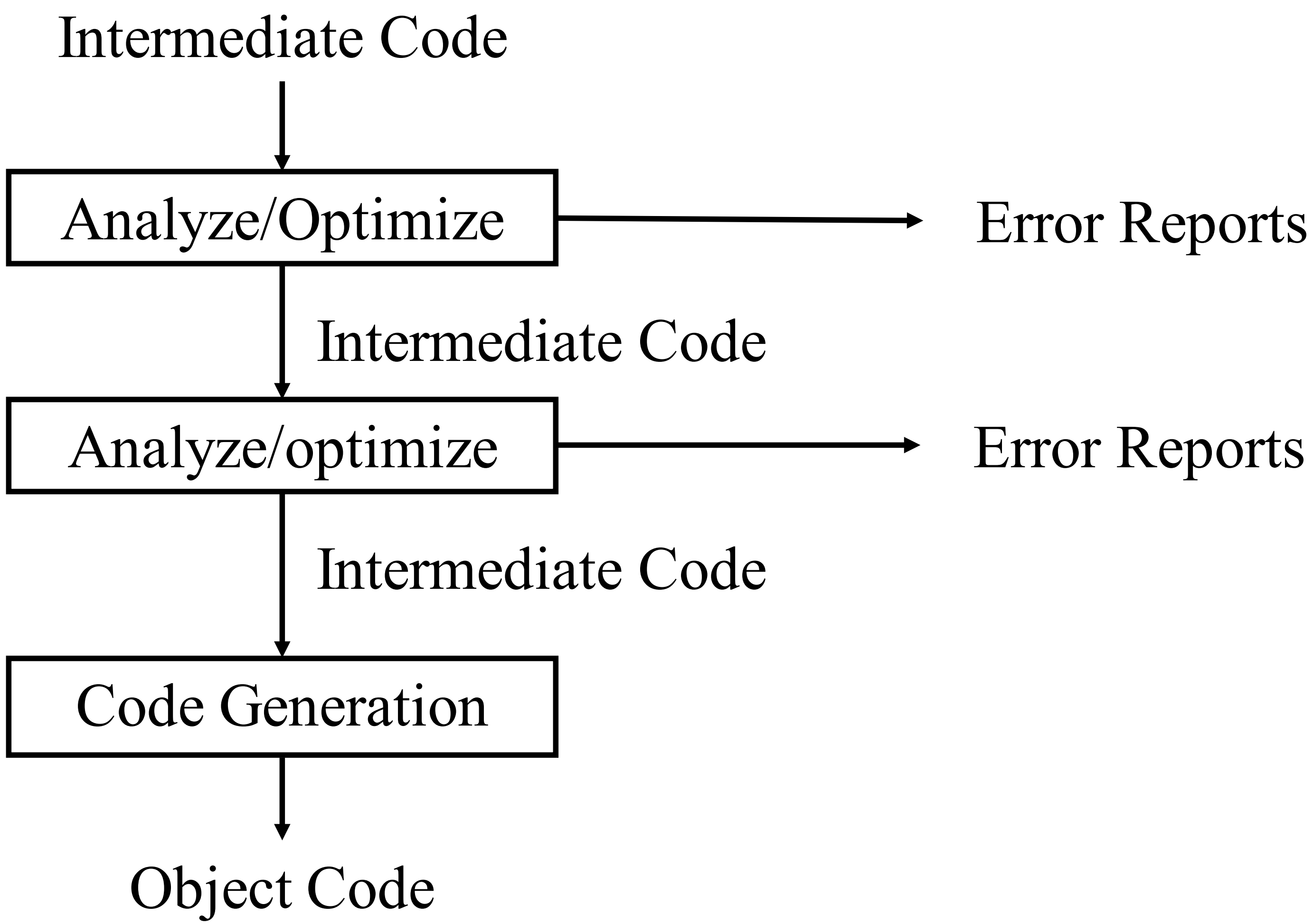
Bent Thomsen
Department of Computer Science
Aalborg University

With acknowledgement to Norm Hutchinson and Mooly Sagiv whose slides this lecture is based on.

The "Phases" of a Compiler



The “Phases” of a Compiler



Compiler Optimizations

The code generated by the code generators discussed so far are not very efficient:

- They compute some values at runtime that could be known at compile time
- They compute values more times than necessary
- They produce code that will never be executed

We can do better! We can do code transformations

- Code transformations are performed for a variety of reasons among which are:
  - To reduce the size of the code
  - To reduce the running time of the program
  - To take advantage of machine idioms
- Code optimizations include:
  - Peephole optimizatioons
  - Constant folding
  - Common sub-expression elimination
  - Code motion
  - Dead code elimination
- Mathematically, the generation of optimal code is undecidable.

Criteria for code-improving transformations

- Preserve meaning of programs (safety)
  - Potentially unsafe transformations
    - Associative reorder of operands
    - Movement of expressions and code sequences
    - Loop unrolling
- Must be worth the effort (profitability) and
  - on average, speed up programs

- **90/10 Rule:** Programs spend 90% of their execution time in 10% of the code. Identify and improve "hot spots" rather than trying to improve everything.

Peephole optimizations

- Recognition of program patterns that could be rewritten to produce faster code
- Can be done at several levels in the compiler:
  - AST rewrite
  - IR level rewrite
  - Bytecode
  - Target Code

- The general idea:
  - Pattern => replacement

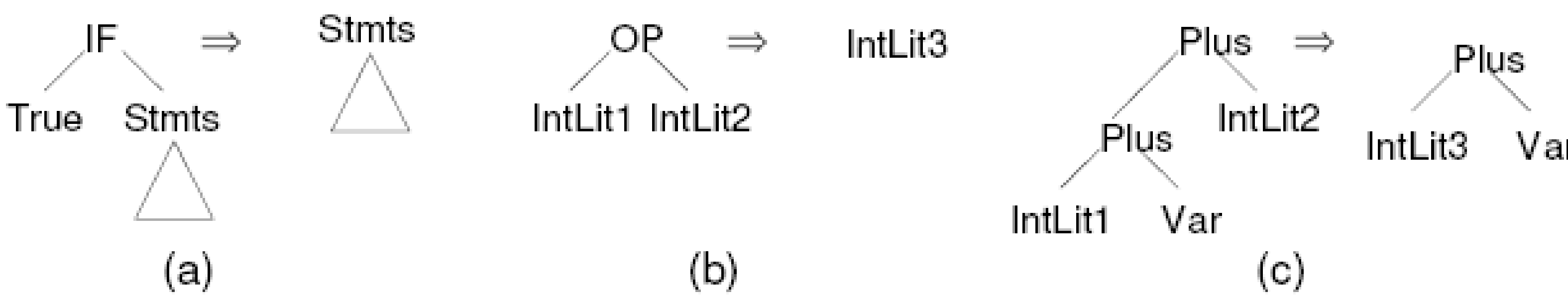


Figure 13.31: AST-Level Peephole Optimization

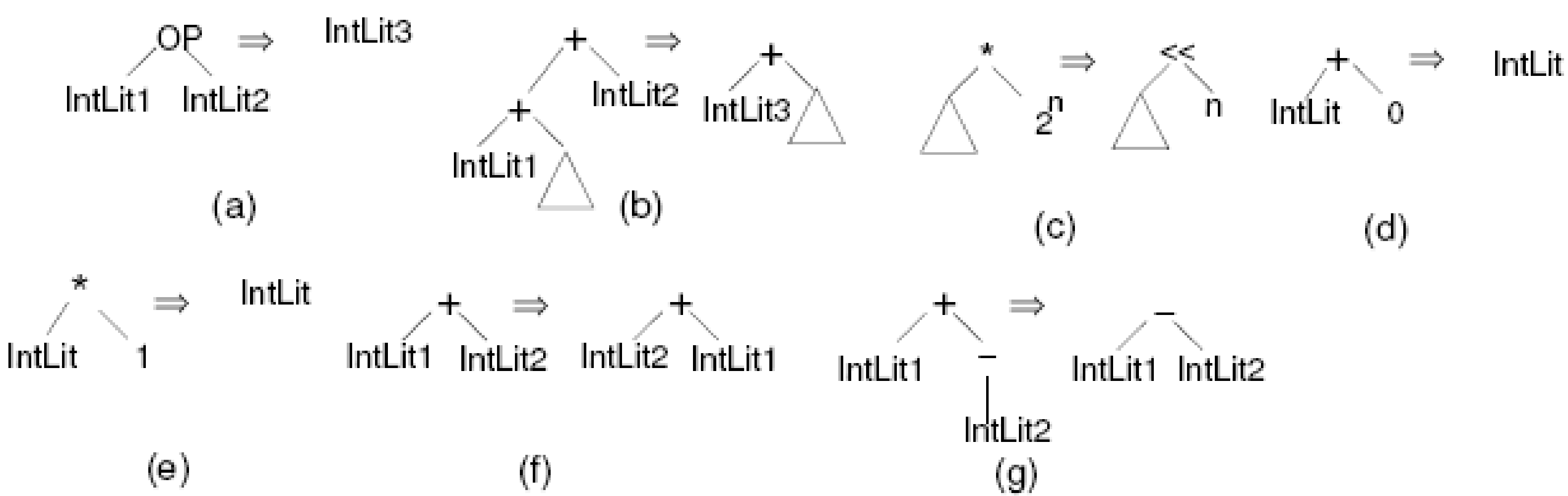


Figure 13.32: IR-Level Peephole Optimizations

```
ldc IntLit1   ⇒  ldc IntLit3
ldc IntLit2
iadd
```

(a)

```
ldc IntLit1
{Bytecode            {Bytecode
 sequence        ⇒    sequence
 for operand}         for operand}
iadd                 ldc IntLit3
ldc IntLit2          iadd
iadd
```

(b)

```
ldc 2^n       ldc n
imul    ⇒     ishl
```

(c)

```
ldc IntLit    ⇒  ldc IntLit
iconst_0
iadd
```

(d)

```
ldc IntLit    ⇒  ldc IntLit
iconst_1
imul
```

(e)

```
ldc IntLit1   ⇒  ldc IntLit2
ldc IntLit2      ldc IntLit1
iadd             iadd
```

(f)

```
ldc IntLit1   ⇒  ldc IntLit1
ldc IntLit2      ldc IntLit2
ineg             isub
iadd
```

(g)

Figure 13.33: Bytecode-Level Peephole Optimizations

```
    beq $reg,$0,L1    ⇒    bneq $reg,$0,L2
    b L2
L1:                        L1:
```

(a)

```
    b L1    ⇒
L1:              L1:
```

(b)

```
    b L1    ⇒        b L2
L1: b L2         L1: b L2
```

(c)

```
move $reg,$reg    ⇒    (nothing)
```

(d)

```
sw $reg,loc    ⇒    sw $reg,loc
lw $reg,loc
```

(e)

Figure 13.34: Code-Level Peephole Optimizations

Constant folding

- Consider:

```
static double pi = 3.1416;
double volume = 4/3 * pi * r * r * r;
```

- The compiler could compute 4 / 3 * pi as 4.1888 before the program runs. This saves how many instructions?

- What is wrong with the programmer writing

  `4.1888 * r * r * r`?

Constant folding II

- Consider:

```
struct { int y, m, d; } holidays[6];
holidays[2].m = 12;
holidays[2].d = 25;
```

- If the address of `holidays` is `x`, what is the address of `holidays[2].m`?
- Could the programmer evaluate this at compile time? Safely?

Common sub-expression elimination

- Consider:

```
int t = (x - y) * (x - y + z);
```

- Computing x – y takes three instructions, could we save some of them?

Common sub-expression elimination II

```
int t = (x - y) * (x - y + z);
```

Naïve code:

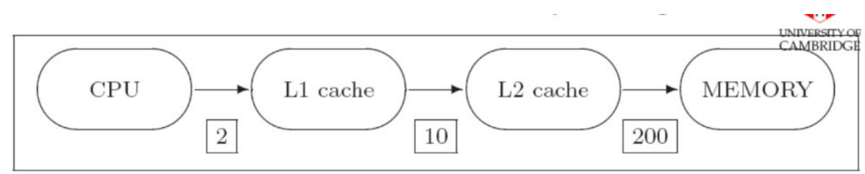
iload x
iload y
isub
iload x
iload y
isub
iload z
iadd
Imult
istore t

Common sub-expression elimination II
Programmer tries to be clever

```
int tmp = (x - y)
int t = tmp * (tmp + z);
```

Naïve code:	New code:
iload x	iload x
iload y	iload y
isub	isub
iload x	istore tmp
iload y	iload tmp
isub	iload tmp
iload z	iload z
iadd	iadd
Imult	Imult
istore t	istore t

Is this code better or worse?



Common sub-expression elimination II

```
int t = (x - y) * (x - y + z);
```

Naïve code:

```
iload x
iload y
isub
iload x
iload y
isub
iload z
iadd
Imult
istore t
```

Better code:

```
iload x
iload y
isub
dup
iload z
iadd
Imult
istore t
```

Common sub-expression elimination III

- Consider:

```
struct { int y, m, d; } holidays[6];
holidays[i].m = 12;
holidays[i].d = 25;
```

- The address of `holidays[i]` is a common subexpression.

Common sub-expression elimination IV

- But, be careful!

```
int t = (x - y++) * (x - y++ + z);
```

- Is `x - y++` still a common sub-expression?

Code motion

- Consider:

```
char name[3][10];
for (int i = 0; i < 3; i++) {
     for (int j = 0; j < 10; j++) {
          name[i][j] = 'a';
```

- Computing the address of `name[i][j]` is `address[name] + (i * 10) + j`
- Most of that computation is constant throughout the inner loop

```
address[name] + (i * 10)
```

Code motion II

- You can think of this as rewriting the original code:

```
char name[3][10];
for (int i = 0; i < 3; i++) {
     for (int j = 0; j < 10; j++) {
          name[i][j] = 'a';
```

as

```
char name[3][10];
for (int i = 0; i < 3; i++) {
     char *x = &(name[i][0]);
     for (int j = 0; j < 10; j++) {
          x[j] = 'a';
```

Dead code elimination

- Consider:

```
int f(int x, int y, int z)
{
    int t = (x - y) * (x - y + z);
    return 6;
}
```

- Computing `t` takes many instructions, but the value of `t` is never used.
- We call the value of `t` "dead" (or the variable `t` dead) because it can never affect the final value of the computation. Computing dead values and assigning to dead variables is wasteful.

Dead code elimination II

- But consider:

```
int f(int x, int y, int z)
{
    int t = x * y;
    int r = t * z;
    t = (x - y) * (x - y + z);
    return r;
}
```

- Now `t` is only dead for part of its existence. Hmm…

Optimization implementation

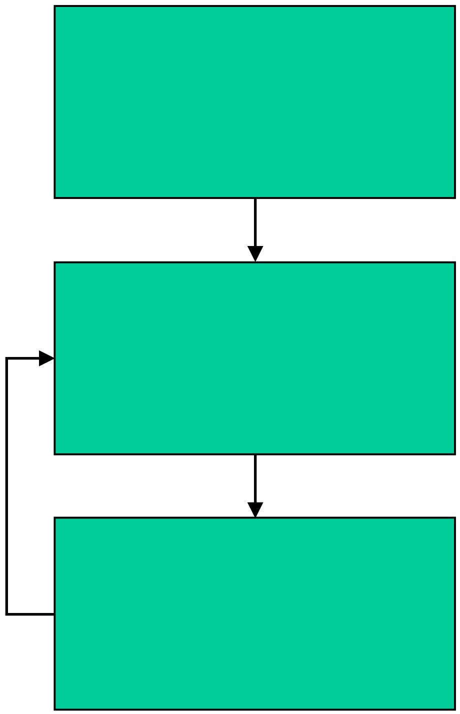
- What do we need to know in order to apply an optimization?
  - –Constant folding
  - –Common sub-expression elimination
  - –Code motion
  - –Dead code elimination
- Is the optimization correct or safe?
- Is the optimization an improvement?
- What sort of analyses do we need to perform to get the required information?

Control-Flow Analysis

- The purpose of Control-Flow Analysis is to determine the control structure of a program
  - determine possible control flow paths
  - find basic blocks and loops
- A Basic Block (BB) is a sequence of instructions entered only at the beginning and left only at the end.
- The Control-Flow Graph (CFG) of a program is a directed graph G=(N, E) whose nodes N represent the basic blocks in the program and whose edges E represent transfers of control between basic blocks.

Basic blocks

- A basic block is a sequence of instructions entered only at the beginning and left only at the end.
- A flow graph is a collection of basic blocks connected by edges indicating the flow of control.



Finding basic blocks

```
    iconst_1
    istore 2
    iconst_2
    istore 3
Label_1:
    iload 3
    iload 1
    if_icmplt  Label_4
    iconst_0
    goto Label_5
Label_4:
    iconst_1
Label_5:
    ifeq Label_2
    iload 2
    iload 3
    imul
    dup
    istore 2
    pop
Label_3:
    iload 3
    dup
    iconst_1
    iadd
    istore 3
    pop
    goto Label_1
Label_2:
    iload 2
    ireturn
```

Finding basic blocks II

```
        iconst_1
        istore 2
        iconst_2
        istore 3
```

```
Label_1:
        iload 3
        iload 1
        if_icmplt Label_4
```

```
        iconst_0
        goto Label_5
```

```
Label_4:
        iconst_1
```

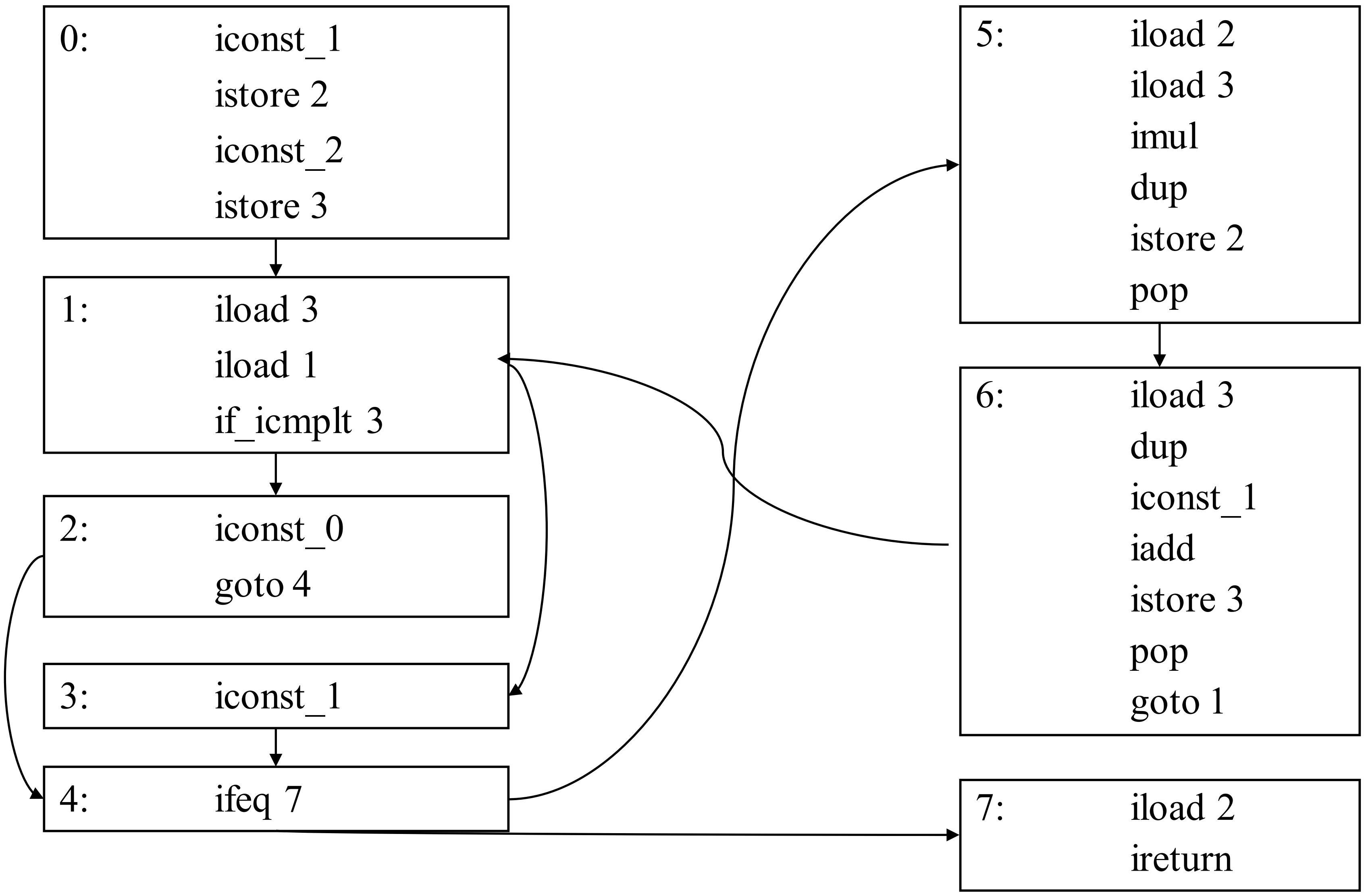
```
Label_5:
        ifeq Label_2
```

```
        iload 2
        iload 3
        imul
        dup
        istore 2
        pop
```

```
Label_3:
        iload 3
        dup
        iconst_1
        iadd
        istore 3
        pop
        goto Label_1
```

```
Label_2:
        iload 2
        ireturn
```

Flow graphs


0: iconst_1
istore 2
iconst_2
istore 3
1: iload 3
iload 1
if_icmplt 3
2: iconst_0
goto 4
3: iconst_1
4: ifeq 7
5: iload 2
iload 3
imul
dup
istore 2
pop
6: iload 3
dup
iconst_1
iadd
istore 3
pop
goto 1
7: iload 2
ireturn

```
procedure formBasicBlocks( )
    leaders ← { first instruction in stream }
    foreach instruction s in the stream do                                  (23)
        targets ← { distinct targets branched to from s }                   (24)
        if |targets| > 1
        then
            foreach t ∈ targets do  leaders ← leaders ∪ { t }
    foreach l ∈ leaders do                                                  (25)
        block(l) ← { l }
        s ← next instruction after l
        while s ∉ leaders and s ≠ ⊥ do
            block(l) ← Block(l) ∪ { s }
            s ← Next instruction in stream
end
```

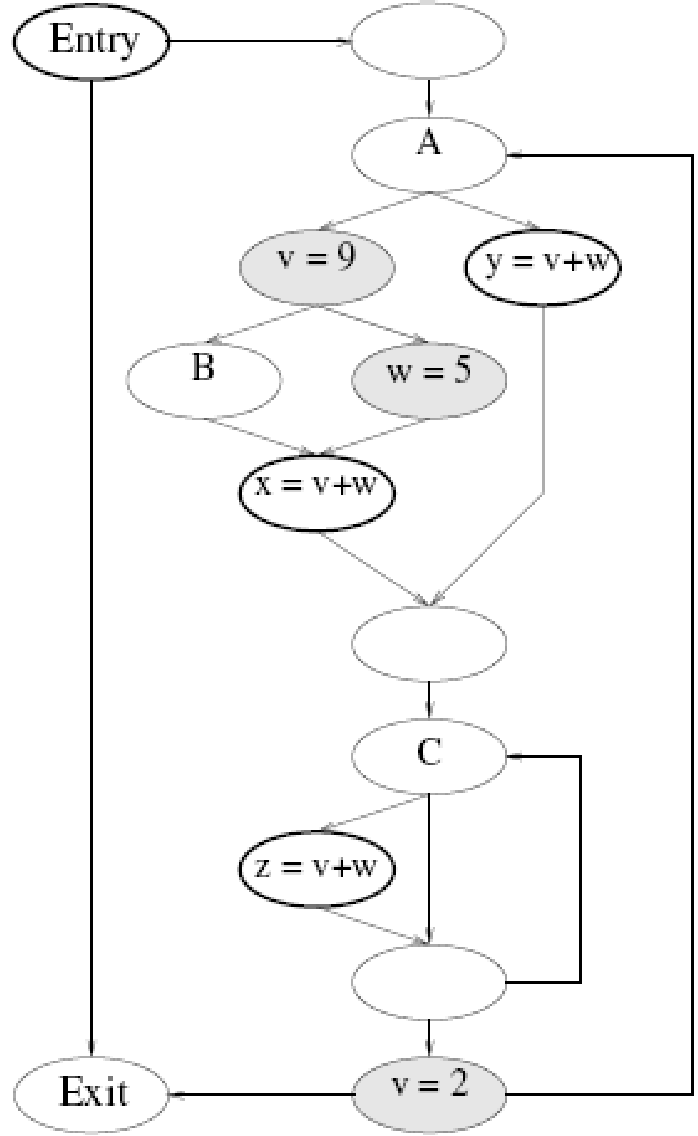
Figure 14.7: Partitioning of instructions into basic blocks. The $\perp$ value in pseudocode means *undefined* and is typically denoted as **null** in most programming languages.

Data-Flow Analysis

- The purpose of Data-Flow Analysis is to provide global information about how a procedure manipulates its data.
- Examples:
  - Live variable analysis
    - Which variable are still alive?
    - Needed for: register allocation, dead-code elimination
  - Reaching definitions
    - What points in program does each variable definition reach?
    - Needed for: copy- and constant propagation
- Available expressions
  - Which expressions computed earlier still have same value?
  - Needed for: common sub-expression elimination.

```
u ← 5
repeat
    if r
    then
        v ← 9
        if p
        then  u ← 6
        else  w ← 5
        x ← v + w
    else  y ← v + w
    u ← 7
    repeat
        if q
        then
            z ← v + w   (83)
    until r
    v ← 2
until s
```

(a)



(b)

Figure 14.41: (a) A program; (b) Its control flow graph.

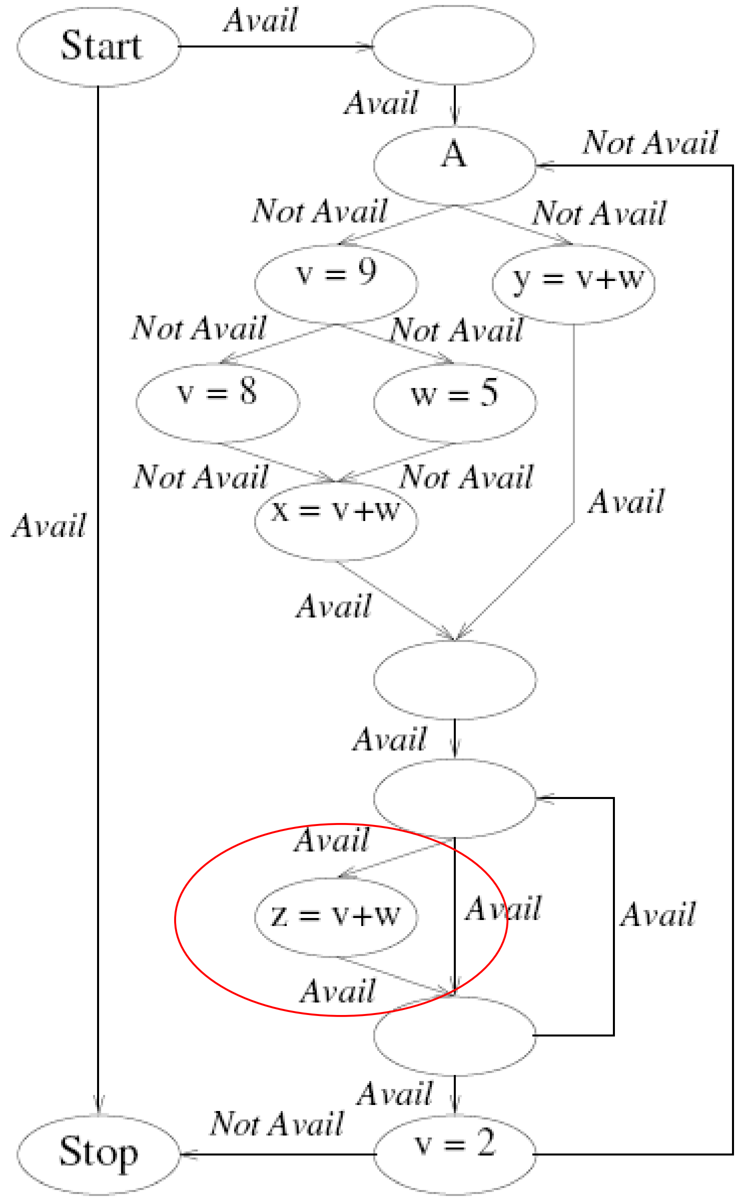


Figure 14.42: Solution throughout the flow graph of Figure 14.41(b) for the availability of expression $v + w$.

Optimizations within a BB

- Everything you need to know is easy to determine
- For example: live variable analysis
  - –Start at the end of the block and work backwards
  - –Assume everything is live at the end of the BB
  - –Copy live/dead info for the instruction
  - –If you see an assignment to x, then mark x "dead"
  - –If you see a reference to y, then mark y "live"

```
5:        iload 2
          iload 3
          imul
          dup
          istore 2
          pop
```

live: 1, 2, 3
live: 1, 3
live: 1, 3
live: 1, 3
live: 1, 3
live: 1, 2, 3
live: 1, 2, 3

Global optimizations

- Global means "between basic blocks"
- We must know what happens across block boundaries
- For example: live variable analysis
  - The liveness of a value depends on its later uses perhaps in other blocks
  - What values does this block define and use?

```
5:      iload 2
        iload 3
        imul
        dup
        istore 2
        pop
```

Define: 2
Use: 2, 3

Global live variable analysis

- We define four sets for each BB
  - def == variables with defined values
  - use == variables used before they are defined
  - in == variables live at the beginning of a BB
  - out == variables live at the end of a BB
- These sets are related by the following equations:
  - $in[B] = use[B] \cup (out[B] - def[B])$
  - $out[B] = \cup_S in[S]$ where S is a successor of B

Solving data flow equations

- Iterative solution:
  - Start with empty set
  - Iteratively apply constraints
  - Stop when we reach a fixed point

For all instructions in[I] = out[I] = ∅

Repeat

For each instruction I

$$\text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I]$$

For each basic block B

$$\text{out}[B] = \bigcup_{B' \in \text{succ}(B)} \text{in}[B']$$

Until no new changes in sets

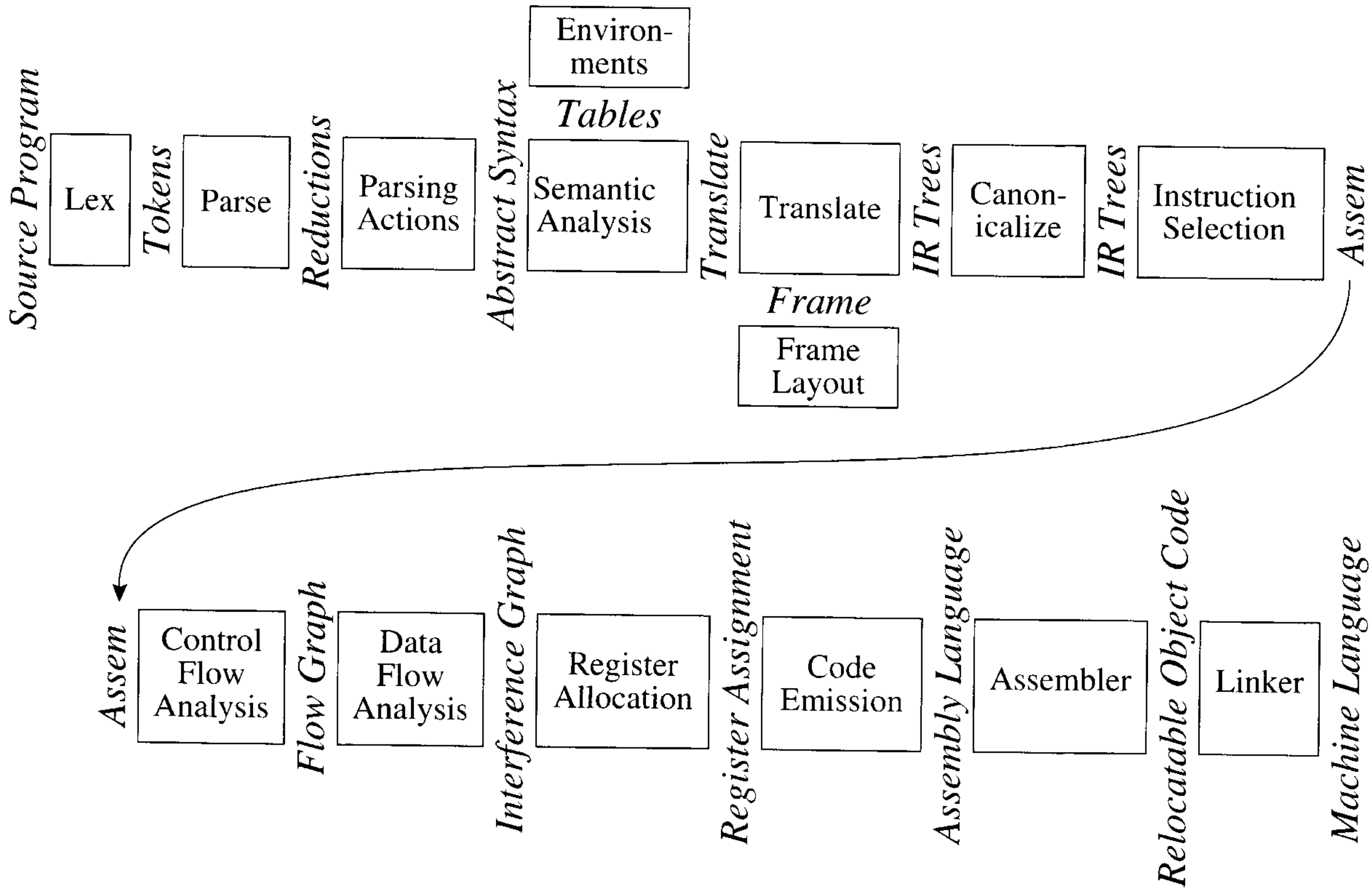
Dead code elimination

- Armed with global live variable information we redo the local live variable analysis with correct liveness information at the end of the block out[B]
- Whenever we see an assignment to a variable that is marked dead, we eliminate it.

Static Analysis

- Automatic derivation of static properties which hold on every execution leading to a program location
- Example Static Analysis Problems
  - Live variables
  - Reaching definitions
  - Expressions that are "available"
  - Dead code
  - Pointer variables that never point into the same location
  - Points in the program in which it is safe to free an object
  - An invocation of a virtual method whose address is unique
  - Statements that can be executed in parallel
  - An access to a variable which must be in cache
  - Integer intervals
  - Security properties
  - WCET and Schedulability
  - …

A somewhat more complex compiler



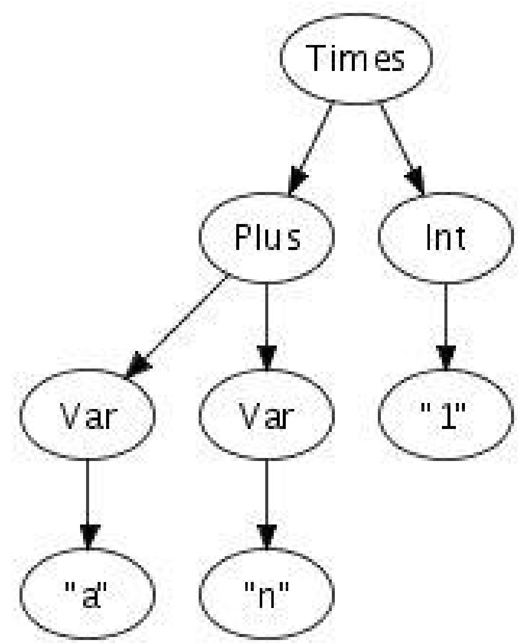
Learning More about Optimizations

- Read chapter 9-12 in the new Dragon Book
  - Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Addison-Wesley, ISBN 0-321-21091-3
- Read the ultimate reference on program analysis
  - Principles of Program Analysis Flemming Nielson, Hanne Riis Nielson, Chris Hankin: Principles of Program Analysis. Springer (Corrected 2nd printing, 452 pages, ISBN 3-540-65410-0), 2005.
- Use one of the frameworks:
  - **Soot: a Java Optimization Framework**
    - http://www.sable.mcgill.ca/soot
  - WALA: The T. J. Watson Libraries for Analysis
    - http://wala.sourceforge.net/wiki/index.php/Main_Page

Pause

Remember the exercises before this course?

- 2.Write a Java program that implements a data structure for the following tree
- 3.Extend your Java program to traverse the tree depth-first and print out information in nodes and leaves as it goes along.
- 4.Write a Java program that can read the string "a + n * 1" and produce a collection of objects containing the individual symbols when blank spaces are ignored (or used as separator).



Remember the exercises before this course?

- 2.Make a drawing or description of the phases (internals) of a compiler (without reading the books or searching the Internet) – save this for comparison with your knowledge after the course.
- 4.Create a list of language features group members would like in a new language. Are any of these features in conflict with each other? How would you prioritize the features?
- 5.Discuss what is needed to define a new programming language. Write down your conclusions for comparison with your knowledge after the course.

Organization of a Compiler

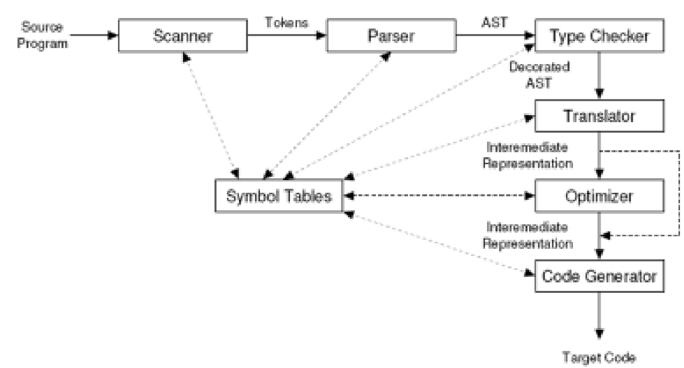


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

4

What was this course about?

- Programming Language Design
  - Concepts and Paradigms
  - Ideas and philosophy
  - Syntax and Semantics
- Compiler Construction
  - Tools and Techniques
  - Implementations
  - The nuts and bolts

Curricula

Studie ordningen i de gode gamle dage ☺

The purpose of the course is contribute to the student gaining knowledge of important principles in programming languages and understanding of techniques for describing and compiling programming languages.

Sprog og oversættelse / Language and Compiler Construction (SPO)
Omfang: 5 ECTS-point.
Forudsætninger: Programmeringserfaring svarende til projektenheden på 3. semester samt kendskab til imperativ og objektorienteret programmering svarende til 1. - og 2. semesters kurser i programmering.

Mål:

Viden:
Den studerende skal opnå viden om væsentlige principper i programmeringssprog, samt forståelse af teknikker til beskrivelse og oversættelse af sprog generelt, herunder:

- Abstraktionsprincippet, kontrol- og datastrukturer, blokstruktur og scopebegrebet, parametermekanismer og typeækvivalens
- Oversættelse, herunder leksikalsk, syntaktisk, og statisk semantisk analyse, samt kodegenering
- Køretids-omgivelser, herunder lagerallokering samt strukturer til understøttelse af procedurer og funktioner

Færdigheder:
Den studerende skal opnå følgende færdigheder:

- Kunne redegøre for de berørte teknikker og begreber inden for sprogdesign og oversætterkonstruktion ved brug af fagets terminologi og notation for beskrivelse og implementation af programmeringssprog
- Kunne redegøre for hvordan implementations teknikker influerer sprog design
- Kunne ræsonnere datalogisk om og med de berørte begreber og teknikker

Kompetencer: Den studerende skal kunne beskrive, analysere og implementere programmeringssprog og skal kunne redegøre for de enkelte faser og sammenhængen mellem faserne i en oversætter

Undervisningsform: Kursus
Prøveform: Mundtlig eller skriftlig prøve
Bedømmelse: Ekstern bedømmelse efter 7-trins-skala
Vurderingskriterier: Se Rammestudieordningen.

What is expected of you at the end?

- One goal for this course is for you to be able to explain concepts, techniques, tools and theories to others
  - Your future colleagues, customers and boss
  - (especially me and the examiner at the exam ;-)
- That implies you have to
  - Understand the concepts and theories
  - Know how to use the tools and techniques
  - Be able to put it all together
- I.e. You have to know and know that you know

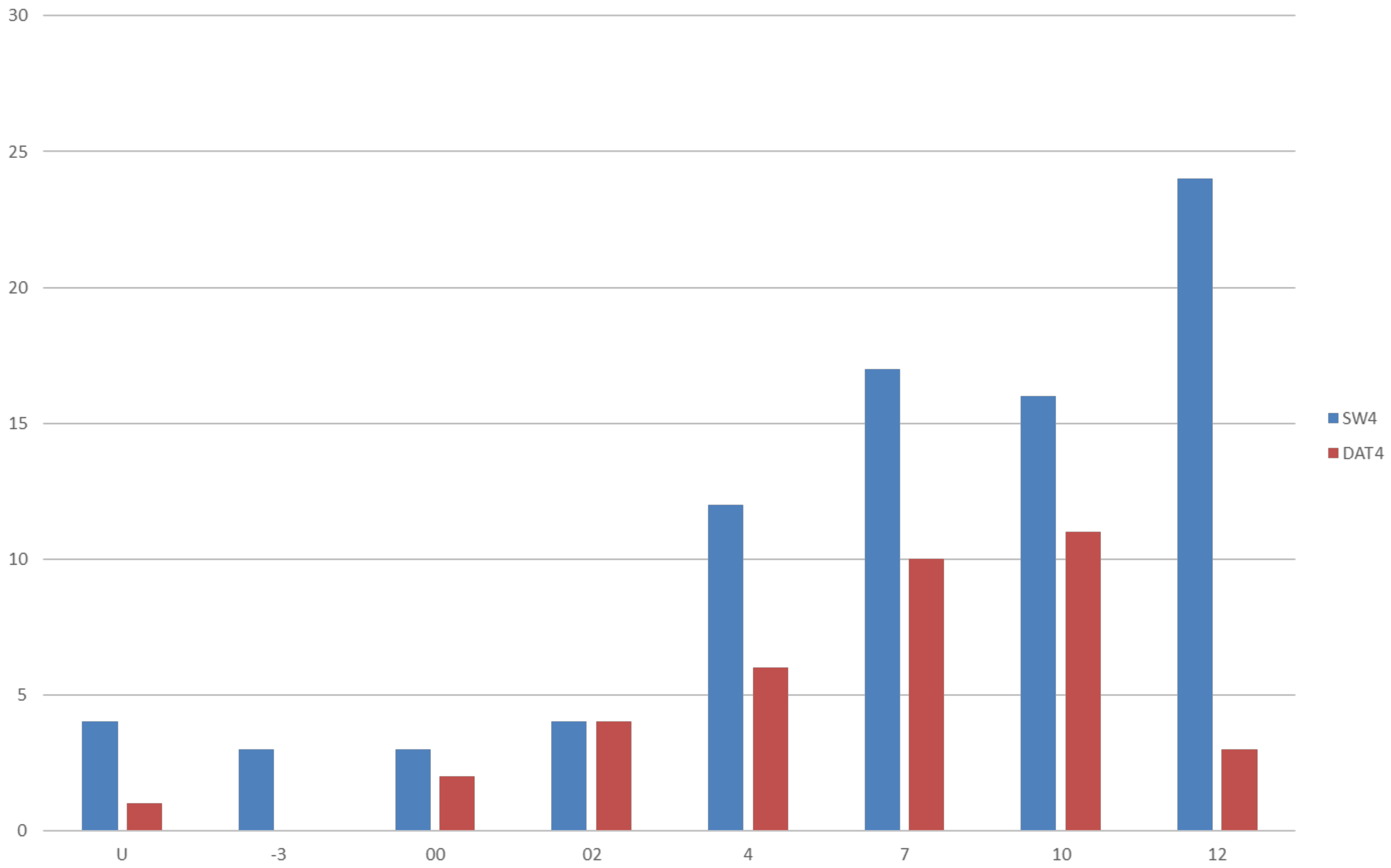
Exam

- 15 minute video presentation exam
  - To be recorded in 1 hours
  - Your subject and questions will be released in DE
- Subjects are already published
  - So you know roughly what we will ask you !!
  - For each published question there will be some questions you do not know before hand.
  - For each question there will be a set of slides available that you can choose to use for your presentation
    - note you do not need to use all the available slides.
  - you may draw on slides, add slides, or choose to only use the slides provided
    - If you modify the provided slides, it is a good idea to state this at the beginning of the presentation.

The 8 Questions

1. Language Design and Control Structures
2. Structure of the compiler
3. Lexical analysis
4. Parsing
5. Semantic Analysis
6. Run-time organization
7. Heap allocation and Garbage Collection
8. Code Generation

And how did it go last year?



Important

- At the end of the course you should …
- Know
  – Which theories and techniques exist
  – Which tools exist
- Be able to choose “the right ones”
  – Objective criteria
  – Subjective criteria
- Be able to argue and justify your choices!

The <u>Most</u> Important Open Problem in Computing

<u>Increasing Programmer Productivity</u>

– Write programs correctly
– Write programs quickly
– Write programs easily

- Why?
  – Decreases support cost
  – Decreases development cost
  – Decreases time to market
  – Increases satisfaction

Why Programming Languages?

3 ways of increasing programmer productivity:

1. Process (software engineering)
   - Controlling programmers
2. Tools (verification, static analysis, program generation)
   - Important, but generally of narrow applicability
3. Language design --- the center of the universe!
   - Core abstractions, mechanisms, services, guarantees
   - Affect how programmers approach a task (C *vs*. SML)
   - Multi-paradigm integration

New Programming Language! Why Should I Care?

- The problem is not designing a new language
  - It's easy! Thousands of languages have been developed

- The problem is how to get wide adoption of the new language
  - It's hard! Challenges include
    - Competition
    - Usefulness
    - Interoperability
    - Fear

"It's a good idea, but it's a new idea; therefore, I fear it and must reject it."
--- Homer Simpson

- The financial rewards are low, but …

Famous Danish Computer Scientists

- Peter Nauer
  - BNF and Algol
- Per Brinck Hansen
  - Monitors and Concurrent Pascal
- Dines Bjørner
  - VDM and ADA
- Bjarne Straustrup
  - C++
- Mads Tofte
  - SML
- Rasmus Lerdorf
  - PhP
- Anders Hejlsberg
  - Turbo Pascal and C#
- Lars Bak
  - Java HotSpot VM, V8 and DART

- Jacob Nielsen

INTERNETAVISEN
Jyllands-Posten

Offentliggjort 24. februar 2006 14:01 - opdateret 14:06 Tip en ven Print-version

Fornem IT-pris til dansker

Som den første dansker nogensinde tildeles Peter Naur, professor emeritus ved Københavns Universitet, ACM's Turing Award - også kaldet datalogiens svar på en Nobelpris.

Prisen er datalogiens højeste udmærkelse og uddeles en gang om året til personer, som har ydet et afgørende og varigt bidrag til området. Den ledsages af et beløb på 100.000 dollars, svarende til ca. 620.000 kroner, som er skænket af virksomheden Intel. Prisen vil blive overrakt ved en banket den 20. maj i San Francisco.

Ifølge ACM's priskomité har Peter Naur fået prisen for "grundlæggende bidrag til udformningen af programmeringssprog og definitionen af Algol 60, til udformningen af oversættere og til det kreative og praktiske arbejde med programmering".

/ritzau/

Tilbage til forsiden Til toppen af siden Tip en ven Print-version

Tre danskere blandt verdens 150 it-helte

Tre danske it-pionerer har fundet vej til listen over alle tiders største it-helte.

Af Torben R. Simonsen, 13. februar 2007 kl. 09:49

It-mediesyndikatet Sys-con har spurgt i sit internationale net af it-medier og ikke mindst deres læsere, hvilke it-personligheder der har haft størst betydning.

Listen med de 150 mest betydningsfulde it-helte er nu færdig og tre danskere har fundet vej til listen.

Sys-con har ikke rangeret heltene indbyrdes, men giver blot en alfabetisk liste.

De tre danskere på listen er:

Anders Hejlsberg, der i dag er ansat i Microsoft, men som tidligere har stået bag udviklingen af Turbo Pascal og programmeringssproget C#.

Rasmus Lerdorf er med på listen for sit arbejde med udvikling af scriptsproget PHP

Bjarne Stroustrup er med for udarbejdelse af det oprindelige design til og implementering af C++.

Hverken Janus Friis eller Niklas Zennström, der blandt andet står bag Kazaa og Skype, har fundet vej til listen.

Mindre overraskende er det, at man kan finde personer som Microsoft-stifter Bill Gates, Intel-stifter Gordon Moore (Moores Lov) og skaberen af det moderne internet Tim Berners-Lee.

Relaterede links

SENESTE NYT | SENESTE DEBAT

- 10:39 › **Version2 arrangerer format battle**
- 10:20 › **Mangel på folk sender løn-opgaver til Multidata**
- 10:13 › **Stormløb mod servere forsinker selvangivelser**
- 09:51 › **Danskerne er vilde med Microsoft**
- 09:07 › **Bankernes EDB Central outsourcer pc-drift**
- 08:38 › **Belgisk ODF-pioner på Christiansborg**
- 08:04 › **Derfor mangler ÆØÅ på din bon**
- 07:41 › **Microsoft tester rettelse til Windows-sårbarhed**
- 15:45 › **Softwarefejl lukker bank-it landet over**
- 14:37 › **Google på vej med PowerPoint-dræber**

// Flere nyheder // Tilmeld nyhedsbrev

SENESTE BLOG-INDLÆG

- **Poul-Henning Kamp**: Utroligt hvad 7Watt kan | 7
- **Peter Toft**: Firefox hitter i Europa - men ikke i stokkonservative Danmark :-(| 20
- **Peter Toft**: Bestil mÍlk pÈ Jensens Býfhus | 16
- **Poul-Henning Kamp**: IPSEC sutter | 13
- **Peter Toft**: Hvilke nyheds-sites bør Linux folk følge? | 21

EMNER *C#*
SEKTION *Udvikling*

Se kommentarer (11)

Udvikler Anders Hejlsberg vinder dansk it-hæder

Manden bag både Turbo Pascal, Delphi og C# bliver årets vinder af IT-Prisen 2014, der uddeles af foreningen IT-Branchen.

Af Jesper Kildebogaard Fredag, 14. marts 2014 - 9:45

Mens nogle kalder sig serie-iværksættere, må Anders Hejlsberg kunne kalde sig serie-udvikler af programmeringssprog. Gennem sin lange karriere har han nemlig stået bag det ene toneangivende sprog efter det andet.

Den indsats får han nu et stort skulderklap for af den danske it-branche. På årsmødet for foreningen IT-Branchen blev IT-Prisen 2014 tildelt Anders Hejlsberg, som siden 1996 har arbejdet for Microsoft i USA.

Her har han senest været leder af udviklingen af Typescript, som er Microsofts bud på en forbedring af Javascript - samtidig med at man bevarer fuld kompabilitet med Javascript. Det kan du læse mere om i Version2's interview med Anders Hejlsberg fra 2012, da Typescript blev lanceret.

Læs også: Anders Hejlsberg: Sådan styrer jeg C#-udviklingen

Mere berømt er hans arbejde med C# og .Net, der i dag bliver brugt af millioner af udviklere i Microsoft-miljøer. Han arbejdede også på J++ og Microsoft Foundation Classes.

Før jobbet hos Microsoft udviklede han Delphi og Turbo Pascal hos Borland, der var et stort navn tilbage i 1980'erne. Det var et job hos Borland, der i 1987 trak Anders Hejlsberg til USA, hvor han har boet siden.

»Ingen danskere har som Anders Hejlsberg haft indflydelse på den digitale udvikling i verden. Han har gennem tre årtier påvirket udviklingen af de programmeringssprog, som er grundlaget for vores moderne kommunikationssamfund,« udtaler adm. direktør i IT-Branchen Morten Bangsgaard i en pressemeddelelse.

IT-Prisen er 'den største kollegiale hæder', der bliver uddelt i den danske it-branche, skriver foreningen selv. Prisen er de seneste år gået til blandt andet Michael Seifert, direktør for Sitecore, Lars Frelle-Petersen, direktør for Digitaliseringsstyrelsen, og it-iværksætteren Thomas Madsen-Mygdal. It-mediet Computerworld og IT-Branchen står bag prisen.

Fancy joining this crowd?

- Look forward to the PP (Programming Paradigms) course
  - on SW7/DAT7/IT7
- Look forward to the Advanced Programming course
  - On SW8/IT8
- Specialize in Programming Technology
  - on DAT9/DAT10 or SW9/SW10 or IT9/IT10

- Research Programme in Programming Technology
    - Programmatic Program Construction
    - Real-time programming in Java (and C)
    - Big Data and Functional Programming
      - Popular Parallel Programming (P3)
      - Prescriptive Analytics
    - Energy Aware Programming

- ”The P-gang”:
    - Kurt Nørmark
    - Lone Leth
    - Bent Thomsen
    - Thomas Bøgholm

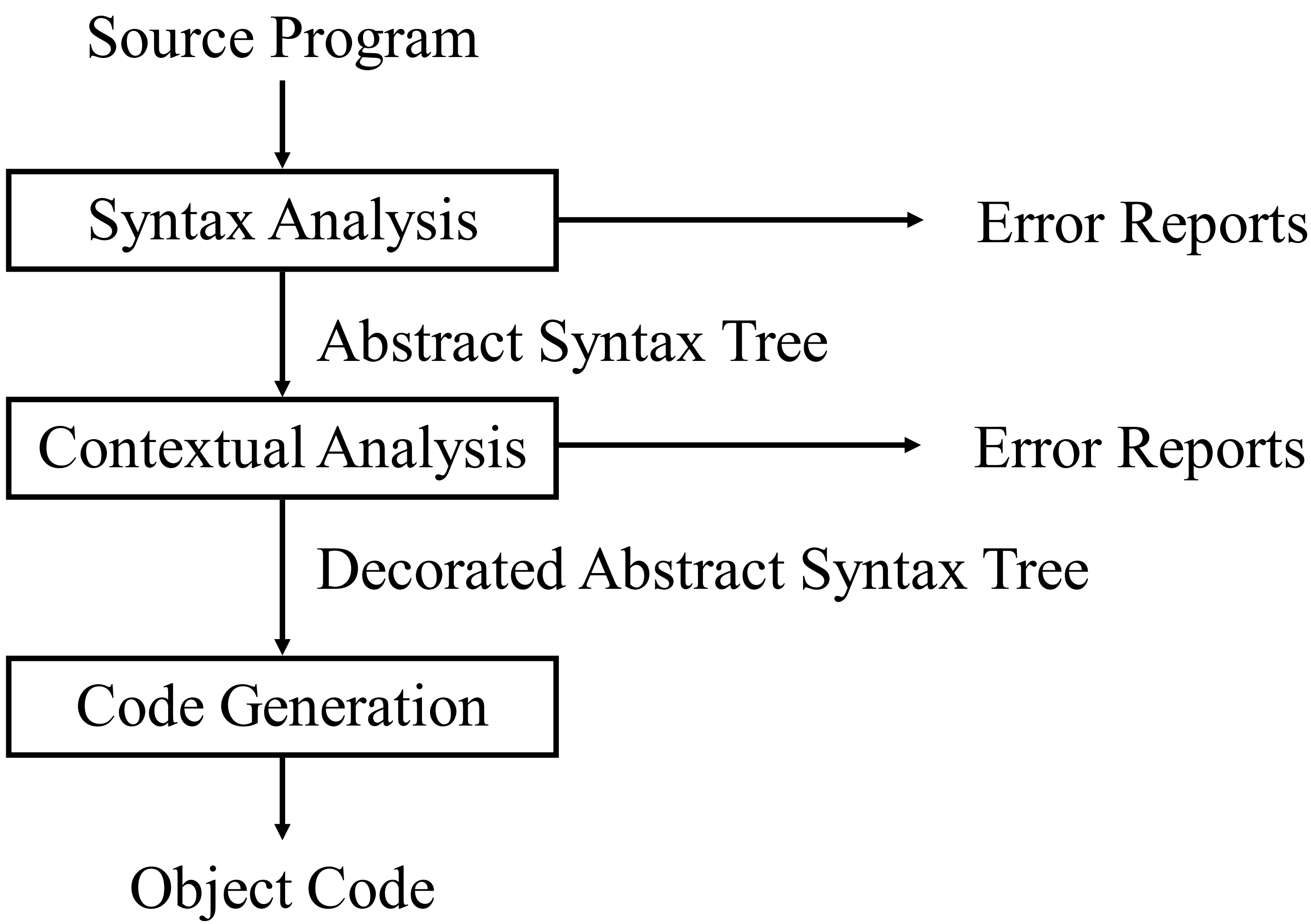
What I promised you at the start of the course

Ideas, principles and techniques to help you

- Design your own programming language or design your own extensions to an existing language
- Tools and techniques to implement a compiler or an interpreter
- Lots of knowledge about programming

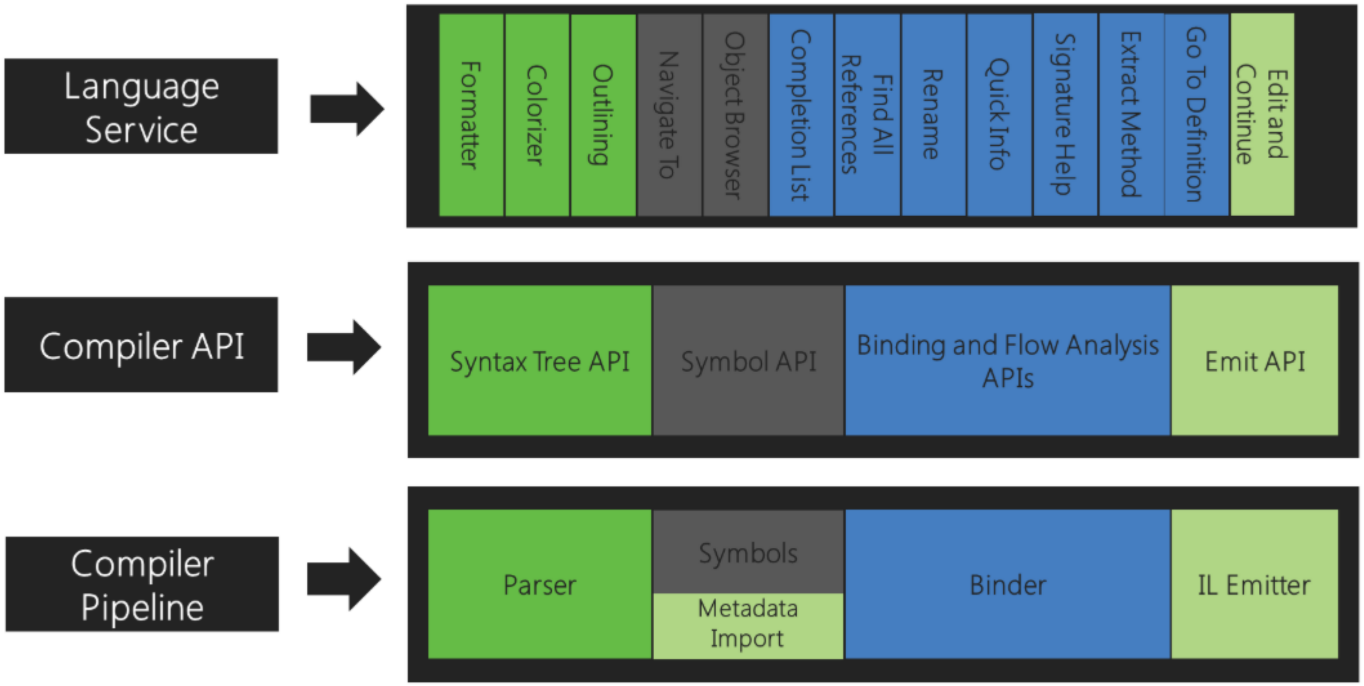
I hope you feel you got what I promised

The "Phases" of a Compiler



Is this picture still valid or is it how compilers were taught 30 years ago?

.NET Compiler Platform ("Roslyn") Overview



Corresponding to each of those phases, an object model is surfaced that allows access to the information at that phase:
The parsing phase is exposed as a syntax tree,
the declaration phase as a hierarchical symbol table,
the binding phase as a model that exposes the result of the compiler's semantic analysis
the emit phase as an API that produces IL byte codes.

Programming Language design

- Designing a new programming language or extending an existing programming language usually follows an iterative approach:

1. Create ideas for the programming language or extensions
2. Describe/define the programming language or extensions
3. Implement the programming language or extensions
4. Evaluate the programming language or extensions
5. If not satisfied, goto 1

Discount Method for Evaluating Programming Languages

1. Create tasks specific to the language being tested - tasks that the participants of the experiment should solve. Estimate the time needed for each task (max 1 hour)
2. Create a short sample sheet of code examples in the language being tested, which the participants can use as a guideline for solving the tasks.
3. Prepare setup (e.g. use of NotePad++ and recorder) and do a sample test with 1 person.
   - Adjust tasks if needed
4. Perform the test on each participant, i.e. make them solve the tasks defined in step 1. (Use approx. 5 test persons)
5. Each participant should be interviewed briefly after the test, where the language and the tasks can be discussed.
6. Analyze the resulting data to produce a list of problems
   - Cosmetic problems, Serious problems, Critical problems

Discount Method for Evaluating Programming Languages

- Method inspired by the Discount Usability Evaluation (DUE) method and Instant Data Analysis (IDA) method
- Reference:
  - Svetomir Kurtev, Tommy Aagaard Christensen, and Bent Thomsen.
  - Discount method for programming language evaluation.
  - In Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2016). ACM, New York, NY, USA, 1-8. DOI: https://doi.org/10.1145/3001878.3001879

Finally

Keep in mind, the compiler is the program from which all other programs arise. If your compiler is under par, all programs created by the compiler will also be under par. No matter the purpose or use -- your own enlightenment about compilers or commercial applications -- you want to be patient and do a good job with this program; in other words, don't try to throw this together on a weekend.

Asking a computer programmer to tell you how to write a compiler is like saying to Picasso, "Teach me to paint like you."

Sigh Well, Picasso tried.