Individual Exercises - Lecture 1

The following exercises you may prefer to do on your own, e.g. just after you have read the literature, and discuss the outcome with your group:

1. *Do Sebesta Review Questions 1 to 29 on page 54 and 55.*
    1. Why is it useful for a programmer to have the ability to learn new languages, even though he or she may have a good knowledge of a number of programming languages?

       Some languages are better suited for certain domains (Domain specific language)

       C would be more suited for operative systems, where python would be more suited for machine learning.

       Learning a language from a different paradigm (object oriented, functional, imperative) can give a deeper understanding of concepts that are available in most languages. For example learning a functional language like Haskell can improve your ability to utilize the functional programming features of Python.

    2. Why is it essential to choose an appropriate programming language for a specific software solution?

       The main advantage is increasing programmer productivity. As various languages are suited for different domains, a specific software solution should be written in a language, which suits that solution best.

       For example, writing safety critical applications for a bank, you might want to choose a language with a lot of compile time checks, to decrease risk of errors in the product.

    3. Which programming language for scientific applications was the first to be used successfully?

       Fortran

    4. Which is the first successful high-level programming language for business?

       COBOL

    5. In which programming language were most of the AI applications developed prior to 1990?

       LISP

    6. Which is the most popular markup language for Web development?

       HTML

7. Why is a list of programming language evaluation criteria for the development of software controversial?

It is difficult to get people to agree on the ranking of criteria.

8. How does the overall simplicity of a programming language affect its readability? A language with a large number of basic constructs is more difficult to read/learn than one with a smaller number.

9. Why is the VAX instruction design orthogonal? By operating on registers and memory.

VAX has only a small set of basic constructs, which then must work with each other in various ways to achieve more complex functionality.

10. Why does too much orthogonality cause problems?

Too many possible combinations can lead to an increase in complexity.

11. Explain how “writability” is used as a measure of how easily a language can be used to create programs.

For example a simple “Hello World!”-program in Java requires writing a class with public, static, void, main, i.e. learning both classes, access modifiers, static keywords, void return value and the main construct to simply print to the console. Printing to the console in java is also rather cumbersome requiring access to System.out.println(“Hello World!”);. Compare this to python, where one simply writes print(“Hello world!”).

12. Why is too much orthogonality a detriment to “writability”?

Errors in programs can go undetected when nearly any combination of primitives is legal. E.g if operators have more uses in a single context, one could think an expression has one outcome, however the outcome might be another. 13. Give an example of expressivity in a language.

A simple example is compound assignments:

i += 5 (as opposed to i = i + 5)

Another common example is for each loops (for example seen in Python, C#, Kotlin, etc.).:

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

    print(x)
```

This is arguably more expressive than having to maintain an index variable and access items directly through list indexing

14. What is type checking?

Assertions put in place to ensure that the types of the variables in an expression are compatible.

For example, a typechecker may warn you that 5 + “some text” is an invalid operation because the addition operator does not support mixing integers and strings. Some languages like Haskell include comprehensive type checking at compile time, which significantly decreases the risk of runtime errors. This, however, at the cost of writability.

15. Give an example of how the failure to type check, at either compile time or run time, can lead to countless program errors.

If the typechecker fails to typecheck an assignment of incompatible types e.g. int i = “hello world”, a type is treated as a different type, which leads to countless errors.

16. How is the total cost of a programming language calculated?

The total cost of a programming language is the cost of learning, writing, compiling, executing and maintaining the code.

17. What is portability of a language?

Portability is the ability to run programs on multiple different machines/architectures. Portability is most strongly influenced by the degree of standardization of the language. Some languages are not standardized at all, making programs in these languages very difficult to move from one implementation to another.

18. What is the use of the well-definedness criterion?

The completeness and precision of the language’s official defining document. 19. How does the execution of a machine code program on a von Neumann architecture computer occur?

The execution of a machine code program on a von Neumann architecture computer occurs in a process called the fetch-execute cycle.

20. What two programming language deficiencies were discovered as a result of the research in software development in the 1970s?

Incompleteness of type checking and inadequacy of control statements requiring the extensive use of gotos, which decreases readability and maintainability significantly. 21. What are the three fundamental features of an object-oriented programming language?

Encapsulation, inheritance and polymorphism.

22. What language was the first to support the three fundamental features of object-oriented programming?

Smalltalk

23. What is an example of two language design criteria that are in direct conflict with each other?

Two criteria that conflict are reliability and cost of execution. For example, the Java language definition demands that all references to array elements be checked to ensure that the index or indices are in their legal ranges. This step adds a great deal to the cost of execution of Java programs that contain large numbers of references to array elements. C does not require index range checking, so C programs execute faster than semantically equivalent Java programs, at the expense of providing fewer correctness guarantees. The designers of Java traded execution efficiency for reliability.

24. What are the three general methods of implementing a programming language?
    Compilation, pure-interpretation, hybrid-interpretation.
25. Which produces faster program execution, a compiler or a pure interpreter?
    Compiler.

26. What role does the symbol table play in a compiler?
    The symbol table serves as a database for the compilation process. The primary contents of the symbol table are the type and attribute information of each user-defined name in the program. This information is placed in the symbol table by the lexical and syntax analyzers and is used by the semantic analyzer and the code generator.
27. What does a linker do?
    The process of collecting system programs and linking them to user programs is called linking and loading, or sometimes just linking. It is accomplished by a systems program called a linker. In addition to systems programs, user programs must often be linked to previously compiled user programs that reside in libraries. So the linker not only links a given program to system programs, but also it may link it to other user programs.
28. Why is the von Neumann bottleneck important?
    The speed of the connection between a computer's memory and its processor often determines the speed of the computer, because instructions often can be executed faster than they can be moved to the processor for execution. This connection is called the von Neumann bottleneck
29. What are the advantages in implementing a language with a pure interpreter? It may be more platform independent/portable, as the programs can run anywhere that the interpreter can run (no need to write a compiler that targets several different architectures).

2. *Use Table 1.1 on page 31 in Sebesta's book to evaluate the C programming language*
   Simplicity: Few types, but several operators used for more tasks. i.e & and *

Orthogonality: Low. For example arrays cannot be returned from a function, but can be passed as arguments.

Data types: Only primitives. Very few

Syntax design: Originally, C was considered high level in its syntax. Since then, however, higher level languages have been developed, and C is now often no longer considered high level.

Support for abstractions: Low. No classes or generics.

Expressivity: low. Sorting, Strings, collections etc. require extensive programming. Type checking: Strict type checking at compile time.

Exception handling: None, usually error codes in the form of ints are used Restricted aliasing: Yes, int, 'double' etc. cannot be used as Identifiers. *3. Use Table 1.1 on page 31 in Sebesta's book to evaluate the C# programming language*

Simplicity: C# has many constructs like Java, but then also constructs like structs and gotos. For this reason C# cannot be considered a simple language

Orthogonality: C# has relatively high orthogonality (The ability to combine primitive constructs of the language). For example 'x += y' is a shorthand for addition of the form x = x + y, but when used in the context of event publishers, it means adding a listener function 'y' to the event publisher 'x'.

Data types: Contains both primitive and non-primitive types like classes

Syntax design: Uses similar syntax to contemporary languages like C and Java in order to increase familiarity.

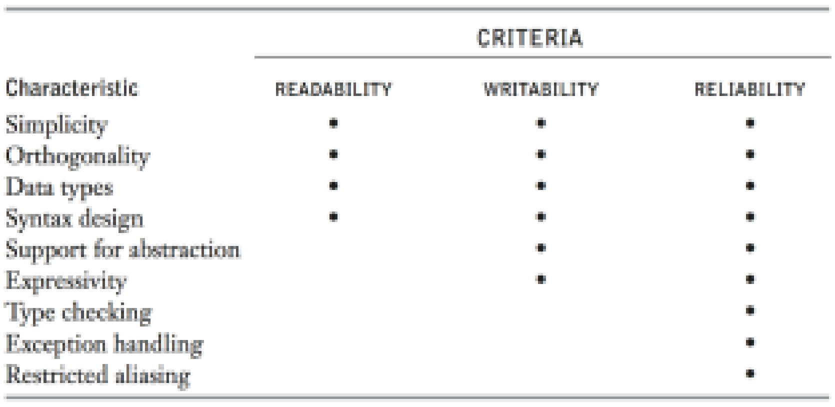
Support for abstraction: Yes, classes

Expressivity: Highly expressive (e.g compound statements, lists, strings) Type checking: Strict type checking at compile time and runtime.

Exception handling: Yes (try, catch, finally)

Restricted aliasing: Yes, int, double etc. cannot be used as identifiers.

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			•

4. Do Fisher et al. exercise 3 on page 26 (exercise 5 page 55 in GE)

C compilers are almost always written in C. This raises something of a “chicken and egg” problem—how was the first C compiler for a particular system created? If you need to create the first compiler for language X on system Y, one approach is to create a cross-compiler. A cross-compiler runs on system Z but generates code for system Y.

Explain how, starting with a compiler for language X that runs on systemZ, you might use cross-compilation to create a compiler for language X, written in X, that runs on system Y and generates code for system Y. What extra problems arise if system Y is “bare”—that is, has no operating system or compilers for any language? (Recall that Unix R is written in C and thus must be compiled before its facilities can be used.)

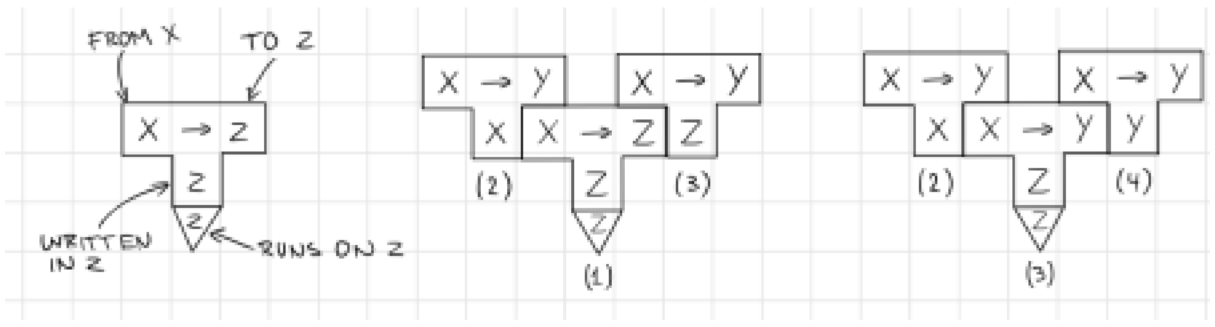
It can be difficult to wrap your head around this exercise. The key thing to learn here is that a compiler is just a program. The compiler itself is written in a programming language and the compiler code itself can be compiled. Furthermore the text of the exercise is a bit ambiguous as it does not make sense to talk about a compiler written in X running on system Y. We can talk about a compiler from X to Y written in X that we compile to a compiler from X to Y written in Y (or running on Y assuming that Y is machine language)

We are given a compiler(**1**) from X -> Z running on system Z (and more precisely written in Z).

To start off, create a compiler(**2**) from X -> Y written in X.

Next, compile the new compiler(**2**) using the given X -> Z compiler(**1**) that runs on Z. Now you have a compiler(**3**) from X -> Y that runs on system Z.

Finally you can compile (**2**) using (**3**), which gives you a compiler(**4**) from X->Y running on Y.



5. *Do Fisher et al. exercise 7 on page 28 (exercise 10 on page 57 in GE)*

Most C compilers (including the GCC compilers) allow a user to examine the machine instructions generated for a given source program. Run the following program through such a C compiler and examine the instructions generated for the for loop. Next, recompile the program, enabling optimization, and reexamine the instructions generated for the for loop. What improvements have been made? Assuming that the program spends all of its time in the for loop, estimate the speedup obtained. Write a suitable main C function that allocates and initializes a million-element array to pass to proc. Execute and time the unoptimized

and optimized versions of the program and evaluate the accuracy of your

```
int proc(int a[]) {
    int sum = 0, i;
    for (i=0; i < 1000000; i++)
        sum += a[i];
    return sum;
}
```

estimate.

O0 optimization

```
	.file	"proc.c"
	.text
	.globl	_proc
	.def	_proc;	.scl	2;	.type	32;	.endef
_proc:
LFB14:
	.cfi_startproc
	pushl	%ebp
	.cfi_def_cfa_offset 8
	.cfi_offset 5, -8
	movl	%esp, %ebp
	.cfi_def_cfa_register 5
	subl	$16, %esp
	movl	$0, -4(%ebp)
	movl	$0, -8(%ebp)
	jmp	L2
L3:
	movl	-8(%ebp), %eax
	leal	0(,%eax,4), %edx
	movl	8(%ebp), %eax
	addl	%edx, %eax
	movl	(%eax), %eax
	addl	%eax, -4(%ebp)
	addl	$1, -8(%ebp)
L2:
	cmpl	$999999, -8(%ebp)
	jle	L3
	movl	-4(%ebp), %eax
	leave
	.cfi_restore 5
	.cfi_def_cfa 4, 4
	ret
	.cfi_endproc
LFE14:
	.def	___main;	.scl	2;	.type	32;	.endef
	.globl	_main
	.def	_main;	.scl	2;	.type	32;	.endef
_main:
LFB15:
	.cfi_startproc
	pushl	%ebp
	.cfi_def_cfa_offset 8
	.cfi_offset 5, -8
	movl	%esp, %ebp
	.cfi_def_cfa_register 5
	andl	$-16, %esp
	movl	$4000032, %eax
	call	___chkstk_ms
	subl	%eax, %esp
	call	___main
	movl	$0, 4000028(%esp)
	jmp	L6
L7:
	movl	4000028(%esp), %eax
	movl	$2, 28(%esp,%eax,4)
	addl	$1, 4000028(%esp)
L6:
	cmpl	$999999, 4000028(%esp)
	jle	L7
	leal	28(%esp), %eax
	movl	%eax, (%esp)
	call	_proc
	movl	$0, %eax
	leave
	.cfi_restore 5
	.cfi_def_cfa 4, 4
	ret
	.cfi_endproc
LFE15:
	.ident	"GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"
```

O3 optimization

```
	.file	"proc.c"
	.text
	.p2align 4,,15
	.globl	_proc
	.def	_proc;	.scl	2;	.type	32;	.endef
_proc:
LFB24:
	.cfi_startproc
	movl	4(%esp), %edx
	xorl	%eax, %eax
	leal	4000000(%edx), %ecx
	.p2align 4,,10
L2:
	addl	(%edx), %eax
	addl	$4, %edx
	cmpl	%edx, %ecx
	jne	L2
	rep ret
	.cfi_endproc
LFE24:
	.def	___main;	.scl	2;	.type	32;	.endef
	.section	.text.startup,"x"
	.p2align 4,,15
	.globl	_main
	.def	_main;	.scl	2;	.type	32;	.endef
_main:
LFB25:
	.cfi_startproc
	pushl	%ebp
	.cfi_def_cfa_offset 8
	.cfi_offset 5, -8
	movl	%esp, %ebp
	.cfi_def_cfa_register 5
	andl	$-16, %esp
	call	___main
	xorl	%eax, %eax
	leave
	.cfi_restore 5
```

```
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
LFE25:
    .ident "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"
```

*Note that compilation time varies depending on the environment.

** O0 only optimizes compilation time and O3 has a range of optimization flags to be set, which includes executional optimization. Check the flags on https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

The compiler optimization -O3 speeds up the execution ~17%.

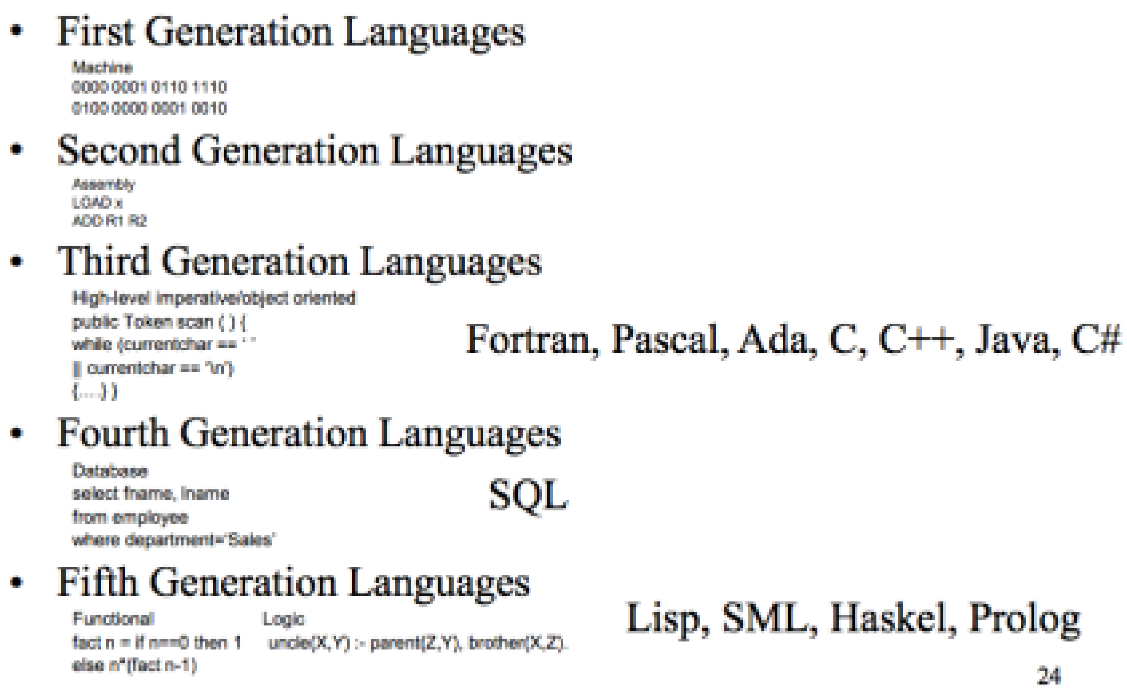
Group Exercises - Lecture 1

The following exercises are best done as group discussions:

1. *Review your individual list of programming languages and make a new list of all the computer languages that group members know.*

   Check your lists from lecture 0.

2. *Categorize the above languages according to generations (1st to 5th (or 6th))*



- 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

24

3. *Categorize the above languages according to paradigm (imperative, object oriented, declarative, special)*

1. Imperative: C, Fortran, Pascal, Ada (Also a little OOP)

2. Object Oriented: C++, Java, C#

3. Declarative: Haskel, SQL, Prolog, SML, LISP

4. *Create a new 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?*

Consider features like classes, typing, types, procedures, functions, lambda etc. Should it have a for loop? If so, how should it look?

Example of a feature:

ForForloop which executes like a nested for-loop

for(int i = 0, int j = 0; i < 100, j < 3; i++, j++) { /* code */ }

5. *Do Sebesta exercise 1, 2, and 3 page 55.*

1. *Do you believe our capacity for abstract thought is influenced by our language skills? Support your opinion.*

   Does an OOP language allow you to think in different abstractions than an imperative language?

2. *What are some features of specific programming languages you know whose rationales are a mystery to you?*

   Maybe you don't see the rationale behind having both structs and classes, when classes can just as easily be used for the same purpose as structs?

3. *What arguments can you make for the idea of a single language for all programming domains?*

   Having a single language for all domains would reduce fragmentation, but at the cost of excessive bloat. The language would be very big and complex. Many programming languages have features from multiple domains (mainly OOP and functional programming features).

6. Do Sebesta exercise 4, 7, 10 and 17 page 56.

4. *What arguments can you make against the idea of a single language for all programming domains?*

Languages can be optimized for a specific domain, both in terms of performance, but also writability, readability, simplicity etc.

7. *Java uses a right brace to mark the end of all compound statements. What are the arguments for and against this design?*

Pros: Easier to see where a statement ends and multiple statements can be on a line. Cons: More code to write for the programmer without any added functionality.

10. *What are the arguments for writing efficient programs even though hardware is relatively inexpensive?*

Lower energy consumption and computation is never free. Additionally, it is sometimes difficult to assess if the hardware in the environment is indeed inexpensive and available with high performance.

17. *Some programming languages—for example, Pascal—have used the semicolon to separate statements, while Java uses it to terminate statements. Which of these, in your opinion, is most natural and least likely to result in syntax errors? Support your answer.*

To explain the difference further: in pascal, the last statement before an END does not require a tailing semicolon(because it is not followed by further statements), but all previous ones do. In java every statement must be terminated by a semicolon.

7. Do Sebesta exercise 18 page 56.

18. Many contemporary languages allow two kinds of comments: one in which delimiters are used on both ends (multiple-line comments), and one in which a delimiter marks only the beginning of the comment (one-line comments). Discuss the advantages and disadvantages of each of these with respect to our criteria.

One-line: less typing for single line comments but more work to comment out multiple lines

Multiple-line: less typing to comment out multiple lines but more typing for one. Multiline comments can also serve to increase writability, since it allows to quickly comment out sections of code for debugging purposes.

8. Do Fisher et al. Exercise 5 and 6 on page 27-28 (exercise 7 and 14 on page 56 resp. 58 in GE)

5. To allow the creation of camera-ready documents, languages like TeX and LaTeX have been created. These languages can be thought of as varieties of programming languages whose output controls a printer or display. Source language commands control details like spacing, font choice, point size, and special symbols. Using the syntax-directed compiler structure of Figure 1.4, suggest the kind of processing that might occur in each compiler phase if TeX or LaTeX input was being translated.

An alternative to "programming" documents is to use a sophisticated editor such as that provided in Microsoft Word or Adobe FrameMaker to interactively enter and edit the document. (Editing operations allow the choice of fonts, selection of point size, inclusion of special symbols, and so on.) This approach to document preparation is called **WYSI WYG**—what you see is what you get—because the exact form of the document is always visible.

What are the relative advantages and disadvantages of the two ap- proaches? Do analogues exist for ordinary programming languages?

WYSIWYG editors provide an easy interface that most people can use, while LaTeX makes it easier to automate creation of documents so they can be made from templates (Class files).

6. Although compilers are designed to translate a particular language, they often allow calls to subprograms that are coded in some other language (typically, Fortran, C, or assembler). Why are such “foreign calls” allowed? In what ways do they complicate compilation?

FFI allows a programming language to interact with another language, which makes it possible to use foreign libraries and services. This can be useful if the “host” programming language is too slow (like Python) or if it does not provide this functionality at all. Compilation is complicated, since now the compiler has to interleave the output of the two languages to create the result.

9. *Do Fisher et al. Exercise 8 on page 28 (exercise 11 on page 57 in GE) 8. C is sometimes called the* ***universal assembly language*** *in light of its ability to be very efficiently implemented on a wide variety of computer architectures. In light of this characterization, some compiler writers have chosen to generate C code as their output instead of a particular machine language. What are the advantages to this approach to compilation? Are there any disadvantages?*

Advantages: Easy portability. C can be compiled to fast assembly code with existing C compilers. Now the compiler writes do not have to replicate all of the optimizations already built into the C compiler.

Disadvantages: It might be difficult to translate some high-level abstractions to C. You lose low level control, and you might not be able to fully optimize the performance of the new features that are in your language.

Individual Exercises - Lecture 2

1. Do Sebesta Review Questions 49 on page 128 and 50 on page 129 *49. How does the typing system of PHP and JavaScript differ from that of Java? JavaScipt and PHP use dynamic type binding, while Java uses static type binding. Dynamic type binding is more flexible, but disallows type checking at compile time.*

   50. What array structure is included in C# but not in C, C++, or Java? C# includes both jagged arrays (also in C, C++, C# and Java) and rectangular arrays. Rectangular arrays have a rectangular shape and entries are accessed using multiple indexes, i.e. [x, y], as opposed to [x][y] for jagged arrays.

2. Do Sebesta Programming Exercises 1,2 and 3 on page 131
   1. To understand the value of records in a programming language, write a small program in a C-based language that uses an array of structs that store student information, including name, age, GPA as a float, and grade level as a string (e.g., "freshmen," etc.). Also, write the same program in the same language without using structs.
   Without structs, you would use an extra dimension in the array for the properties. See an example of this in the code snippet (C#) below.

```
class Program
{
   static void Main(string[] args)
   {
       /*
        * Struct version
        */
       // Easy initialization
       Student s1 = new Student("Walter", 50, 4.0f, "Senior");
       Student s2 = new Student("Jesse", 25, 1.0f, "Freshmen");

       Student[] students = new Student[2];
       students[0] = s1;
       students[1] = s2;

       for (var i = 0; i < students.Length; i++)
       {
           // Easy and readable field access
           Console.WriteLine(students[i].Name);
       }
       /*
        * Array representation
```

```
             */
            object[,] arrStudents = new object[2, 4];
            arrStudents[0, 0] = "Walter";
            arrStudents[0, 1] = 50;
            arrStudents[0, 2] = 4.0f;
            arrStudents[0, 3] = "Senior";
            arrStudents[1, 0] = "Jesse";
            arrStudents[1, 1] = 25;
            arrStudents[1, 2] = 1.0f;
            arrStudents[1, 3] = "Freshmen";

            for (var i = 0; i < arrStudents.GetLength(0); i++)
            {
                // Difficult to read field access
                Console.WriteLine(arrStudents[i,0]);
            }

        }
    }

    public struct Student
    {
        public Student(string name, int age, float gpa, string gradeLevel)
        {
            Name = name;
            Age = age;
            GPA = gpa;
            GradeLevel = gradeLevel;
        }

        public string Name;
        public int Age;
        public float GPA;
        public string GradeLevel;

    }
```

2. To understand the value of recursion in a programming language, write a program that implements quicksort, first using recursion and then without recursion. As you can see below, the recursive version of QuickSort is significantly more readable. You can see the implementations in several different languages her: https://www.geeksforgeeks.org/quick-sort/

```
// Java program for implementation of recursive QuickSort
```

```java
import java.util.*;

class QuickSort {
    /* This function takes last element as pivot,
    places the pivot element at its correct
    position in sorted array, and places all
    smaller (smaller than pivot) to left of
    pivot and all greater elements to right
    of pivot */
    static int partition(int arr[], int low, int high) {
        int pivot = arr[high];
        int i = (low - 1); // index of smaller element
        for (int j = low; j <= high - 1; j++) {
            // If current element is smaller than or
            // equal to pivot
            if (arr[j] <= pivot) {
                i++;

                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }
    /* The main function that implements QuickSort()
    arr[] --> Array to be sorted,
    low --> Starting index,
    high --> Ending index */
    static void qSort(int arr[], int low, int high) {
        if (low < high) {
            /* pi is partitioning index, arr[pi] is
            now at right place */
            int pi = partition(arr, low, high);

            // Recursively sort elements before
            // partition and after partition
            qSort(arr, low, pi - 1);
            qSort(arr, pi + 1, high);
        }
    }

    // Driver code
    public static void main(String args[])
    {
        int n = 5;
        int arr[] = { 4, 2, 6, 9, 2 };

        qSort(arr, 0, n - 1);

        for (int i = 0; i < n; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

Iterative version can be found on the next page.

```
// Java program for implementation of iterative QuickSort
import java.util.*;

class QuickSort {
    /* This function takes last element as pivot,
    places the pivot element at its correct
    position in sorted array, and places all
    smaller (smaller than pivot) to left of
    pivot and all greater elements to right
    of pivot */
    static int partition(int arr[], int low, int high) {
        int pivot = arr[high];

        // index of smaller element
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++) {
            // If current element is smaller than or
            // equal to pivot
            if (arr[j] <= pivot) {
                i++;

                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }

    /* A[] --> Array to be sorted,
   l --> Starting index,
   h --> Ending index */
    static void quickSortIterative(int arr[], int l, int h) {
        // Create an auxiliary stack
        int[] stack = new int[h - l + 1];

        // initialize top of stack
        int top = -1;
        // push initial values of l and h to stack
        stack[++top] = l;
        stack[++top] = h;

        // Keep popping from stack while is not empty
        while (top >= 0) {
            // Pop h and l
            h = stack[top--];
            l = stack[top--];

            // Set pivot element at its correct position
            // in sorted array
            int p = partition(arr, l, h);

            // If there are elements on left side of pivot,
            // then push left side to stack
            if (p - 1 > l) {
                stack[++top] = l;
                stack[++top] = p - 1;
            }
            // If there are elements on right side of pivot,
            // then push right side to stack
```

```
                if (p + 1 < h) {
                    stack[++top] = p + 1;
                    stack[++top] = h;
                }
            }
        }
        // Driver code
        public static void main(String args[])
        {
            int arr[] = { 4, 3, 5, 2, 1, 3, 2, 3 };
            int n = 8;

            // Function calling
            quickSortIterative(arr, 0, n - 1);

            for (int i = 0; i < n; i++) {
                System.out.print(arr[i] + " ");
            }
        }
    }
```

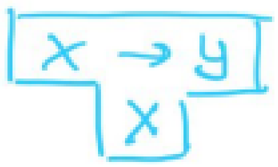
3. To understand the value of counting loops, write a program that implements matrix multiplication using counting loop constructs. Then write the same program using only logical loops—for example, while loops.
While splits definition, check, and increment into three lines, a for loop uses one. Counting loops increases readability of the code.

```
// Matrix multiplication not using counting loops
public static double[][] multiplyMatricesWhileLoops(double[][] M1,
double[][] M2){
    double[][] result = new double[M1.length][M2[0].length];

    int row = 0;
    while(row < result.length){
        int col = 0;
        while(col < result[row].length){
            double cell = 0;
            int i = 0;
            while(i < M2.length){
                cell += M1[row][i] * M2[i][col];
                i++;
            }
            result[row][col] = cell;
            col++;
        }
```

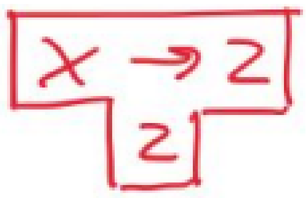
```
            row++;
        }

        return result;
    }


// Matrix multiplication using counting loops
public static double[][] multiplyMatricesCountingLoop(double[][] M1,
double[][] M2) {
    double[][] result = new double[M1.length][M2[0].length];

    for (int row = 0; row < result.length; row++) {
        for (int col = 0; col < result[row].length; col++) {
            double cell = 0;
            for (int i = 0; i < M2.length; i++) {
                cell += M1[row][i] * M2[i][col];
            }
            result[row][col] = cell;
        }
    }
    return result;
}
```

3. (Optional) Exercise 4 from Lecture 1: Do Fisher et al. exercise 3 on page 26 (exercise 5 page 55 in GE) poses some interesting questions about bootstrapping and viewing compilers as pieces of software in general. For this purpose the notion of T-diagrams, also sometimes called tombstone diagrams, has been invented. Read the above references and use T-diagrams to solve the exercise. Draw your T-diagrams using TDiagram Editor for the Flaci toolkit.

   Since I want a compiler from X to Y running on system Y we start by creating this compiler.

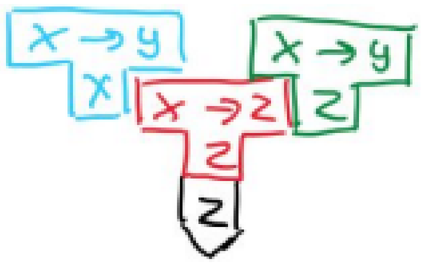


   I assume that I already have a compiler from X to Z that runs on system

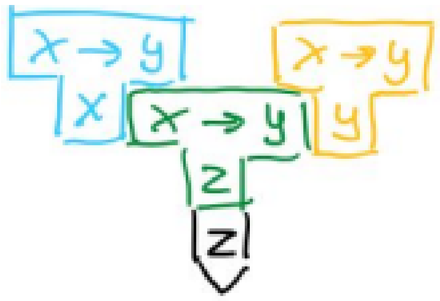


   Z.

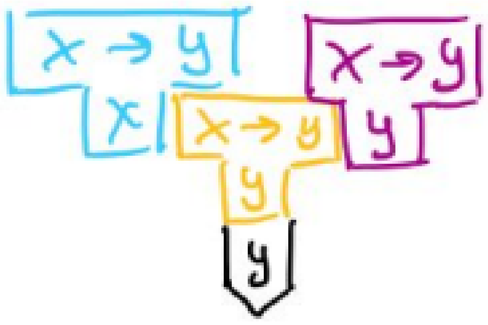
I can use these two pieces to get a compiler from X to Y running on system Z. This is my cross compiler.



In combination with the new cross compiler and system Z i can now create a compiler from X to Y running on Y.



I then transfer the compiler from X to Y to system Y.
I now have a compiler for language X that runs on system Y and generates code for system Y.



4. Do Fisher et al. exercise 4 on page 27

4. Cross-compilation assumes that a compiler for language X exists on some machine. When the first compiler for a new language is created, this assumption does not hold. In this situation, a **bootstrapping** approach can be taken. First, a subset of language X is chosen that is sufficient to implement a simple compiler. Next, a simple compiler for the X subset is written in any available language. This compiler must be correct, but it should not be any more elaborate than is necessary, since it will soon be discarded. Next, the subset compiler for X is rewritten in the X subset and then compiled using the subset compiler previously created. Finally, the X subset, and its compiler, can be enhanced until a complete compiler for

X, written in X, is available.

Assume you are bootstrapping C++ or Java (or some comparable language). Outline a suitable subset language. What language features must be in the language? What other features are desirable?

Required: I/O (File / Memory), Branches (control structure), Iteration, Dynamic Memory

Deseriable: Subroutines, Strings, Boolean

5. Familiarize yourself with the companion web site to Crafting a Compiler http://www.cs.wustl.edu/~cytron/cacweb/e.g. read the FAQ and follow the instructions Installing Eclipse.

For this a virtual machine will be provided with the necessary tools pre installed.

Group Exercises - Lecture 2

1. Do Sebesta exercise 6, 7, 14, 16, 17, 18, 21, 22, 24 on page 129-130 *6. Make an educated guess as to the most common syntax error in Lisp programs.* Too many, too few, or incorrectly positioned parentheses.

7. Lisp began as a pure functional language but gradually acquired more and more imperative features. Why?
To increase performance, but some problems are also hard to express using a pure functional programming language.

14. What are the arguments both for and against the idea of a typeless language?
For: More freedom to create abstractions and less typing. If everything is well thought out errors should not occur.
Against: Increased amounts of bugs that are difficult to find, since in reality most programs are not perfectly written.

16. What is your opinion of the argument that languages that are too complex are too dangerous to use, and we should therefore keep all languages small and simple?
Complexity makes it easier to introduce subtle bugs without knowing. Users might only use parts of the language, while other users use different parts of the language. This might lead to each sub-group not understanding each other's language subset. Last, as languages die the programs written in it might not, this could introduce legacy code that is very hard for developers to debug and maintain.

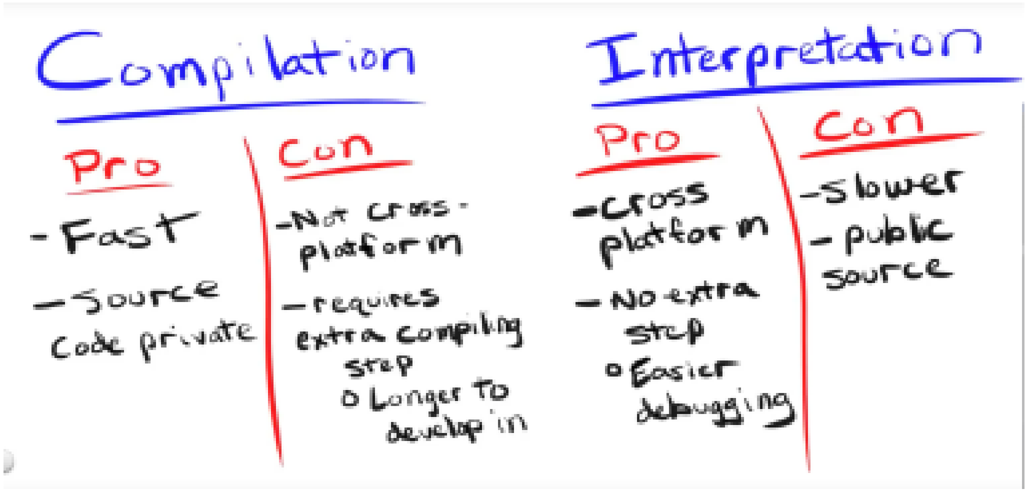
17. Do you think language design by committee is a good idea? Support your opinion.
A committee can help create a standard accepted by multiple parties (e.g. C++

committee has multiple companies), but also makes the process slower. However, if the members of the committee do not have shared goals, the resulting language might contain different features with the same functionality and be very complex.
Open-question: is open-source language design the same as design by committee (Python PEP, Rust RFC, etc)?

18. Languages continually evolve. What sort of restrictions do you think are appropriate for changes in programming languages? Compare your answers with the evolution of Fortran.
Backwards compatibility might be very important. Or at least a very good compiler, compiling the old code to working new code. It may, however, be difficult to do while keeping the semantics of the program.
Fortran is well known for having a high level of backwards compatibility.
21. In recent years data structures have evolved in scripting languages to replace traditional arrays. Explain the chronological sequence of these developments.

It all started as direct memory management with just a block of memory. Then we got static size arrays and linked lists. By now most programming languages contain some kind of arrays.

22. Explain two reasons why pure interpretation is an acceptable implementation method for several recent scripting languages.
One situation in which pure interpretation is acceptable for scripting languages is when the amount of computation is small, for which the processing time will be negligible. Another situation is when the amount of computation is relatively small and it is done in an interactive environment, where the processor is often idle because of the slow speed of human interactions.



Compilation

Pro	Con
- Fast - Source code private	- Not cross-platform - requires extra compiling step o Longer to develop in

Interpretation

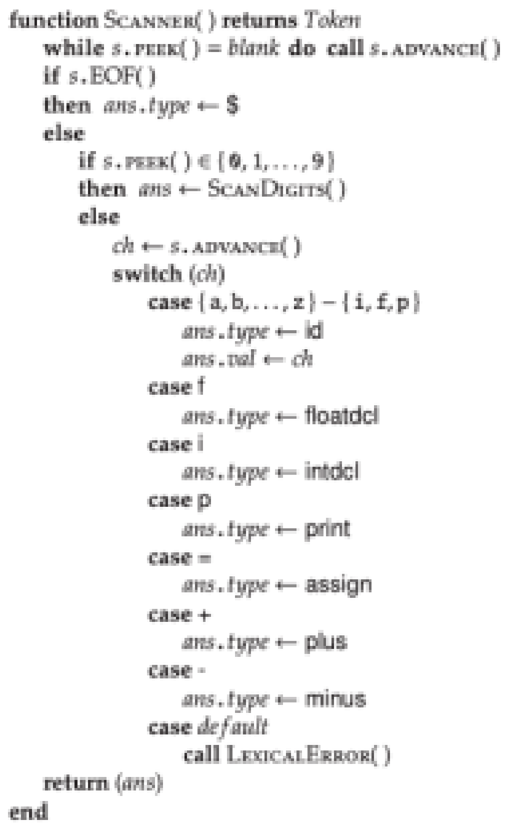
Pro	Con
- cross platform - No extra step o Easier debugging	- Slower - public source

24. Give a brief general description of a markup-programming hybrid language. A markup-programming hybrid language is a markup language in which some of the elements can specify programming actions, such as control flow and computation.
The concept of literate programming is close, here a markdown language embeds blocks of code written in one or more programming languages (R, Emacs Org-mode,

Jupyter Notebook).

Individual Exercises - Lecture 3

1. Do Fisher et al. exercise 3,4,6 page 54-55 (exercise 3,8,9 page 82-84 in GE)
   3. Extend the ac scanner (Figure 2.5) in the following ways:



```
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.

(a) floatdcl can be represented as either f or float, allowing a more Java-like syntax for declarations.
Alter the f case with the following line:
case f:
    if(s.peek == ‘l’) consume(“loat”))

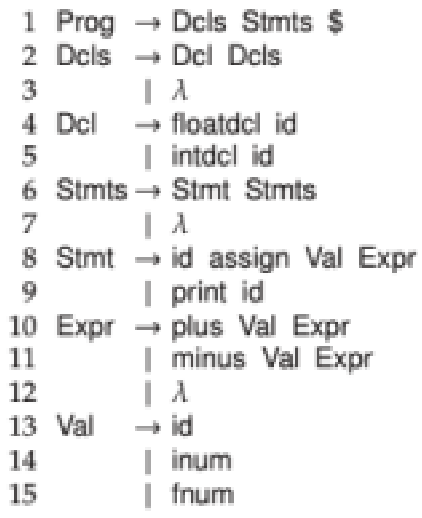
(Consume(‘loat’) automatically calls s.advance() and asserts that the following characters match the given string “loat”. Consume calls LexicalError() if s.advance() does not correspond to ‘l’ ‘o’ ‘a’ ‘t’)

(b) An intdcl can be represented as either i or int
Add to case i:
    if(s.peek == ‘i’) consume(“nt”))
(c) A num may be entered in exponential (scientific) form. That is, an ac num may be suffixed with an optionally signed exponent (1.0e10, 123e-22 or

0.31415926535e1).

```
function ScanDigits() returns token
    tok.val = " "
    while s.peek() ∈ {0, 1 .... 9} do
        tok.val = tok.val + s.advance()
    if s.peek() == "." or s.peek() == "e"
        tok.type = fnum

        if s.peek() == "."
             tok.val = tok.val + s.advance()
             while s.peek() ∈ {0, 1 .... 9} do
                 tok.val = tok.val + s.advance()

        if s.peek() == "e"
             tok.val = tok.val + s.advance()
             if s.peek() == "-"
                 tok.val = tok.val + s.advance()
            while s.peek() ∈ {0, 1 .... 9} do
                 tok.val = tok.val + s.advance()
    else
        tok.type = inum
    return (tok)
end
```

4. *Write a recursive descent parser for each of the non-terminals in figure*



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

2.1 Figure 2.1: Context-free grammar for ac.

```
Procedure Prog()
    call Dcls()
```

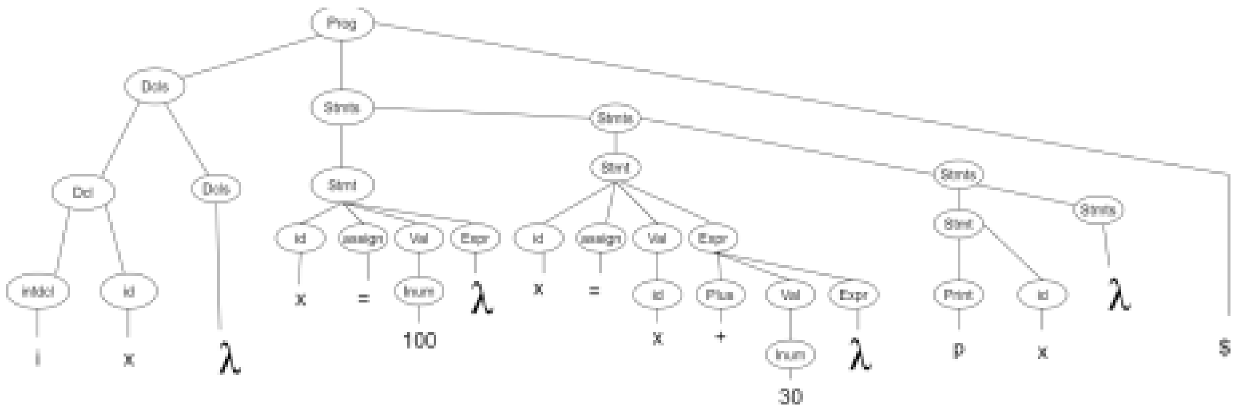
```
      call Stmts()
      call MATCH(ts, $)
end


Procedure Dcls()
      if ts.peek() = floatdcl or ts.peek() = intdcl
      then
            call Dcl()
            call Dcls()
      else
            if ts.peek() = $ OR ts.peek() = id OR ts.peek() = print
            then
                  /* Do nothing*/
            else
                  call ERROR()
end


Procedure Dcl()
      if ts.peek() = floatdcl
      then
            call MATCH(ts, floatdcl)
            call MATCH(ts, id)
      else
            if ts.peek() = intdcl
            then
                  call MATCH(ts, intdcl)
                  call MATCH(ts, id)
            else
                  call ERROR()
end


Procedure Stmts()
      if ts.peek() = id or ts.peek() = print
      then
            call Stmt()
            call Stmts()
      else
            if ts.peek() = $
            then
                  /* Do nothing*/
            else
                call ERROR()
end


Procedure Stmt()
        if ts.peek() = id
        then
            call MATCH(ts, id)
            Call MATCH(ts, assign)
            Call Val()
            Call Expr()
        else
            if ts.peek() = print
            then
                call MATCH(ts, print)
                call MATCH(ts, id)
            else
                call ERROR()
end


Procedure Expr()
        if ts.peek() = plus
        then
            call MATCH(ts, plus)
            call Val()
            call Expr()
        else
            if ts.peek() = minus
            then
                call MATCH(ts, minus)
                call Val()
                call Expr()
end


Procedure Val()
        if ts.peek() = id
        then
            call MATCH(ts, id)
        else
            if ts.peek() = inum
            then
                call MATCH(ts, inum)
            else
                if ts.peek() = fnum
                then
                    call MATCH(ts, fnum)
                else
                    call ERROR()
end
```

6. Variables are considered uninitialized after they are declared in some programming languages. In ac a variable must be given a value in an assignment statement before it can be correctly used in an expression or print statement. Suggest how to extend ac's semantic analysis (Section 2.7) to detect variables that are used before they are properly initialized.

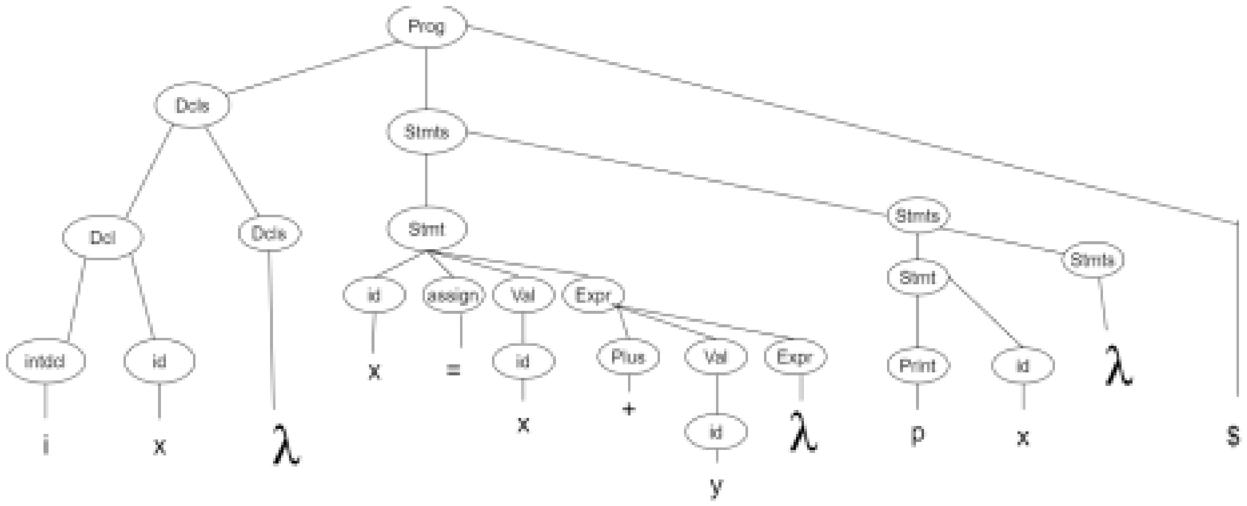
Extend the symbol table, such that it contains information whether a variable has been initialized or not. In this way it is possible to distinguish between declared and initialized.

2. *Based on the grammar in Figure 2.1 construct the parse tree for each of the following:*

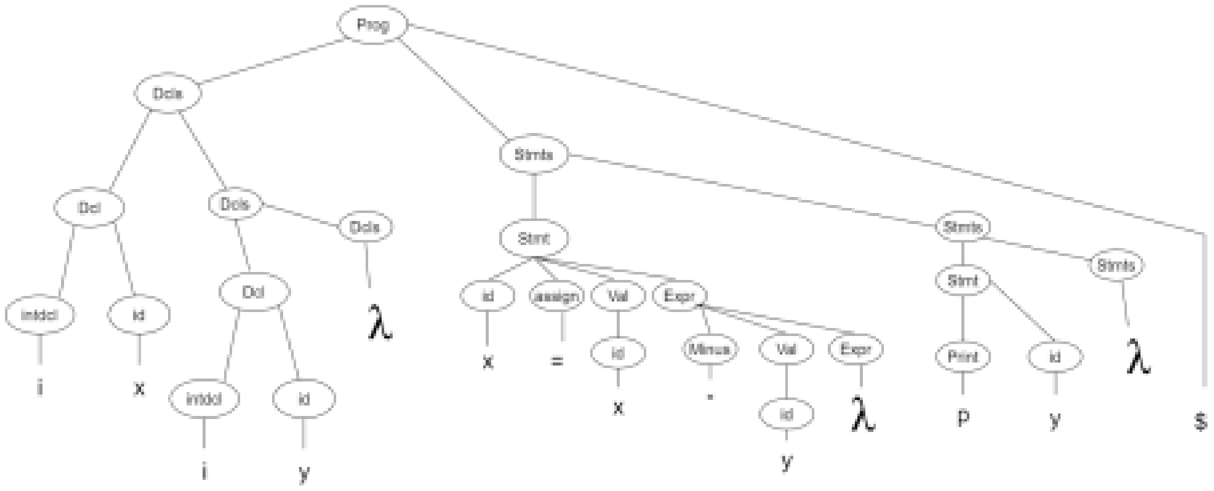
a) i x x = 100 x = x + 30 p x



b) i x x = x + y p x



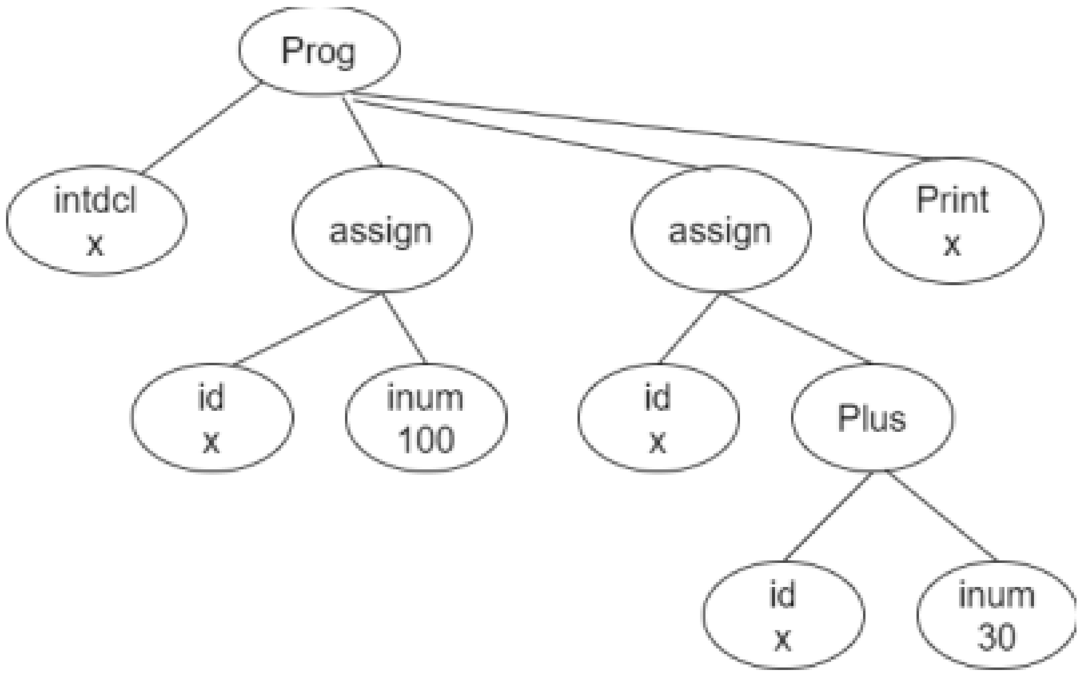
c) i x i y x = x - y p y



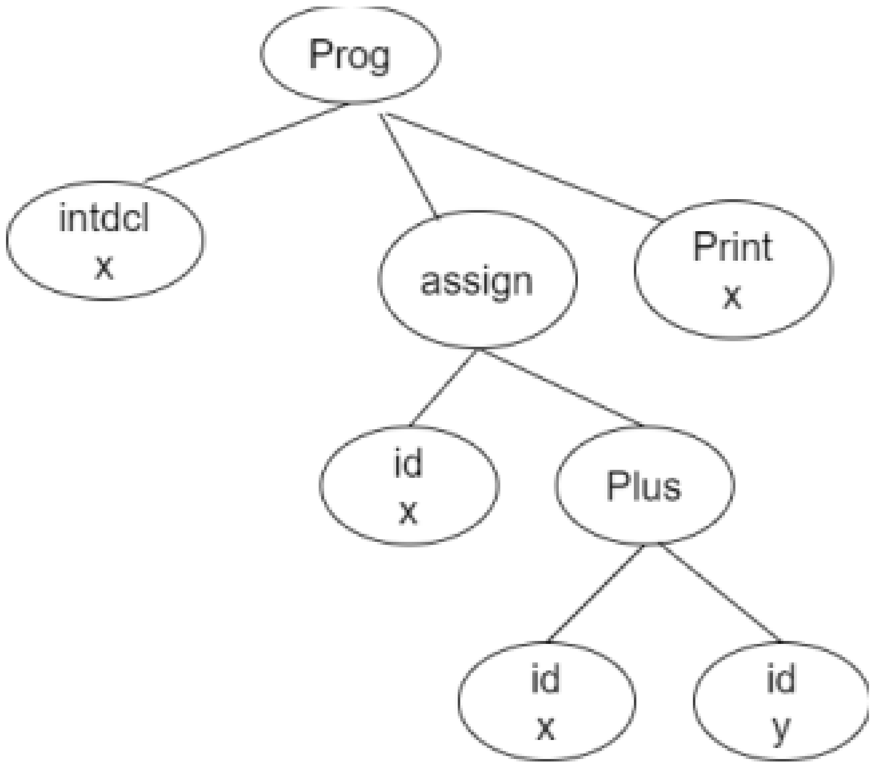
3. *For each of the programs in exercise 2:*

a) Construct the AST

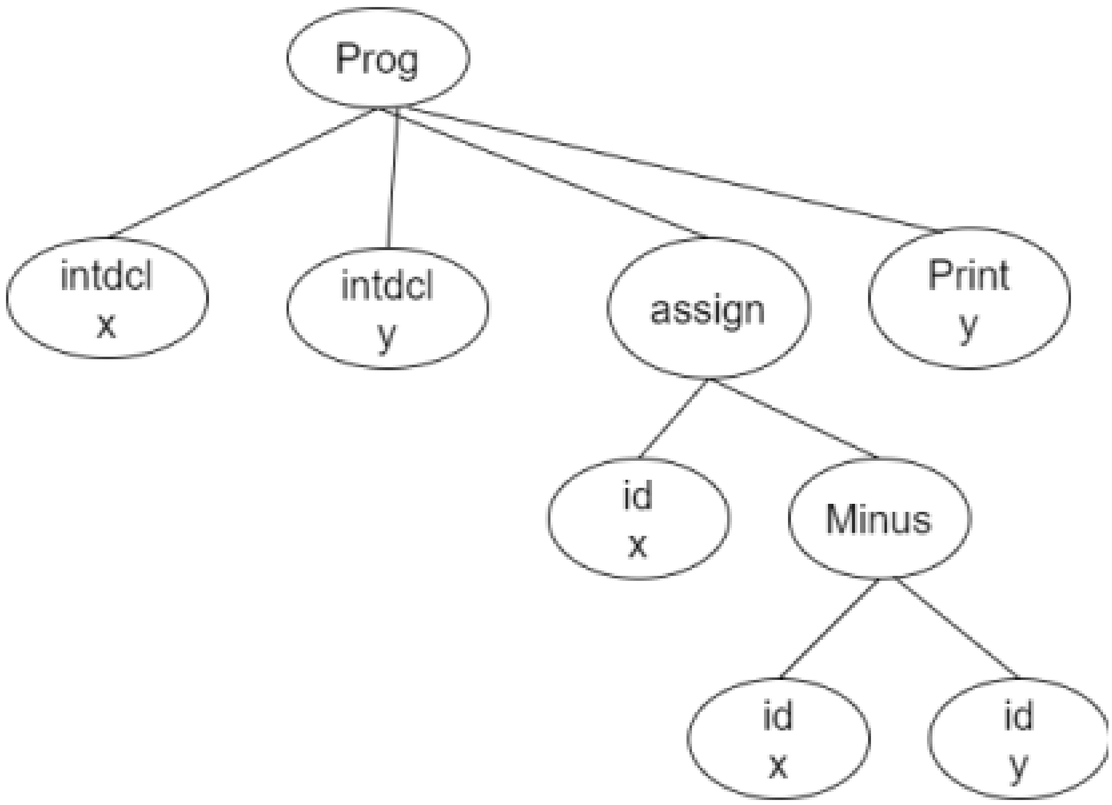
i x x = 100 x = x + 30 p x



i x x = x + y p x



i x i y x = x - y p y



b) According to the definition of ac, is the input semantically correct? If not, what changes to the ac language would render the program semantically correct For "i x x = x + y p x" y is not declared before it is used. The ac language requires an identifier to be declared prior to use.
For "i x i y x = x - y p y" y and x is not initialized before use. The ac language requires an identifier to be initialized prior use.

c) With ac changes in place that make the program semantically correct, show the code that would be generated from these programs

i x x = 100 x = x + 30 p x

Code	*Source Comments*
100 *sx* *0 k*	*x = 100 Push 100 on stack Pop stack, storing the* *popped value in reg x* *Reset precision to integer*
lx *30* *+*	*x = x + 30 Push value of reg x on stack* *Push 30 on stack* *Pop 2 elements from*

sx	*stack and push the sum* *Store top stack value in x*
lx *p* *si*	*p x Push value of the x register* *Print the top-of-stack value* *Pop the stack by storing into the i register*

i x i y x = 30 y = 30 x = x + y p x

Code	Source Comments
30 Sy 0k	y = 30 Push 30 on stack *Pop stack, storing the popped value in reg y* *Reset precision to integer*
30 sx 0 k	x = 30 Push 30 to stack Pop and store top of stack in reg x Set precision to Integer

Lx	x = x +y Push value of reg x on stack
Ly	Push value of reg y on stack
+	Pop 2 elements from stack and push the sum
sx	Store top stack value in x
0k	Reset precision to integer
Lx	p x Push value of reg x
P	Print the top-of-stack value
si	Pop the stack by storing

into the i register

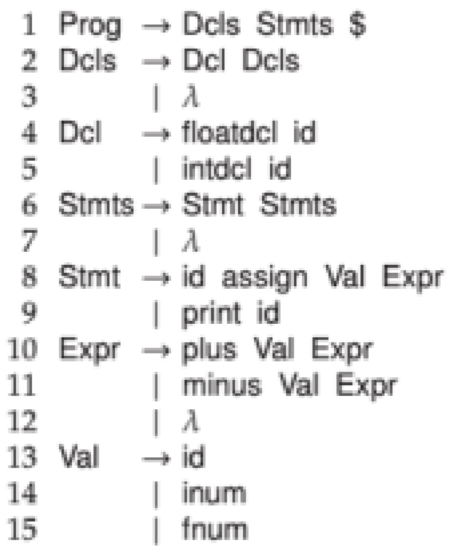
i x i y x = 30 y = 30 x = x - y p y

Code	Source Comments
30 sx 0 k	x = 30 Push 30 to stack Pop and store top of stack in reg x Set precision to Integer
30	y = 30 Push 30 to stack
sy 0 k	Pop and store top of stack to reg y Set precision to integer
lx ly - sx	x = x -y Push value of reg x to stack Push value of reg y to stack Pop two elements from stack and push difference Pop stack and store in reg x
ly p	p y Push value of reg y to the stack Pop and print

4. *(optional but recommended) Follow the studio associated with Crafting a Compiler Chapter 2: A Simple Compiler http://www.cs.wustl.edu/~cytron/cacweb/Chapters/2/studio.shtml You can find the source zip file in the General_Course_Material directory https://www.moodle.aau.dk/pluginfile.php/154451/mod_folder/content/0/Studio_Code.zip?forcedownload=1*
For this a virtual machine will be provided with the necessary tools pre installed.

Group Exercises - Lecture 3

The following exercises are best done as group discussions:



```
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.

1. Do Fisher et al. Exercise 1,2,8,9,10,12 page 54-55 (exercise 4,5,6,10,12,13 on page 82-84 in GE)

1. The CFG shown in Figure 2.1 defines the syntax of ac programs. Explain how this grammar enables you to answer the following questions.
(a) Can an ac program contain only declarations (and no statements)? Yes, Using Rule 7, the nonterminal Stmts can derive to the null string. (Fisher,page 33, figure 2.1)
(b) Can a print statement precede all assignment statements?
Yes, The rules for Stmts impose no constraints on the ordering of a sequence of instances of the Stmt.

2. Sometimes it is necessary to modify the syntax of a programming lan- guage. This is done by changing the CFG that the language uses. What changes would have to be made to ac's CFG (Figure 2.1) to implement the following changes? (a) All ac programs must contain at least one statement.

Prog -> Dcls Stmt Stmts $
(b) All integer declarations must precede all float declarations

Prog -> IntDcls FloatDcls Stmts $

FloatDcls-> floatdcl id FloatDcls

| λ

IntDcls -> intdcl id IntDcls

| λ

(c) The first statement in any ac program must be an assignment statement.

```
Prog -> Dcls Assignment Stmts $
      | Dcls $
Assignment -> id assign Val Expr
Stmt -> Assignment | print id
```

8. The grammar for ac shown in Figure 2.1 requires all declarations to precede all executable statements. In this exercise, the ac language is extended so that declarations and executable statements can be interspersed. However, an identifier cannot be mentioned in an executable statement until it has been declared.

(a) Modify the CFG in Figure 2.1 to accommodate this language extension.

Checking that a variable is declared before use cannot be enforced in the CFG. This is part of the semantic analysis and should be done using the symbol table. We can, however, change the CFG to allow for Dcls and Stmt to be interspersed.

```
Prog -> Lines $
Lines -> Dcl Lines
        | Stmt Lines
        | λ
Dcl -> floatdcl id
        | intdcl id
Stmt -> id assign Val Expr
        | print id
Expr -> plus Val Expr
        | minus Val Expr
        | λ
Val -> id
        | inum
        | fnum
```

(b) Discuss any revisions you would consider in the AST design for ac.

Link ‘id’ nodes to their corresponding declaration node

(c) Discuss how semantic analysis is affected by the changes you envision for the CFG and the AST.

Since Dcls and Stmts can be interspersed i now not only have to check if the variable is declared, but also when it was declared. This can for example be done using a symbol table with a column for when a given variable was declared or assigned.

9. The abstract tree design for ac uses a single node to represent a print operation

(see Figure 2.9). Consider an alternative design in which the print operation always has a single id child that represents the variable to be printed. What are the design and implementation issues associated with the two approaches? The existing design is based on the definition of a print command in ac, which only allows specification of a single name for which the corresponding value is to be printed. The implementation that follows is simple because all of the information regarding a print statement resides in a single **abstract syntax tree** (AST) node. The alternate approach suggested in this exercise would more easily accommodate a language change that allowed a list of names or constant values as part of a single print statement. The implementation would be more complicated because traversal of a list of items to print would be required, even if there was only a single item on that list.

10**.** *The code in Figure 2.10 examines an AST node to determine its effect on the symbol table. Explain why the order in which nodes are visited does or does not matter with regard to symbol-table construction.*

Since the result of a call to LookupSymbol does not depend on the order in which symbols are entered into the symbol table, the order in which declaration nodes are processed does affect the lookup process.

If a symbol is declared twice, the error message in EnterSymbol will be triggered regardless of the order in which the declarations are processed, although the error message may be attached to the first declaration of a symbol with more than one declaration if declarations are processed out of order.

12. The last fragment of code generated in Figure 2.15 pops the dc stack and stores the resulting value in register i.

(a) Why was register i chosen to receive the result?

Register **i** is used because **i** is a keyword, which can never be used for a variable

(b) Which other registers could have been chosen without causing any problems for code that might be generated subsequently?

Register **f** or **p** could be chosen, as they are keywords

2. (Optional) Do the group exercise part of the studio associated with Crafting a

Compiler Chapter 2: A Simple Compiler

For this a virtual machine will be provided with the necessary tools pre installed.

3. (Optional) Extend the ac grammar to allow parentheses in expressions i.e. (4-3) + (2-1). How would this affect the ac compiler?

```
Prog → Dcls Stmts $

Dcls → Dcl Dcls

       | λ

Dcl → floatdcl id

       | intdcl id

Stmts → Stmt Stmts

       | λ

Stmt → id assign Val Expr
       | id assign ‘(‘ Val Expr ‘)’ Expr

       | print id

Expr → plus Val Expr

       | plus ‘(‘ Val Expr ‘)’ Expr

       | minus Val Expr
       | minus ‘(‘ Val Expr ‘)’ Expr

       | λ

Val → id

       | inum
       | fnum
```

The compiler would apply precedence according to the parenthesis

Individual Exercises-Lecture 4

1. Browse the language specifications listed above
    - ○ Java: The Java Language Specification, Third Edition - TOC
      http://docs.oracle.com/javase/specs/
    - ○ C#:
      http://www.ecma-international.org/publications/standards/Ecma-334.htm
    - ○ JavaScript (ECMAScript):
      http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pd
    - f ○ Standard ML:
      http://www.lfcs.inf.ed.ac.uk/reports/88/ECS-LFCS-88-62/ECS-LFCS-88-62.pdf

2. (Optional) Do Sebesta Review questions 1, 2, 3, 4, 6, 7, 25 on pages 180-181

1. Define lexeme and token.

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

2. How are programming languages formally defined?

Programming languages are formally defined using a context free grammar (CFG) in combination with structural and operational semantics.

3. In which form is the programming language syntax commonly described? The syntax of programming languages is described using a context free grammar either in Backus-Naur-Form (BNF) or some variant of Extended-Backus-Naur-Form (EBNF).

4. What is a metalanguage? A language used to describe another language, for example Backus-Naur-Form.

5. What is a derivation in the context of grammar? A derivation is the process of replacing a non-terminal token with one of that nonterminal's definitions.

6. What is an ambiguous grammar? A grammar is ambiguous, if there exists an input from which two (or more) different parse trees can be derived.

7. What is a left-recursive grammar? Non-terminals appear recursively on the left side (ex: Expr -> Expr + id). This makes LL(1) parsers impossible.

25. What is the problem with using a software pure interpreter for operational semantics? The detailed characteristics of the particular computer would make actions *difficult to understand. Such a semantic definition would be machine-dependent.*

3. Do Sebesta Problem Set 2a, 2b on page 181 – check your result against the definition

   2a: Write a EBNF for a Java class definition header statement

   http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html

```
ClassDeclaration :== NormalClassDeclaration | EnumDeclaration
NormalClassDeclaration:== ClassModifiersopt class Identifier
TypeParametersopt Superopt Interfacesopt ClassBody
```

   2b: Write a EBNF for a Java method call statement

   http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.12

```
MethodInvocation :== MethodName ( ArgumentListopt )
    | Primary . NonWildTypeArgumentsopt Identifier ( ArgumentListopt
) | super . NonWildTypeArgumentsopt Identifier ( ArgumentListopt ) |
ClassName . super . NonWildTypeArgumentsopt Identifier (
ArgumentListopt )
    | TypeName . NonWildTypeArguments Identifier ( ArgumentListopt )
```

Primary = on objects and “this”

Super = super class method

ClassName = static methods

TypeName = interface

4. Do Sebesta Problem Set 4 on page 181

   4. *Rewrite the BNF of Example 3.4 to add the ++ and -- unary operators of Java*

Old:

```
<assign> = <id> "=" <expr>
<id> = A | B | C
<expr> = <expr> "+" <term>
      | <term>
<term> = <term> "*" <factor>
      | <factor>
<factor> = "("<expr>")"
      | <id>
```

New: Added ++ and -- unary operators

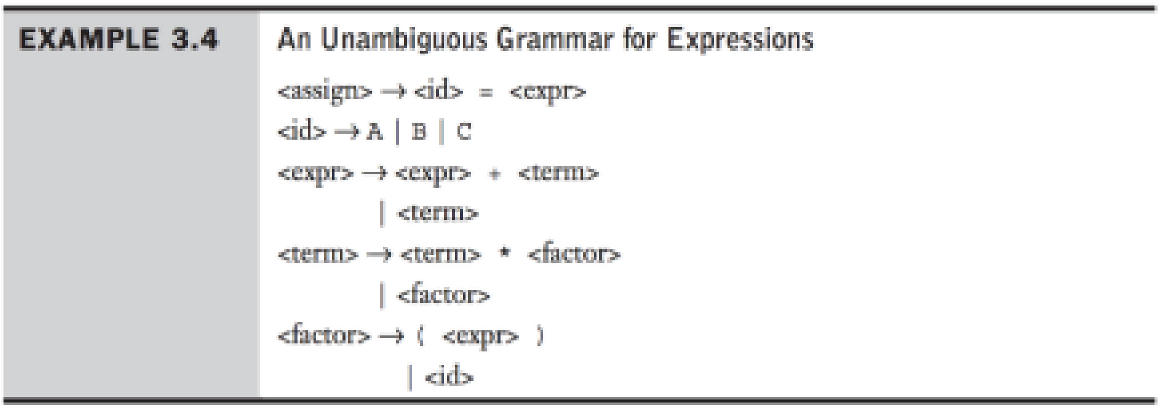
```
<assign> = <id> "=" <expr>
<id> = A | B | C
<expr> = <expr> "+" <term>
       | <term>
<term> = <term> "*" <factor>
       | <factor>
<factor> = "("<expr>")"
       | <id>
       | <id> ++
       | <id> --
       | ++ <id>
       | -- <id>
```

5. Go through the following material
   ○ Skim the paper “Status Report: Specifying JavaScript with ML”
   ○ Skim the Web article A brief history of ECMAScript versions ○
   Also browse the website http://www.jscert.org/index.html

Group Exercises - Lecture 4

1. Discuss the outcome of the individual exercises
   Did you all agree on the results?

2. Do Sebesta exercise 3 on page 183
   3. *Rewrite the BNF of Example 3.4 to to give + precedence over * and force + to be right associative*



EXAMPLE 3.4 An Unambiguous Grammar for Expressions

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
       | <term>
<term> → <term> * <factor>
       | <factor>
<factor> → ( <expr> )
         | <id>
```

Precedence: switch + and * in grammar
Make + right associative: Switch the order of factor and term
Result:

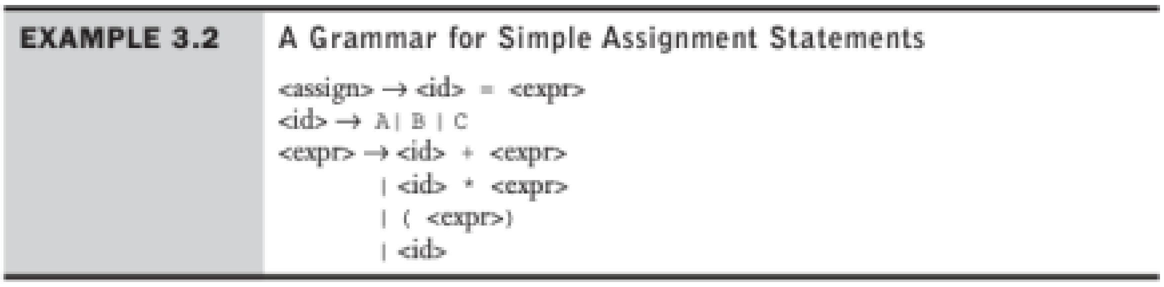
```
<assign> = <id> "=" <expr>
```

```
<id> = A | B | C
<expr> = <expr> "*" <term>
         | <term>
<term> = <factor> "+" <term>
         | <factor>
<factor> = "(" <expr> ")" (factor could be renamed)
           | <id>
```

3. Do Sebesta exercise 6a on page 183

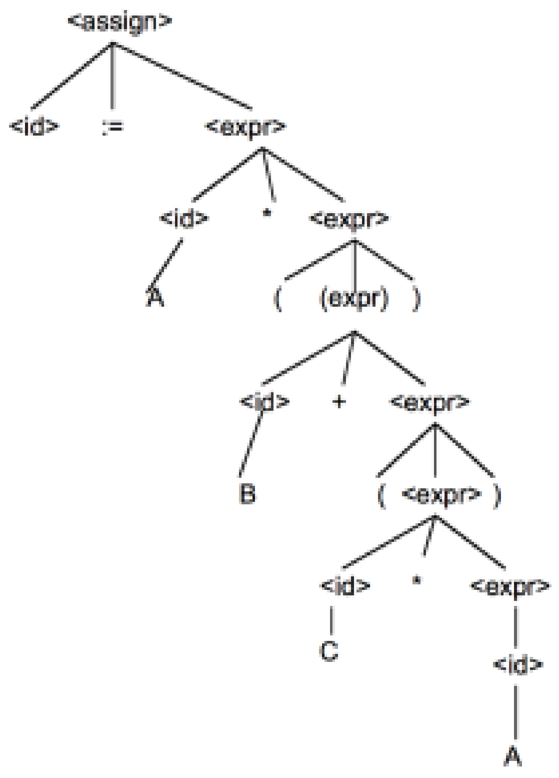
   6a. Using the grammar in Example 3.2, show a parse tree and a leftmost derivation for each of the following statements:

   (a) A = A * (B + (C * A))



EXAMPLE 3.2 A Grammar for Simple Assignment Statements

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
         | <id> * <expr>
         | ( <expr> )
         | <id>
```



6.

(a) <assign> => <id> = <expr>

=> A = <expr>

=> A = <id> * <expr>

=> A = A * <expr>

=> A = A * (<expr>)

=> A = A * (<id> + <expr>)

=> A = A * (B + <expr>)

=> A = A * (B + (<expr>))

=> A = A * (B + (<id> * <expr>))

=> A = A * (B + (C * <expr>))

=> A = A * (B + (C * <id>))

=> A = A * (B + (C * A))

4. Do Sebesta exercise 8 on page 184

8. Prove that the following grammar is ambiguous:

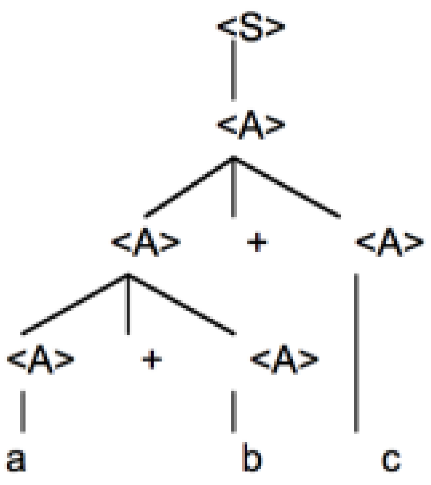
<S> → <A>

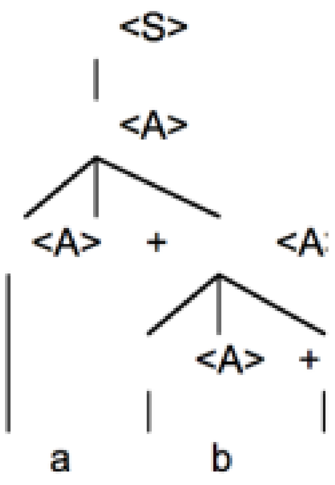
<A> → <A> + <A> | <id>

<id> → a | b | c

The international edition the book uses * instead of +, x, y, z instead of a, b, c

Use string : a + b + c which has two parse trees a + (b + c) vs. (a + b) c





5. Discuss why the specification of ECMAScript version 4 was abandoned

They could not agree on which features to include, so they decided to split the feature set into multiple versions (ES6, 7 8 and so on). ES4 was supposed to be a radical addition to JavaScript, but they just split it into multiple versions. This made it easier to agree on which features to use in ES5.

See examples for ES5 here: https://www.w3schools.com/js/js_es5.asp

6. Discuss why the specification of ECMAScript version 5 is now being formalized and mechanized

Most browsers (major vendors) have supported ES5 since 2012-2013. Most browsers support ES6 (2016-2017). The amount of support warrants a formalization of ES5.

Individual Exercises - Lecture 5

1. Follow the studio associated with Crafting a Compiler Chapter 4

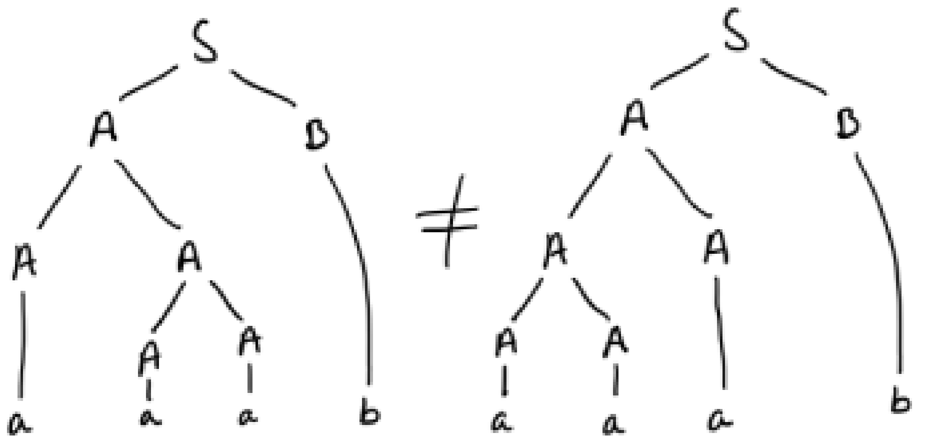
http://www.cs.wustl.edu/~cytron/cacweb/Chapters/4/studio.shtml

For this a virtual machine is provided on moodle with the necessary tools pre installed.

2. For each of the grammars below, describe the language associated with the grammar and determine if the grammar is ambiguous. If the grammar is ambiguous show two parse trees for the same string, otherwise explain why the grammar is not ambiguous (exercise 1 on page 168 in GE) – you may want to do this exercise by hand first and then try out the KfG tool in the AtoCC toolset: http://www.atocc.de/

- a)
  - S -> A B
  - A -> A A | a
  - B -> B B | b

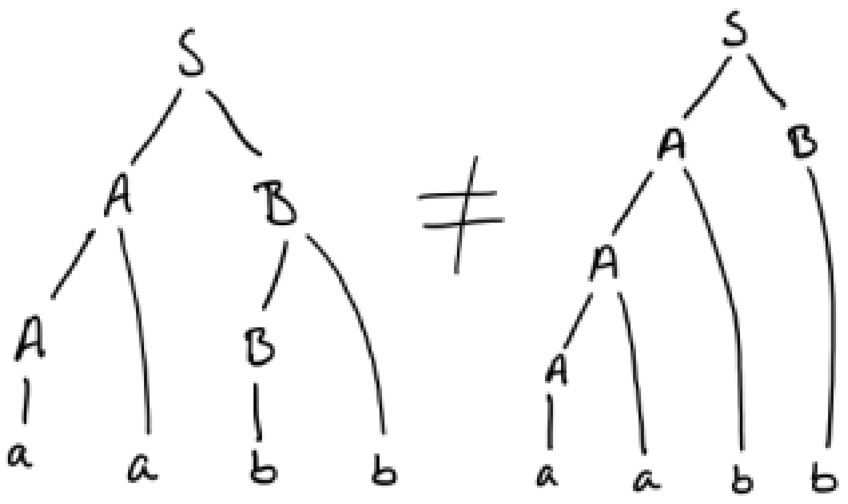
  Ambiguous. Example string: aaab



- b)
  - S -> A B
  - A -> A a | a
  - B -> B b | b

  Not ambiguous
- c)
  - S -> A B
  - A -> A a | A b | a
  - B -> B a | B b | b

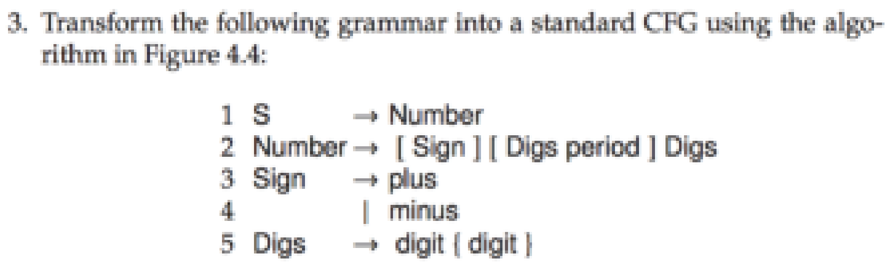
Ambiguous. Example string: aabb



- ○ d)
  - ■ S -> A B
  - ■ A -> A a
  - ■ B -> B b

D: Not ambiguous, but can never reach sentential form anyway, since no derivation contains only terminals.

3. Do Fischer et al exercise 3 on page 138 and exercise 10 on page 5 (exercises 5 and 13 on pages 169-171 in GE)



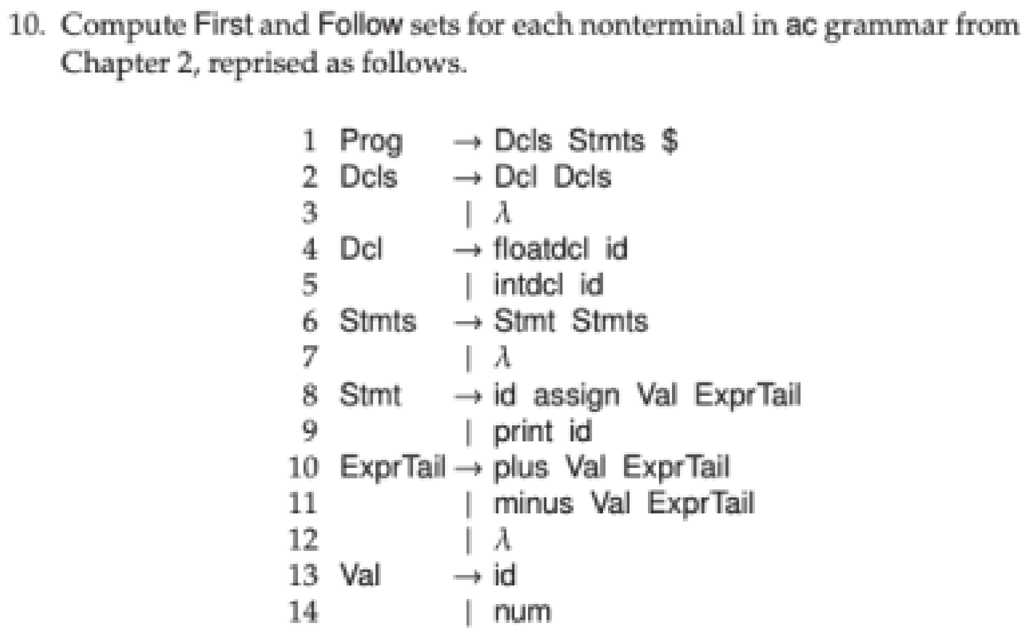
3. Transform the following grammar into a standard CFG using the algorithm in Figure 4.4:

```
1  S       → Number
2  Number  → [ Sign ] [ Digs period ] Digs
3  Sign    → plus
4          |  minus
5  Digs    → digit { digit }
```

The algorithm takes a CFG in EBNF and transforms it to BNF standard form. First we expand Sign to be N1, which either derives Sign or nothing (lambda), since it was originally optional. Next we expand Digs periode to be Digs periode or nothing (lambda), to achieve optionality. Digs is expanded to understand repetition ({ }) by adding a new non-terminal token, which can derive digits

recursively. Result:

1	S	$\rightarrow$	Number
2	Number	$\rightarrow$	N_1 N_2 Digs
3	N_1	$\rightarrow$	Sign
4		$\mid$	λ
5	N_2	$\rightarrow$	Digs period
6		$\mid$	λ
7	Sign	$\rightarrow$	plus
8		$\mid$	minus
9	Digs	$\rightarrow$	digit M
10	M	$\rightarrow$	digit M
11		$\mid$	λ



10. Compute First and Follow sets for each nonterminal in ac grammar from Chapter 2, reprised as follows.

1	Prog	$\rightarrow$	Dcls Stmts \$
2	Dcls	$\rightarrow$	Dcl Dcls
3		|	λ
4	Dcl	$\rightarrow$	floatdcl id
5		|	intdcl id
6	Stmts	$\rightarrow$	Stmt Stmts
7		|	λ
8	Stmt	$\rightarrow$	id assign Val ExprTail
9		|	print id
10	ExprTail	$\rightarrow$	plus Val ExprTail
11		|	minus Val ExprTail
12		|	λ
13	Val	$\rightarrow$	id
14		|	num

First(Prog) = {floatdcl, intdcl, id, print, $}

Follow(Prog) = {}

First(dcls) = {floatdcl, intdcl}

Follow(dcls) = {id, print, $}

First(dcl) = {floatdcl, intdcl)

Follow(dcl) = {floatdcl, intdcl, id, print, $}
First(stmts) = {id, print}

Follow(stmts) = {$}

First(stmt) = {id, print}

Follow(stmt) = {id, print, $}

First(ExprTail) = {plus, minus}

Follow(ExprTail) = {id, print, $}

First(Val) = {id, num}

Follow(Val) = {plus, minus, print, id, $}

4. (optional) Try the ACLA (Ambiguity Checking with Language Approximations) on the grammars from exercise 2 (exercise 1 on page 168 in GE). You can find the ACLA tool on http://services2.brics.dk/java/grammar/demo.html Did your answers agree with the answers from the tool?

5. (optional) You may also try Context Free grammar Tool on http://smlweb.cpsc.ucalgary.ca/start.html or its newest version on http://mdaines.github.io/grammophone/
This is a good exercise for getting better at writing unambiguous grammars as well as getting a better understanding of when a grammar is LL(1), LR(0) etc.

6. (optional) Browse the Grammar Zoo web site http://slebok.github.io/zoo/
Look up your favorite language

7. (optional) Browse the Syntax across languages web site:
http://rigaux.org/language-study/syntax-across-languages/
As an example, have a look at how many different ways a float can be declared!

8. (optional) Do Sebesta Review questions 1,2,4,5,8,10,24 on page 216-217

1. What are the reasons why using BNF is advantageous over using an informal syntax description?
It is very difficult (or even impossible?) to create an unambiguous description of a programming language using sentences in english. BNF allows us to specify grammar that can only be interpreted one way. A formal definition (fx in BNF) can help unify language definition across multiple compiler implementations..

2. How does a lexical analyzer serve as the front end of a syntax analyzer? The lexical analyser performs analysis at the lowest level of program structure, i.e. the individual tokens / lexemes.
4. How can you construct a lexical analyzer with a state diagram? The lexical analyzer can be constructed and modelled as a finite automata. The tokens of a programming language are a regular language, and a lexical analyser is a finite automaton.

5. Describe briefly the three approaches to building a lexical analyzer. 1. Write a formal description of the token patterns in a descriptive language related to regular expressions and then generate the lexical analyser from this. There exist many tools

for this (the original being “Lex”).
2. Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.
3. Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.

8. What are the two distinct goals of syntax analysis?
1. Check whether a given program is syntactically correct, and if not give proper error messages.
2. Produce a complete parse tree that can be used for translation.

10. Describe the recursive-descent parser.
A recursive-descent parser is a simple type of parser, that has a subprogram of each nonterminal in its associated grammar. Recursive-descent parsers do, however, have limitations, for example that it cannot handle left recursion in the grammar, due to the nature of recursive-descent parsers. They always expand from the left, which will obviously lead to stack overflow.

24. Why is a bottom-up parser often called a shift-reduce algorithm? Bottom-up parsers are often called shift-reduce algorithms because shift and reduce are the two most common actions they specify.

Group Exercises - Lecture 5

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Do Fischer et al exercise 1, 7, 8, 9, 13 on pages 138- 141 (exercises 4, 8, 10, 11, and 15 on pages 169-174 in GE)
   1. *While ambiguity is avoided in programming languages, (some) humor can be derived from ambiguity in natural languages. For each of the following English sentences, explain why it is ambiguous. First try to determine multiple grammar diagrams for the sentence. If only one such diagram exists, explain why the meaning of the words makes the sentence ambiguous.*

   (a) I saw an elephant in my pajamas.
   (b) I cannot recommend this student too highly.
   (c) I saw her duck.
   (d) Students avoid boring professors.
   (e) Milk drinkers turn to powder.

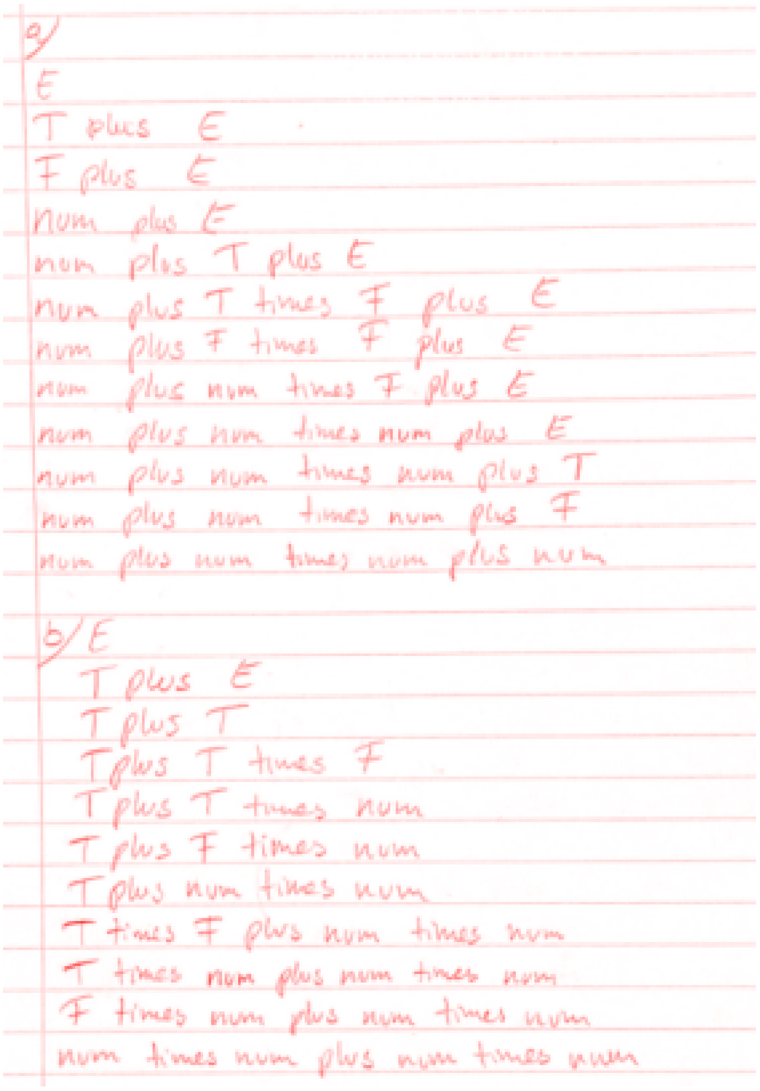
   (a) Was I wearing the pyjamas or was it the elephant?
   (b) Irony or not?
   (c) Is duck a verb or a noun?
   (d) Is boring an adjective or a verb?

(e) Do milk drinkers turn INTO powder or turn to powder as an alternative?

7. *A grammar for infix expressions follows*

```
1  Start → E $
2  E     → T plus E
3        | T
4  T     → T times F
5        | F
6  F     → ( E )
7        | num
```

(a) Show the leftmost derivation of the following string.
num plus num times num plus num $



a/

E

T plus E

F plus E

num plus E

num plus T plus E

num plus T times F plus E

num plus F times F plus E

num plus num times F plus E

num plus num times num plus E

num plus num times num plus T

num plus num times num plus F

num plus num times num plus num

b/ E

T plus E

T plus T

T plus T times F

T plus T times num

T plus F times num

T plus num times num

T times F plus num times num

T times num plus num times num

F times num plus num times num

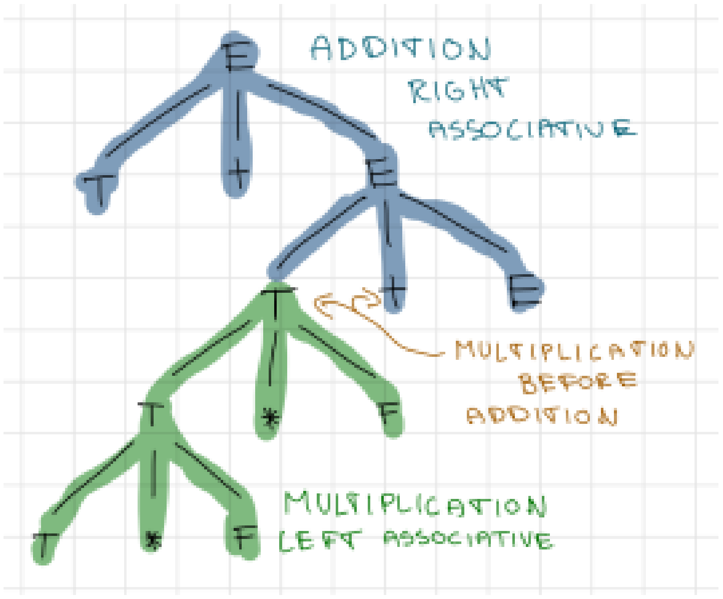
num times num plus num times num

(b) Show the rightmost derivation of the following string.
num times num plus num times num $

(c) Describe how this grammar structures expressions, in terms of the precedence and left- or right- associativity of operators.

Multiplication has precedence over addition (because it is located lower in the parse tree) Addition is right associative (because addition expands right in the parse tree)

Multiplication is left associative (because multiplication expands left in the parse tree)


ADDITION RIGHT ASSOCIATIVE
MULTIPLICATION BEFORE ADDITION
MULTIPLICATION LEFT ASSOCIATIVE
E
T
+
E
T
+
E
T
*
F
T
*
F

8. Consider the following two grammars.

(a)

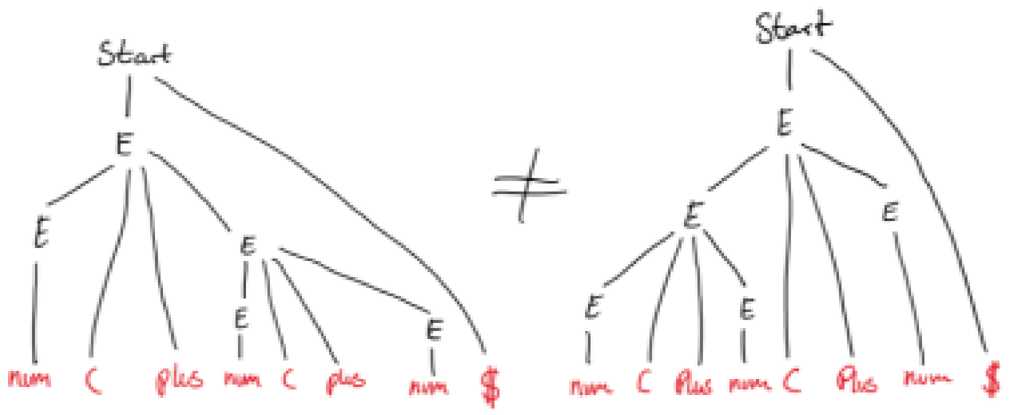
```
1  Start → E $
2  E     → ( E plus E
3        | num
```

(b)

```
1  Start → E $
2  E     → E ( plus E
3        | num
```

Which of these grammars, if any, is ambiguous? Prove your answer by showing two distinct derivations of some input string for the ambiguous grammar(s).

(a) Not ambiguous
(b) It is ambiguous, proven by the following sentence "num (plus (num (plus num" which can be derived in two different ways yielding two different parse trees.



9. Compute First and Follow sets for the nonterminals of the following grammar.

```
1  S → a S e
2    | B
3  B → b B e
4    | C
5  C → c C e
6    | d
```

First(S) = {a, b, c, d}
Follow(S) = {e}

First(B) = {b, c, d}
Follow(B) = {e}

First(C) = {c, d}
Follow(C) = {e}

Individual Exercises - Lecture 6

1. (optional - but recommended) Follow the studio associated with Crafting a Compiler Chapter 3 http://www.cs.wustl.edu/~cytron/cacweb/Chapters/3/studio.shtml A virtual machine is available on Moodle with the necessary tools pre installed.

2. Do Fischer et al exercise 1, 2, 9, 22 on pages 106-112 (exercise 3, 4, 14, 23 on pages 134-141 in GE)

   1. Assume the following text is presented to a C scanner:

```
main(){
    const float payment = 384.00;
    float bal;
    int month = 0;
    bal=15000;
    while (bal>0){
        printf("Month: %2d  Balance: %10.2f\n", month, bal);
        bal=bal-payment+0.015*bal;
        month=month+1;
    }
}
```

What token sequence is produced? For which tokens must extra information be returned in addition to the token code?

ID(main), LPAREN, RPAREN, LBRACE

CONST, FLOATDCL, ID(payment), ASSIGN, FLOATNUM(384.00), SEMICOLON

FLOATDCL, ID(bal), SEMICOLON

INTDCL, ID(month) ASSIGN INTNUM(0) SEMICOLON

Etc…

2. How many lexical errors (if any) appear in the following C program? How should each error be handled by the scanner?

```
main(){
        if(1<2.)a=1.0else a=1.0e-n;
        subr('aa',"aaaaa (newline in string not allowed without \ )
                aaaaa");
        /* That's all (*/ missing)
}
```

The blue letters will cause errors. The first one needs a decimal after the decimal point. The scanner could then give an error message saying that this is the case. The second one needs space between 0 and e, otherwise the scanner cannot create a token from it. This is a bit harder since C actually supports scientific numbers like

1.5321E10, so the error message might suggest either creating a scientific number. C also does not allow multiple characters in a single quote string. It cannot suggest that it would expect a space followed by else, since if statements in C do not require an else clause. The scanner will also have problems with the last comment, since it is not closed. The error message for this could be quite difficult, since an unclosed multiline comment simply in effect erases the rest of the file. Now the parser must see it while creating the function, since it is no longer closed in curly brackets.

9. Most compilers can produce a source listing of the program being compiled. This listing is usually just a copy of the source file, perhaps embellished with line numbers and page breaks. Assume you are to produce a prettyprinted listing.

(a) How would you modify a Lex scanner specification to produce a prettyprinted listing (that is, a listing with text properly indented, if-else pairs aligned, and so on)?
The lex scanner should keep track of and add information about each token's line and its number of indentation.

(b) How are compiler diagnostics and line numbering complicated when a prettyprinted listing is produced?
One token could represent text from multiple lines.

22. Write Lex regular expressions (using character classes if you wish) that match the following sets of strings:

(a) The set of all unprintable ASCII characters (those before the blank and the very last character)
[^ -~]+
Explanation: In the ASCII table the printable characters start with white space and ends with tilde. Using the ^ says, that we want anything that is not in this range, i.e. all the non-printable characters.
(b) The string ["""] (that is, a left bracket, three double quotes, and a right bracket)
\[\”\”\”\]
(c) The string x^12,345 (your solution should be far less than 12,345 characters in length)
X{12345}
This accepts exactly 12345 X's in a row.

3. *Write a regular expression definition for unsigned integer literals excluding those that contain unnecessary leading zeroes. Thus 0, 1, 123, 10000 are included, but 00, 0123 and 0010000 are excluded.*
   ([1-9][0-9]*)|0
   Either we have 0 or we have one non-zero number followed by zero or more numbers from 0 to 9.
4. *You have scanned an integer literal into a character buffer (e.g. using yytext). You now*

want to convert the string representation to a numeric (int) form. However, the string may represent a value too large for the int form. Explain how to convert a string representation of an integer literal into numeric form with full overflow checking.

You could read the string into a char buffer, each entry representing one digit of the number. Then loop through the digits of the buffer from left to right, adding them to a variable as such:

```
char[] digit_buffer = ... // *Read input number here*
int number = 0;
for (digit in digit_buffer)
    number = number * 10 + digit

if (number > int32.MAX_VALUE)
    throw OverFlowException()
```

As we are working in a base 10 number system, we multiply by 10 for each digit added.

This solution assumes that the language of the scanner supports large enough ints or has its own overflow detection system.

To avoid depending on overflow checking in the scanner language, you can define a string containing the largest allowed number and then compare it to the input string one digit at a time:

```
const char[] largest_allowed_number = new char[]{2, 1, 4, 7, ...}
char[] digit_buffer = ... # *Read input number here*

# We have an overflow if there are more digits than allowed
if (digit_buffer.length > largest_allowed_number.length)
throw OverFlowException()
# If the numbers have the same length we must check the digits
# individually left to right
if (digit_buffer.length == largest_allowed_number.length)
   for (i in range(digit_buffer.length))
       if (digit_buffer[i] > largest_allowed_number[i])
          throw OverFlowException();
       if (digit_buffer[i] < largest_allowed_number[i])
          break;
```

5. *(optional) Modify the studio associated with Crafting a Compiler Chapter 3 to produce a lexer for the ac language from Lecture 3 using the lexical grammar for ac in figure 2.3 on page 36 in (64 in GE edition) of Fischer et. al. Take a look at the Yylex source*

file in the autogen package and compare the code with the hand written ScannerCode class from Studio 2

You can use the virtual machine available on Moodle for this exercise.

Group Exercises - Lecture 6

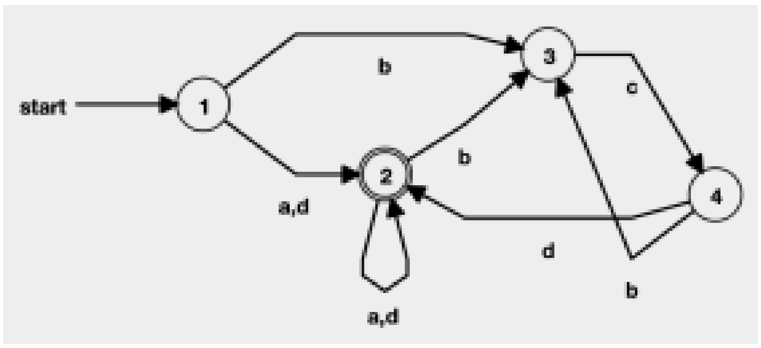
The following exercises are best done as group discussions:

1. Discuss the outcome of the individual exercises

   Did you all agree on the exercise?

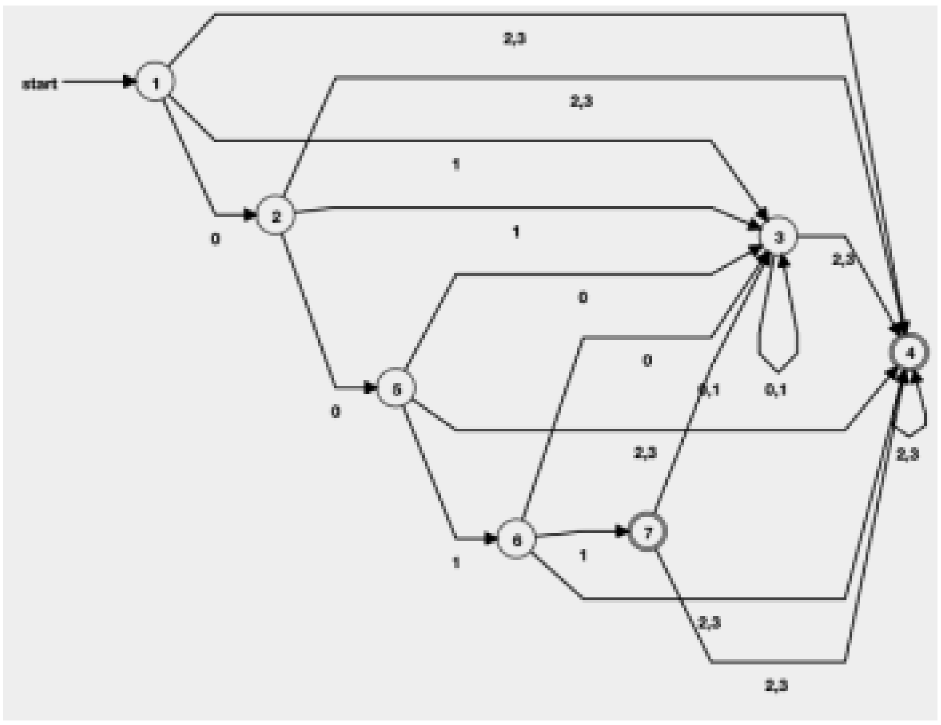
2. Do Fischer et al exercises 4, 6, 7, 8 on pages 106-112 (exercises 5, 6, 7, 9, 12, 13 on pages 134-141 in GE)

   4. Write DFAs that recognize the tokens defined by the following regular expressions:
   Note that in the following DFAs there are implicit transitions (ie. not shown on the figure) from each state to an error state, whenever an illegal input is read. (a)
   (a | (bc)*d)+

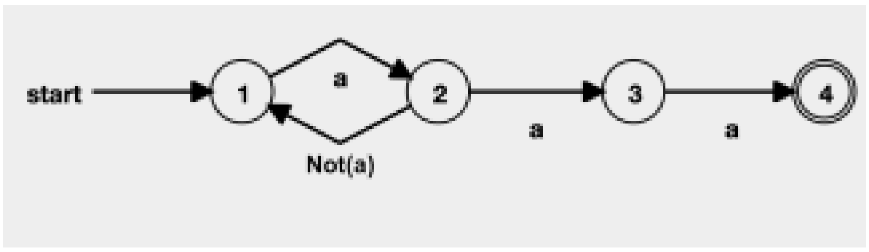


3. (b)

   ((0 | 1)*(2 | 3)+) | 0011



(c)
(a Not(a))*aaa



6. Write a regular expression that defines a C-like comment delimited by /* and */. Individual *'s and /'s may appear in the comment body, but the pair */ may not.
(\/*)(.|\n)*?(*\/)
We start with a /* with both characters having to be escaped with backslash. This may be followed by any number of characters and lastly followed by a */ (again escaped). The ? makes it stop the first time, it finds a */ in the end.

7. Define a token class AlmostReserved to be those identifiers that are not reserved words but that would be if a single character were changed. Why is it useful to know that an identifier is "almost" a reserved word? How would you generalize a scanner to recognize AlmostReserved tokens as well as ordinary reserved words and identifiers?
AlmostReserved tokens could be used to identify possible syntax errors caused by misspelling reserved keywords. This can be used to provide a warning for possible (off-by-one) syntax errors, and make it easy for the compiler to provide a "did you mean x?" error message to the user.

The grammar used to drive the parser must be rewritten to treat the token classes

AlmostReserved and Identifier as interchangeable until an error is recognized. If the parsing error occurred in a state that would accept a reserved word and the input token was AlmostReserved, then the value of the token would be checked to see if it was a variant of the expected reserved word. If so, the reserved word could be assumed to be the intended input token and parsing could be restarted from the state it was in before the error.

AlmostReserved tokens would have to be recognized by using an extension of one of the reserved word lookup techniques described at the end of Section 3.7.1 on page 79. Because of the large number of identifiers that are one character change removed from reserved words, it would cause a huge increase in the number of states in the scanner tables to recognize them by generating regular expressions to recognize each of the possibilities.

8. When a compiler is first designed and implemented, it is wise to concentrate on correctness and simplicity of design. After the compiler is fully implemented and tested, you may need to increase compilation speed. How would you determine whether the scanner component of a compiler is a significant performance bottleneck? If it is, what might you do to improve performance (without affecting compiler correctness)?

To test the performance one could test the entire compiler with a profiler that checks how much time is spent in each function. This would reveal if some scanner functions were slow. Alternatively one could collect start time and end time for each compiler phase to reveal if one phase (for example the parsing phase) is much slower than the others.

To improve the scanning speed one could use a scanner generator like Flex or GLA, which are designed to generate very fast scanners.

If your scanner is hand-coded, avoid reading single characters, which can be very expensive. Instead bulk load into buffers, perhaps even double buffers. Performance can also be increased by copying as few characters as possible from the input buffers to the tokens. For example only copy the values, if they are needed for later analysis, for example literal values.

4. (optional) Read the Lab associated with Crafting a Compiler Chapter 3 http://www.cs.wustl.edu/~cytron/cacweb/Chapters/3/lab.shtml and take a look at the "official" solution in the file FischerLab356solutions.zip in the General Course material directory https://www.moodle.aau.dk/pluginfile.php/154451/mod_folder/content/0/FischerLab356solutions.zip?forcedownload=1 (You may of course Follow the Lab without looking at the solution, but be warned that this is a tough one) For this a virtual machine will be provided with the necessary tools pre installed.

Individual Exercises - Lecture 7

1. (optional - recommended) Follow the studio associated with Crafting a Compiler Chapter 5
   http://www.cs.wustl.edu/~cytron/cacweb/Chapters/5/studio.shtml
   For this a virtual machine with the necessary tools pre installed is available on moodle.

2. Do Fischer et al exercise 1, 2 (d,e optional), 5, 6 (optional) on pages 173-178
   1. For each of the following grammars, determine whether the grammar is 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. $\mathit{first}[X_1], \mathit{first}[X_2], \ldots, \mathit{first}[X_n]$ are all pairwise disjoint
2. *If* $X_i \Rightarrow^* \varepsilon$ then $\mathit{first}[X_j] \cap \mathit{follow}[X] = \emptyset$, for $1 \le j \le n. i \ne j$

If G is ε-free then 1 is sufficient

(a)
```
1 S → A B c
2 A → a
3   | λ
4 B → b
5   | λ
```

(c)
```
1 S → A B B A
2 A → a
3   | λ
4 B → b
5   | λ
```

(b)
```
1 S → A b
2 A → a
3   | B
4   | λ
5 B → b
6   | λ
```

(d)
```
1 S → a S e
2   | B
3 B → b B e
4   | C
5 C → c C e
6   | d
```

The algorithm says that for each non-terminal the predict set for its productions must be disjoint, where the predict set is defined as:

$$\text{predict}(A \to \alpha) = \begin{cases} \text{first}(\alpha) & \text{if } \alpha \text{ does not derive } \lambda \\ \text{first}(\alpha) \cup \text{follow}(A) & \text{if } \alpha \to^* \lambda \end{cases}$$

a)
First we create the predict set for A.
Predict(A -> a) = {a}
Predict(A -> λ) = {b, c}
This rule is valid for an LL(1) parser, since both productions are disjoint.

Predict(B -> b) = {b}
Predict(B -> λ) = {c}
This rule is valid for an LL(1) parser, since both productions are disjoint.

b)
Predict(A -> a) = {a}
Predict(A -> B) = {b}
Predict(A -> λ) = {b}
Since Predict(A -> B) == Predict(A -> λ), i.e not disjoint, this rule is not valid for an LL(1) parser, since it cannot distinguish the two options.
The grammar is also ambiguous.

c)

Predict(A -> a) = {a}
Predict(A -> λ) = {a, b}
Since a ∈ Predict(A -> a) and a ∈ Predict(A -> λ) this rule, and therefore the entire grammar, is not valid for an LL(1) parser.
This grammar is also ambiguous.

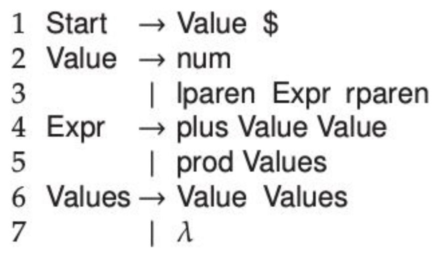
d)

Predict(S -> a S e) = {a}
Predict(S -> B) = {b, c, d}
No conflicts

Predict(B -> b B e) = {b}
Predict(B -> C) = {c ,d}
No conflicts

Predict(C -> c C e) = {c}
Predict(C -> d) = {d}
No conflicts

2. Consider the following grammar, which is already suitable for LL(1) parsing:



1	Start	→	Value $
2	Value	→	num
3		|	lparen Expr rparen
4	Expr	→	plus Value Value
5		|	prod Values
6	Values	→	Value Values
7		|	λ

(a) Construct first and follow sets for each nonterminal in the grammar
First(Value) = {num, lparen}
First(Expr) = {plus, prod}
First(Values) = {num, lparen}

Follow(Value) = {$, num, lparen, rparen}
Follow(Expr) = {rparen}
Follow(Values) = {rparen}

(b) Construct Predict sets for the grammar

Predict(Value -> num) = {num}
Predict(Value -> lparen Expr rparen) = {lparen}

Predict(Expr -> plus Value Value) = {plus}
Predict(Expr -> prod Values) = {prod}

Predict(Values -> Value Values) = {num, lparen}
Predict(Values -> λ) = {rparen}

C) Construct recursive-descent parser based on the grammar

Here is a pseudo code version of a recursive-descent parser. In the comments of the code you can see why we peek for the tokens, that we do. We use the mechanical approach of first checking the predict set for each production to tell which one should be used. It is done in the order in which they appear in the grammar. For lambda productions the follow-set is checked.

```
Procedure Start()
      call Value()
      call Match(ts, $)
end
```

```
Procedure Value()
      // Predict(Value -> num) = {num}
      if ts.peek() = num
      then
            call Match(ts, num)
      // Predict(Value -> lapren Expr rparen) = {lparen}
      else if ts.peek() = lparen
      then
            call Match(ts, lparen)
            call Expr()
            call Match(ts, rparen)
      else
            call ERROR()
end
```

```
Procedure Expr()
      // Predict(Expr -> plus Value value) = {plus}
      if ts.peek() = plus
      then
            call Match(ts, plus)
            Value()
            Value()
      // Predict(Expr -> prod Values) = {prod}
      else if ts.peek() = prod
      then
            call Match(ts, prod)
            call Values()
      else
            call ERROR()
end
```

```
Procedure Values()
      // Predict(Values -> Value Values) = {num, lparen}
      if ts.peek() ∈ {num, lparen}
      then
            call Value()
            call Values()
      else
            // Follow(Values) = {rparen}, due to lambda production
            if ts.peek() ∈ {rparen}
            then
                  /* Do nothing for lambda-production */
            else
                  call ERROR()
end
```

D) (OPTIONAL) Add code into the parser to compute sums and products as indicated
You could change the Value() and Expr() to return a number, and change Values() to return the list of numbers that are read.
Inside the Expr() function you could then add or multiply the numbers returned from Value()/Values() calls based on whether the token you read is a ‘plus’ or ‘prod’.

E) (OPTIONAL) Build an LL(1) parse table on the grammar
A parse table tells us which rule to use depending on which rule we are inside and which token we get as the next one. You can use the predict sets to construct this table.
There is very good example of an LL(1) parser using a parse table here:
http://www.cs.ecu.edu/karl/5220/spr16/Notes/Top-down/LL1.html

5. Transform the following grammar into LL(1) form using the techniques presented in Section 5.5:

1	DeclList	→	DeclList ; Decl
2		|	Decl
3	Decl	→	IdList : Type
4	IdList	→	IdList , id
5		|	id
6	Type	→	ScalarType
7		|	array (ScalarTypeList) of Type
8	ScalarType	→	id
9		|	Bound .. Bound
10	Bound	→	Sign intconstant
11		|	id
12	Sign	→	+
13		|	−
14		|	λ
15	ScalarTypelist	→	ScalarTypeList , ScalarType
16		|	ScalarType

Grammars are not LL(1) because of prediction conflicts. Two common causes of prediction conflicts are left recursion and common prefixes. This grammar contains both. Productions 1, 4 and 15 are left recursive; they will need to be rewritten. More subtly, productions 8 and 9 share a common prefix, id. This is because Bound can generate an initial id. Hence productions 8 and 9 will need revision.

Method EliminateLeftRecursion in Figure 5.15 on page 157 transforms left-recursive productions into an equivalent LL(1) form. The idea is that a left-recursive production may be used zero or more times before another non- left-recursive production is used. The net effect is to generate the result of the non-left-recursive production followed by zero or more occurrences of the left-recursive production's "tail." Thus for productions 1 and 2, a Decl must be produced by production 2 followed by zero or more "tails" of production 1 "; Decl". So production 1 is rewritten to first generate Decl followed by a new non-terminal, DeclListTail. DeclListTail generates zero or more occurrences of "; Decl". The rewritten productions are:

DeclList → Decl DeclListTail
DeclListTail → ; Decl DeclListTail
| λ

Similar transformations are applied to productions 4 and 5, and 15 and 16, yielding:

```
IdList       → id IdListTail
IdListTail →  , id IdListTail
                  |  λ

ScalarTypeList       → ScalarType ScalarTypeListTail
ScalarTypeListTail → , ScalarType ScalarTypeListTail
                              |   λ
```

To solve the prediction conflict between productions 8 and 9, we split production 9 into two alternatives: a bound that begins with an identifier and a bound that begins with an integer. Then the two productions that begin with id are combined into one, with a BoundSuffix that generates the two valid bound suffixes. We have

```
ScalarType → id BoundSuffix
                    |  Sign intconstant .. Bound
BoundSuffix → .. Bound
BoundSuffix → λ
```

6. (Optional) Run your solution to Exercise 5 through any LL(1) parser generator to verify that it is actually LL(1). How do you know that your solution generates the same language as the original grammar?
Try using this online checker:
https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1/

This will check if the grammar is LL(1) and also generate a transition table.

The question of whether two CFGs describe the same language is very difficult. In fact, this class of problems is undecidable.

3. (Optional - but recommended for hands-on experience) Familiarize yourself with JavaCC, Coco/R and ANTLR by browsing their web pages
The virtual machine provided on Moodle has these tools available, along with a set of guides that walks you through a small example language for each tool.

Group Exercises - Lecture 7

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Do Fischer et al exercises 9 and 10 on pages 173-178 (exercises 9 and 12 on pages 205-210 in GE)

   9. Recall that an LL(k) grammar allows k tokens of lookahead. Construct an LL(2) parser for the following grammar:

1	Stmt	$\rightarrow$ id ;
2		| id (IdList) ;
3	IdList	$\rightarrow$ id
4		| id , IdList

```
Procedure Stmt()
    if ts.peek(2) == [id, ';']
         call match(ts, id)
         call match(ts, ';')
         // end of program
    else if ts.peek(2) == [id, '(']
         call match(ts, id)
         call match(ts, '(')
         IdList()
         call match(')')
         call match(';')
    else
         call error()

Procedure IdList()
    if ts.peek(2) == [id, ',']
         call match(ts, id)
         call match(ts, ',')
         IdList()
    else
         call match(ts, id)
```

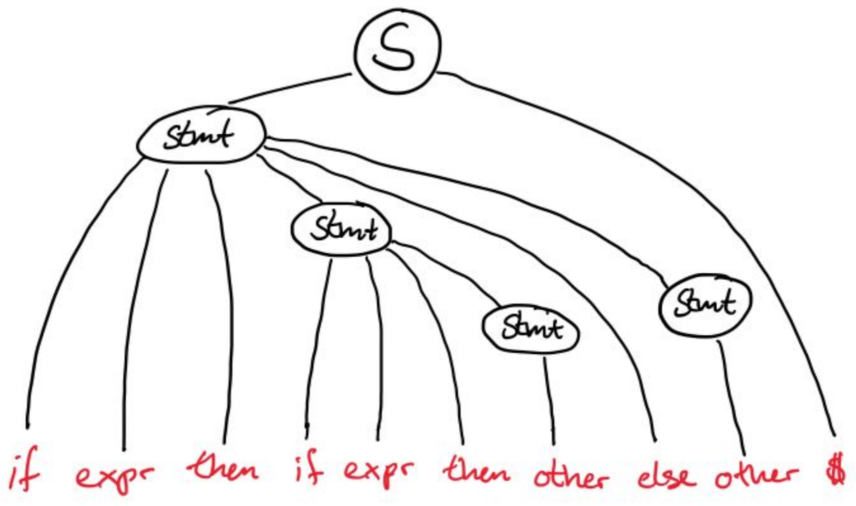
10. Show the two distinct parse trees that can be constructed for

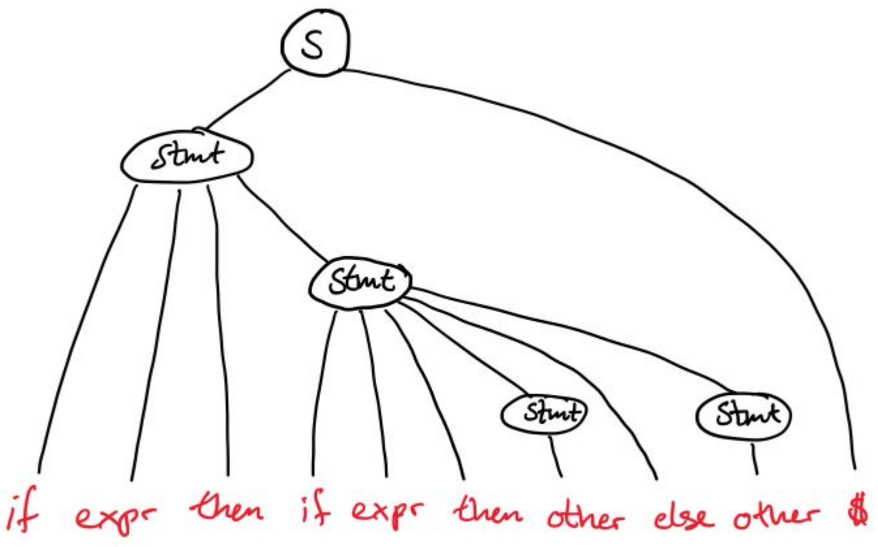
if expr then if expr then other else other

using the grammar given in Figure 5.17. For each parse tree, explain the correspondence of then and else.

1	S	→	Stmt \$
2	Stmt	→	if expr then Stmt else Stmt
3		|	if expr then Stmt
4		|	other

Figure 5.17: Grammar for if-then-else.





In the first tree the else clause is bound to the first if clause. In the second tree, the else clause is bound to the inner if clause.

3. Discuss pros. and cons. of writing a recursive descent parser by hand and using a tool
   Pros (of writing by hand):
   - You can add features, that would otherwise not be traditional feature of a parser
     - See: Python indentation tokens for scopes etc.
     - Or prettification of tokens to make them more readable
   - You get a better understanding of the inner workings of the parser

   Cons:
   - If the language is big with many constructs and tokens, it can take a long time and leaves plenty of room for error to write it by hand.
   - Constructing a parser is a very mechanical process, so if you understand the inner workings, there is no reason to repeatedly implement one.
   - Changes to the syntax of the language is more difficult with a hand crafted parser, whereas a parser generated would simply generate a new one from the modified grammar.

4. Discuss pros. and cons. of JavaCC, Coco/R and ANTLR
   Try out the tools inside the virtual machine provided on moodle! On the Desktop you will find pdf guides that walk you through a simple example for each tool.

   Other than the implementation experience itself, other pros include the fact that the generated parsers often include a lot of optimisations which make them very fast.

5. (optional - but recommended) Follow the Lab associated with Crafting a Compiler Chapter 5 http://www.cs.wustl.edu/~cytron/cacweb/Chapters/5/lab.shtml
   You can use the virtual machine provided on Moodle.

Individual Exercises - Lecture 8

1. (optional - but recommended) Follow the studio associated with Crafting a Compiler Chapter 6 http://www.cs.wustl.edu/~cytron/cacweb/Chapters/6/studio.shtml
   For this a virtual machine will be provided with the necessary tools pre installed.

2. Use **CUP** to do Fischer et al exercises 40, 41, 42, 43 on pages 224-233 (exercise 40, 42, 44, 43 on pages 263-264 in GE)
   The virtual machine available on Moodle has CUP and SableCC installed, along with a pdf guide that details how to use them!

   If you want to run CUP directly on your own machine:
   Download Cup -> Extract Jar file to desktop -> create grammar.cup file on desktop
   (optional - download intellij highlighting plugin for cup files)
   Recreate grammar from exercises in grammar.cup file
   Run in cmd :
   cd Desktop
   java -jar java-cup-11b.jar grammar.cup
   See output for answer to exercises

   Example grammar: http://www2.cs.tum.edu/projects/cup/docs.php#intro
   40. Recall the dangling else problem introduced in Chapter 5. A grammar for a simplified language that allows conditional statements follows:

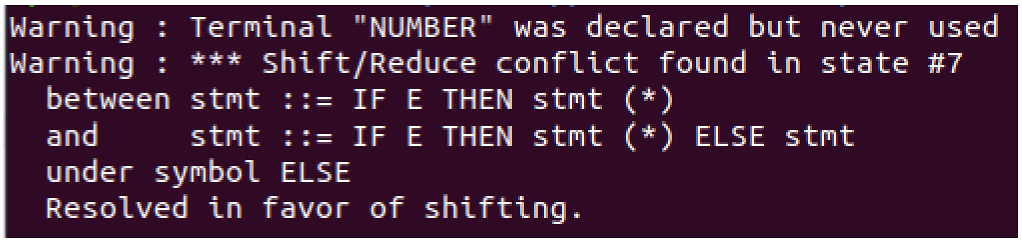
   ```
   1  Start → Stmt $
   2  Stmt → if e then Stmt else Stmt
   3       | if e then Stmt
   4       | other
   ```

   (a) Explain why the grammar is or is not LALR(1).

   First we create a grammar.cup file with the following contents:

   ```
   terminal IF, THEN, ELSE, E, OTHER;
   terminal Integer NUMBER;

   non terminal program, stmt;

   program ::= stmt;
   stmt ::= IF E THEN stmt ELSE stmt
          | IF E THEN stmt
          | OTHER;
   ```

Then we run the JavaCup parser generator with this input. The output warns us that there was a shift reduce conflict, but that it was automatically resolved by favoring shift:



```
Warning : Terminal "NUMBER" was declared but never used
Warning : *** Shift/Reduce conflict found in state #7
  between stmt ::= IF E THEN stmt (*)
  and     stmt ::= IF E THEN stmt (*) ELSE stmt
  under symbol ELSE
  Resolved in favor of shifting.
```

From this we can conclude that the grammar is not LALR(1).

41. Consider the following grammar

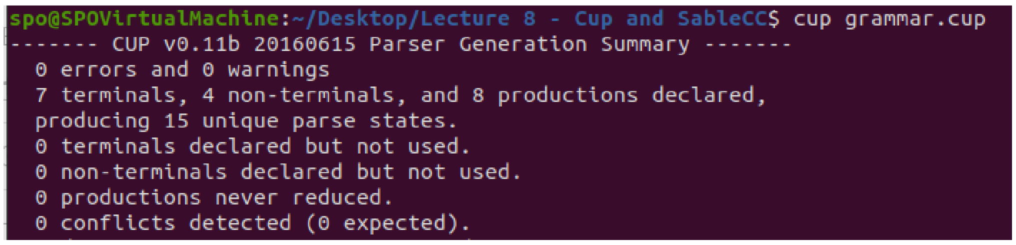
1	Start	→	Stmt $
2	Stmt	→	Matched
3		|	Unmatched
4	Matched	→	if e then Matched else Matched
5		|	other
6	Unmatched	→	if e then Matched else Unmatched
7		|	if e then Unmatched

(a) Explain why the grammar is or is not LALR(1)

First we create a grammar.cup file to represent the grammar:

```
terminal IF, THEN, ELSE, E, OTHER;

non terminal program, stmt, matched, unmatched;

program ::= stmt;
stmt    ::= matched | unmatched;

matched   ::= IF E THEN matched ELSE matched
            | OTHER;
unmatched ::= IF E THEN matched ELSE unmatched
            | IF E THEN unmatched;
```

Then we run the CUP parser generator with this grammar:



```
spo@SPOVirtualMachine:~/Desktop/Lecture 8 - Cup and SableCC$ cup grammar.cup
------- CUP v0.11b 20160615 Parser Generation Summary -------
  0 errors and 0 warnings
  7 terminals, 4 non-terminals, and 8 productions declared,
  producing 15 unique parse states.
  0 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
```

This does not yield any errors, warnings or conflicts, which means that we have successfully generated a LALR(1) parser for this grammar, thereby proving that the language is LALR(1).

(b) Is the language of this grammar the same as the language of the grammar in Exercise 40? Why or why not?

Recall that the problem of determining whether two given grammars generate the same language is undecidable! This means that the tool cannot have an algorithm that answers this question for all pairs of grammars. It should, however, be noted that in special cases it is possible to determine if two grammars describe the same language.

For this specific example, we can reason that these two grammars do not generate the same language:

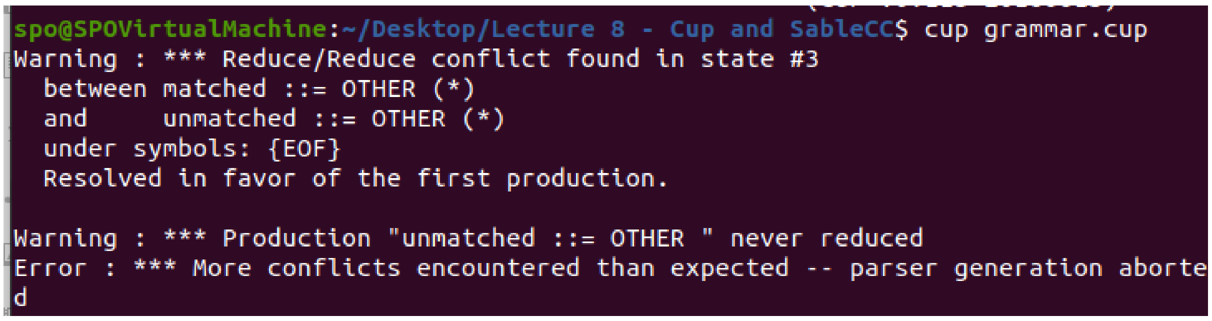
Looking at the grammar from exercise 41 we see that using any of the 'Unmatched' production rules causes infinite recursion. This means that the grammar in exercise 41 cannot produce the string "IF e THEN other" while the grammar from exercise 40 can.

42. Repeat Exercise 41, adding the production Unmatched→other to the grammar.

```
1  Start      → Stmt $
2  Stmt       → Matched
3             | Unmatched
4  Matched    → if e then Matched else Matched
5             | other
6  Unmatched  → if e then Matched else Unmatched
7             | if e then Unmatched
8             | other
```

(a) Explain why the grammar is or is not LALR(1)

Adding the new production to the grammar we now get an error. CUP is telling us that we get a reduce reduce conflict, because of the terminal 'OTHER', meaning the grammar is no longer LALR(1)



```
spo@SPOVirtualMachine:~/Desktop/Lecture 8 - Cup and SableCC$ cup grammar.cup
Warning : *** Reduce/Reduce conflict found in state #3
  between matched ::= OTHER (*)
  and     unmatched ::= OTHER (*)
  under symbols: {EOF}
  Resolved in favor of the first production.

Warning : *** Production "unmatched ::= OTHER " never reduced
Error : *** More conflicts encountered than expected -- parser generation aborte
d
```

(b) Is the language of this grammar the same as the language of the grammar in Exercise 40? Why or why not?

Again, the tool cannot help us determine the equality of the languages described by the grammars. Looking at the grammars, it intuitively seems like they are generating the same strings.

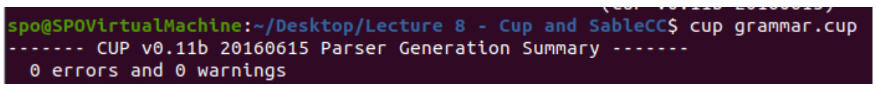
43. Consider the following grammar:

```
1  Start      → Stmt $
2  Stmt       → Matched
3               | Unmatched
4  Matched    → if e then Matched else Matched
5               | other
6  Unmatched  → if e then Matched else Unmatched
7               | if e then Stmt
```

(a) Explain why the grammar is or is not LALR(1)
Creating the grammar.cup file:

```
terminal IF, THEN, ELSE, E, OTHER;

non terminal program, stmt, matched, unmatched;

program ::= stmt;
stmt    ::= matched | unmatched;

matched   ::= IF E THEN matched ELSE matched
            | OTHER;
unmatched ::= IF E THEN matched ELSE unmatched
            | IF E THEN stmt;
```

Running cup on it we get a successfully generated parser:



```
spo@SPOVirtualMachine:~/Desktop/Lecture 8 - Cup and SableCC$ cup grammar.cup
------- CUP v0.11b 20160615 Parser Generation Summary -------
  0 errors and 0 warnings
```

...meaning the grammar is LALR(1).

(b) Is the language of this grammar the same as the language of the grammar in Exercise 40? Why or why not?
Again, the tool cannot help us determine the equality of the languages described by the grammars. Looking at the grammars, it intuitively seems like they are generating the same strings.

3. Familiarize yourself with SableCC by browsing their web pages
   Visit here: https://sablecc.org
   A virtual machine with SableCC installed can be found on moodle along with a guide for running it.

4. Use **SableCC** to do Fischer et al exercises 40, 41, 42, 43 on pages 224-233 (exercise 40, 42, 44, 43 on pages 263-264)
   The virtual machine available on Moodle has CUP and SableCC installed, along with a pdf guide that details how to use them!

40. Recall the dangling else problem introduced in Chapter 5. A grammar for a simplified language that allows conditional statements follows:

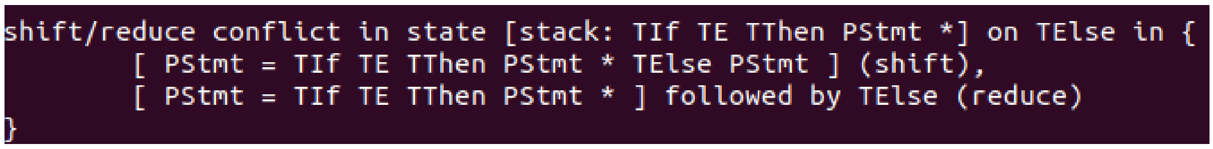
```
1  Start → Stmt $
2  Stmt → if e then Stmt else Stmt
3         | if e then Stmt
4         | other
```

Explain why the grammar is or is not LALR(1).

We create the following SableCC grammar:

```
Helpers
  tab   = 9;

Tokens
  e   = 'exp';
  other = 'other';
  if = 'if';
  then = 'then';
  else = 'else';
  blank = ' ' | tab;

Ignored Tokens
  blank, eol;

Productions
start = stmt;
stmt = {one}   if e then [fst]:stmt else [snd]:stmt
     | {two}   if e then stmt
     | {three} other;
```

When trying to generate a parser with SableCC we get:



```
shift/reduce conflict in state [stack: TIf TE TThen PStmt *] on TElse in {
        [ PStmt = TIf TE TThen PStmt * TElse PStmt ] (shift),
        [ PStmt = TIf TE TThen PStmt * ] followed by TElse (reduce)
}
```

Meaning we have shift/reduce conflict and the grammar is not LALR(1).

41. Consider the following grammar

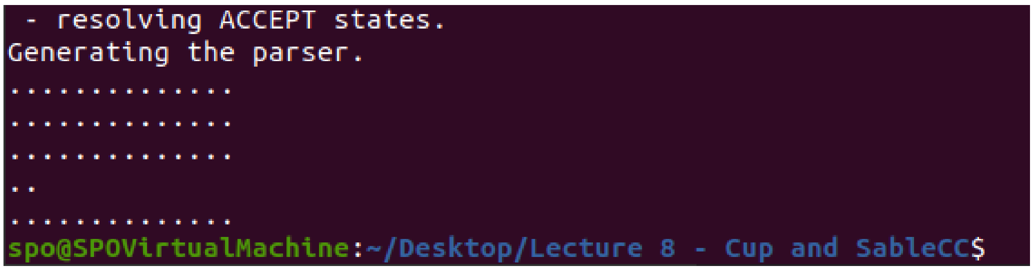
```
1  Start     → Stmt $
2  Stmt      → Matched
3            | Unmatched
4  Matched   → if e then Matched else Matched
5            | other
6  Unmatched → if e then Matched else Unmatched
7            | if e then Unmatched
```

(a) Explain why the grammar is or is not LALR(1)

We use a slightly modified version of the previous grammar:

```
...
Productions
start = stmt;
stmt = {one} matched
     | {two} unmatched;

matched = {one} if e then [first]:matched else [second]:matched
        | {two} other;

unmatched = {one} if e then matched else unmatched
          | {two} if e then unmatched;
```

and run SableCC on it:



```
 - resolving ACCEPT states.
Generating the parser.
..............
..............
..............
..
..............
spo@SPOVirtualMachine:~/Desktop/Lecture 8 - Cup and SableCC$
```

No errors, we now have a LALR(1) parser based on the grammar.

(b) Is the language of this grammar the same as the language of the grammar in Exercise 40? Why or why not?

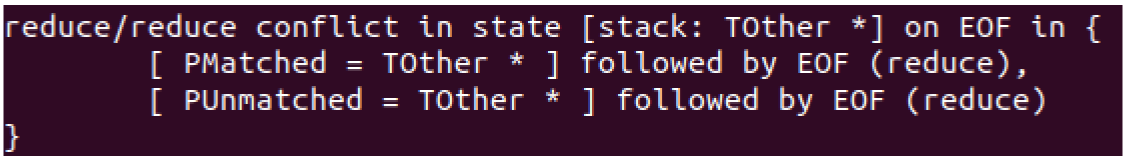
See the answer to the Cup version of this exercise (found earlier in this document).

42. Repeat Exercise 41, adding the production Unmatched→other to the grammar.

```
1  Start     → Stmt $
2  Stmt      → Matched
3            | Unmatched
4  Matched   → if e then Matched else Matched
5            | other
6  Unmatched → if e then Matched else Unmatched
7            | if e then Unmatched
8            | other
```

(a) Explain why the grammar is or is not LALR(1)

Again using the same grammar file but adding the ‘other’ rule and then running SableCC on it we get:



```
reduce/reduce conflict in state [stack: TOther *] on EOF in {
        [ PMatched = TOther * ] followed by EOF (reduce),
        [ PUnmatched = TOther * ] followed by EOF (reduce)
}
```

A reduce reduce conflict, meaning not LALR(1)

(b) Is the language of this grammar the same as the language of the grammar in Exercise 40? Why or why not?

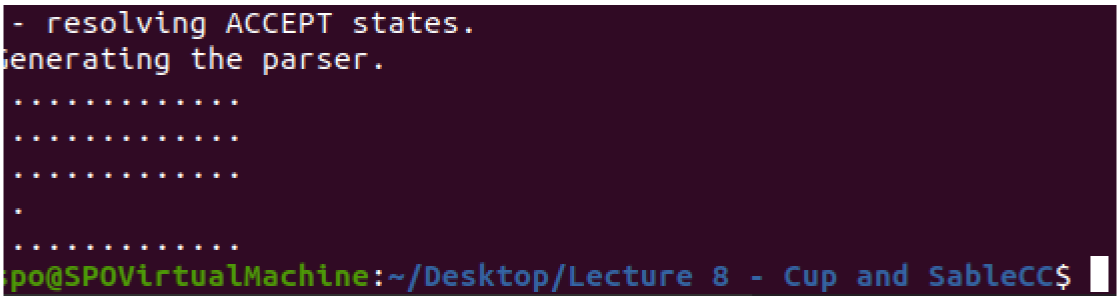
Same answer as the Cup version of this exercise (found earlier in this document).

43. Consider the following grammar:

1	Start	→	Stmt $
2	Stmt	→	Matched
3		|	Unmatched
4	Matched	→	if e then Matched else Matched
5		|	other
6	Unmatched	→	if e then Matched else Unmatched
7		|	if e then Stmt

(a) Explain why the grammar is or is not LALR(1)

Again we modify the previous grammar slightly by replacing ‘unmatched’ with stmt in the last rule. We run SableCC on it and get:



```
 - resolving ACCEPT states.
enerating the parser.
 .............
 .............
 .............
 .
 .............
po@SPOVirtualMachine:~/Desktop/Lecture 8 - Cup and SableCC$
```

The grammar is LALR(1)

(b) Is the language of this grammar the same as the language of the grammar in Exercise 40? Why or why not?

Same answer as the Cup version of this exercise (found earlier in this document).

5. Use The Context Free Grammar Checker on http://smlweb.cpsc.ucalgary.ca/ or its newest version on http://mdaines.github.io/grammophone/ to do Fischer et al exercises 40, 41, 42, 43 on pages 224-233 (exercise 40, 42, 44, 43 on pages 263-264)

   40. Recall the dangling else problem introduced in Chapter 5. A grammar for a simplified language that allows conditional statements follows:

```
1  Start → Stmt $
2  Stmt  → if e then Stmt else Stmt
3        | if e then Stmt
4        | other
```

   Explain why the grammar is or is not LALR(1).
   First enter the grammar into the online grammar checker on http://smlweb.cpsc.ucalgary.ca/ :

```
S -> Stmt.
Stmt -> if e then Stmt else Stmt
   | if e then Stmt
   | other.
```

   Next click vital statistics. This brings up a screen saying that the grammar is not LL(1). To see if the grammar is LALR(1) click on 'generate LALR(1) automaton'. Here we see that a shift/reduce conflict is encountered (dangling else), and for this reason the grammar is not LALR(1).

   41. Consider the following grammar

```
1  Start     → Stmt $
2  Stmt      → Matched
3            | Unmatched
4  Matched   → if e then Matched else Matched
5            | other
6  Unmatched → if e then Matched else Unmatched
7            | if e then Unmatched
```

   (a) Explain why the grammar is or is not LALR(1)
   We enter the grammar into the grammar checker:

```
S -> Matched
   | Unmatched.
Matched -> if e then Matched else Matched
   | other.
Unmatched -> if e then Matched else Unmatched
   | if e then Unmatched.
```

   This shows a screen stating that the grammar is not LL(1), but clicking on 'generate LALR(1) automaton' we see that it is in fact able to generate a LALR(1) parse table for this grammar.

(b) Is the language of this grammar the same as the language of the grammar in Exercise 40? Why or why not?
Same answer as the Cup version of this exercise (found earlier in this document).

42. Repeat Exercise 41, adding the production Unmatched→other to the grammar.

1	Start	→	Stmt $
2	Stmt	→	Matched
3		|	Unmatched
4	Matched	→	if e then Matched else Matched
5		|	other
6	Unmatched	→	if e then Matched else Unmatched
7		|	if e then Unmatched
8		|	other

(a) Explain why the grammar is or is not LALR(1)
Add the ‘other’ rule to the previous grammar.
Clicking on generate LALR(1) automaton we now get a warning that we have a reduce/reduce conflict, meaning the grammar is no longer LALR(1).

(b) Is the language of this grammar the same as the language of the grammar in Exercise 40? Why or why not?
See answer for exercise 42.b from the CUP tool.

43. Consider the following grammar:

1	Start	→	Stmt $
2	Stmt	→	Matched
3		|	Unmatched
4	Matched	→	if e then Matched else Matched
5		|	other
6	Unmatched	→	if e then Matched else Unmatched
7		|	if e then Stmt

(a) Explain why the grammar is or is not LALR(1)
We enter the grammar into the online grammar checker:

```
S -> Smt.
Stmt -> Matched
    | Unmatched.
Matched -> if e then Matched else Matched
    | other.
Unmatched -> if e then Matched else Unmatched
    | if e then Stmt.
```

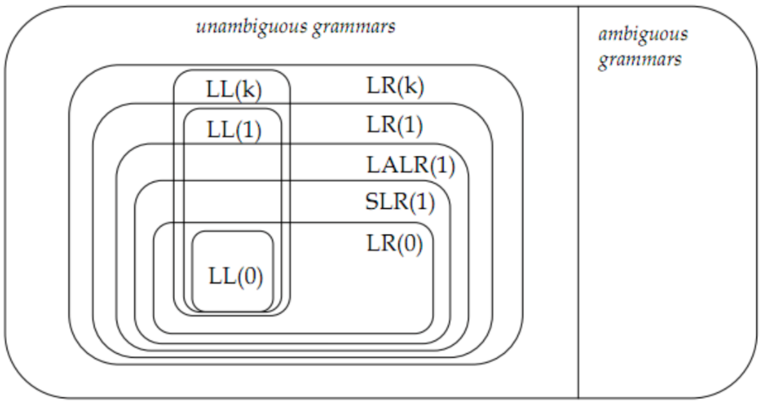
Clicking on ‘generate LALR(1)’ we get a parse table and a message showing that the grammar is LALR(1).
(b) Is the language of this grammar the same as the language of the grammar in Exercise 40? Why or why not?
Same answer as the Cup version of this exercise (found earlier in this document).

Group Exercises - Lecture 08

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Discuss pro. And cons. of LR parsing (e.g. as compared to LL parsing)
   Have a look at this map. An LR(k) parser can parse all languages from the LL parser class, while this is not the case the other way around. The LR parsers can, however, be more difficult to implement, so if the language is within for example the LL(1) realm, it may be a good idea to create an LL(1) parser.

   LL(1) versus LR(k)
   A picture is worth a thousand words:



3. Discuss the pros. and cons. of CUP vs SableCC
   Looking back at the exercises: It was a lot easier to formulate the simple grammars in Cup. A lot of small details were required to correctly specify the grammars in SableCC. For example a nonterminal appearing twice on the rhs of a rule requires both occurrences to be named:

   ```
   stmt = if e then stmt else stmt  // Will not compile due stmt occuring twice
   ```

   vs.

   ```
   stmt = if e then [one]:stmt else [two]:stmt   // Will compile
   ```

   Generally the learning curve seems to be steeper for SableCC compared to Cup. The added complexity does however have some benefits. The previous example allows us to access subnodes by their name instead of their position.

4. Follow the Lab associated with Crafting a Compiler Chapter 6
   http://www.cs.wustl.edu/~cytron/cacweb/Chapters/6/lab.shtml
   Find them inside Eclipse in the virtual machine provided on moodle..

Individual Exercises - Lecture 9

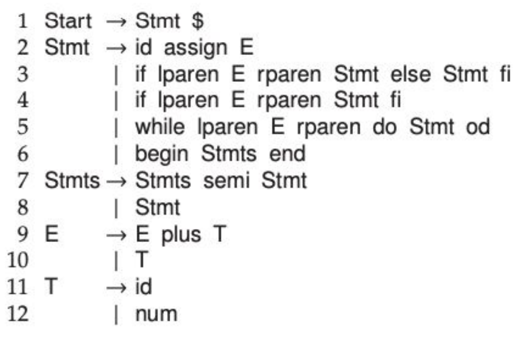
1. (optional - but recommended) Follow the studio associated with Crafting a Compiler Chapter 7
   http://www.cs.wustl.edu/~cytron/cacweb/Chapters/7/studio.shtml
   A virtual machine is available on Moodle with the studio/lab exercises loaded inside Eclipse.

2. (optional - by recommended) Do the exercises associated with the Tutorial on the Visitor Pattern
   Can be found here: https://www.cse.wustl.edu/~cytron/cacweb/Tutorial/Visitor/

3. *Do Fischer et al exercise 10, 12, 22 on pages 272-278 (exercise 11, 15, 22 on pages 304-310 in GE)*

   10. *Consider extending the grammar in Figure 7.14 to include binary subtraction and unary negation, so that the expression "minus y minus x times 3"*
   has the effect of negating y prior to subtraction of the product of x and 3.

   (a) Following the steps outlined in Section 7.4, modify the grammar to include these new operators. Your grammar should grant negation strongest precedence, at a level equivalent to deref. Binary subtraction can appear at the same level as binary addition.

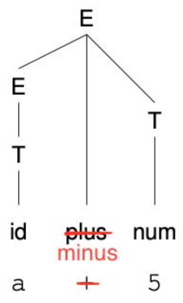


```
 1  Start  → Stmt $
 2  Stmt   → id assign E
 3         | if lparen E rparen Stmt else Stmt fi
 4         | if lparen E rparen Stmt fi
 5         | while lparen E rparen do Stmt od
 6         | begin Stmts end
 7  Stmts  → Stmts semi Stmt
 8         | Stmt
 9  E      → E plus T
10         | T
11  T      → id
12         | num
```

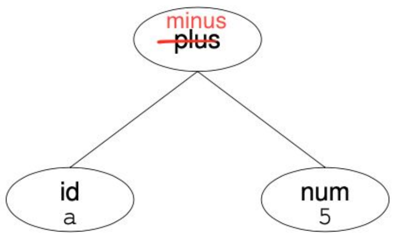
Figure 7.14: Grammar for a simple language.

Alter the modification:
9 E → E plus T
10 | E minus T
11 | T
12 T → id
13 | num
15 | minus num
16 | *minus id*

(b) Modify the chapter's AST design to include subtraction and negation by including a minus operator node.



(a)



(b)

For negation you could introduce a negativeNum node : num(5) -> negativeNum(5),

Or you could keep the current num node and allow for negative numbers: num(-5).

(c) Install the appropriate semantic actions into the parser.

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.

To E$_{result}$ add:

| E_{e1} minus T_{e2}

Result <- MakeFamily(minus, e1, e2)

// This create a MinusNode with children e1 and e2

To T_{result} add:

| minus num_{val}

Result <- MakeNode(-val)

12. *The grammar below generates nested lists of numbers. The semantic actions are intended to count the number of elements just inside each parenthesized list. For each list found by Rule 2 (Marker 44), the number of elements found just inside the list.*
For example, the input
((1 2 3) (1 2 3 4 5 6))
Should print 3, 6 and 2

1	Start	$\rightarrow$ List_{avg} \$	
2	List_{result}	$\rightarrow$ lparen Operands_{ops} rparen	
		PRINT(*count*)	(44)
3		| num_{val}	
4	Operands	$\rightarrow$ Operands List	
		$count \leftarrow count + 1$	
5		| List	
		$count \leftarrow 1$	

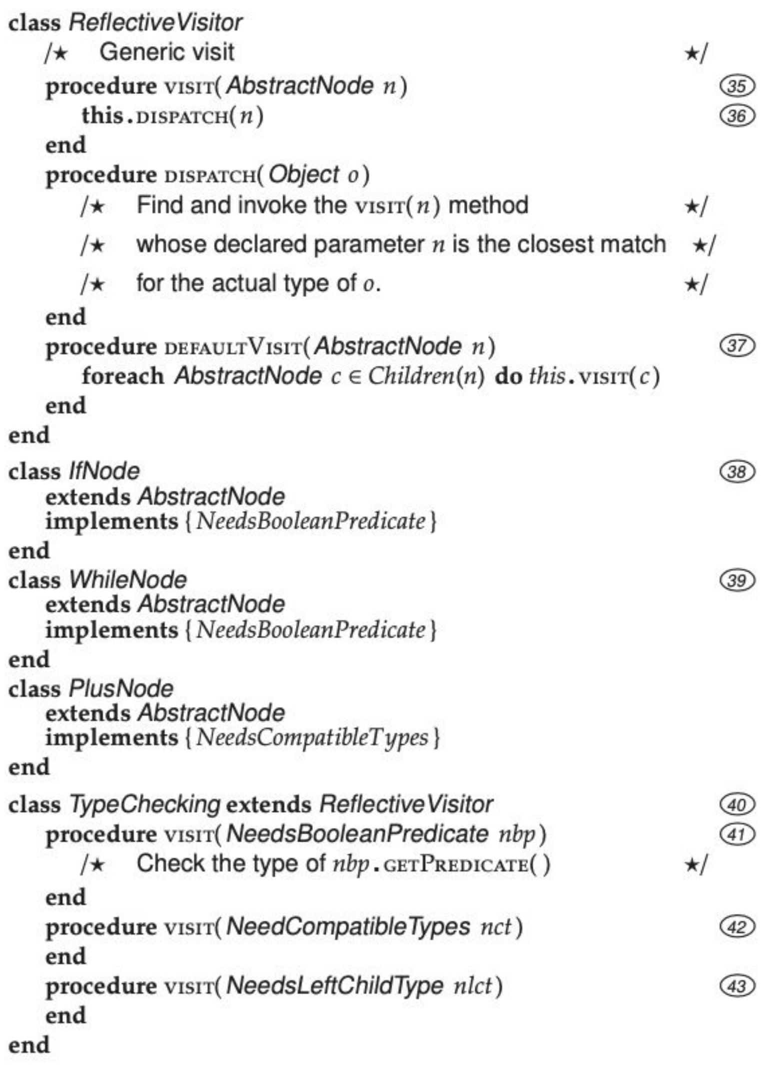
(a) The grammar uses a global variable count to determine the number of elements in a list. What is wrong with that approach?

Relying on a single global variable for counting is a problem because it cannot handle nested lists, since it overwrites the current element count when entering a new list.

(b) Change the semantic actions so that the appropriate values are synthesized by *the rules to allow counting without a global variable.*
Instead of using a global variable, use synthesized attributes. Remember, a synthesized attribute is a value associated with some node in the tree, where the value is calculated from the children of said node. See Fischer figure 7.3 in section 7.2.1 (p. 240).

```
Start                → $list_{avg}$ $

$List_{result}$      → lparen $Operands_{ops}$ rparen
                         Print(ops);
                     | numval

$Operands_{count}$   → $Operands_{previous}$ List
                         count ← previous + 1;
                     | List
                         count ← 1
```

22. In addition to the ***NeedsBooleanPredicate*** type discussed in Section 7.7.3, Figure 7.24 references the following types: ***NeedCompatibleTypes*** and ***NeedsLeftChildType***.



```
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

(a) Which node types inherit from those types?

NeedCompatibleType probably requires all children to have the same type, and NeedComptaibleType could be inherited by operation nodes such as +, -, / etc. For example addition nodes may allow addition of a float and an integer.

NeedsLeftChildType probably requires that the children has the same type as the leftmost child. This might be inherited by assignment nodes, since most languages require the right hand side to be of the same type (or at least castable to the same type), as the left hand side of the assignment.

(b) Describe the actions that should be performed by the visitor at Markers 42 and 43 on behalf of the ***NeedCompatibleTypes*** *and* ***NeedsLeftChildType*** *types.*

NeedCompatibleType

Find the set of types for all immediate children of the node and check the list of compatible types for that node type. For example, addition operations might allow adding doubles and floats, but not doubles and integers.

NeedsLeftChildType

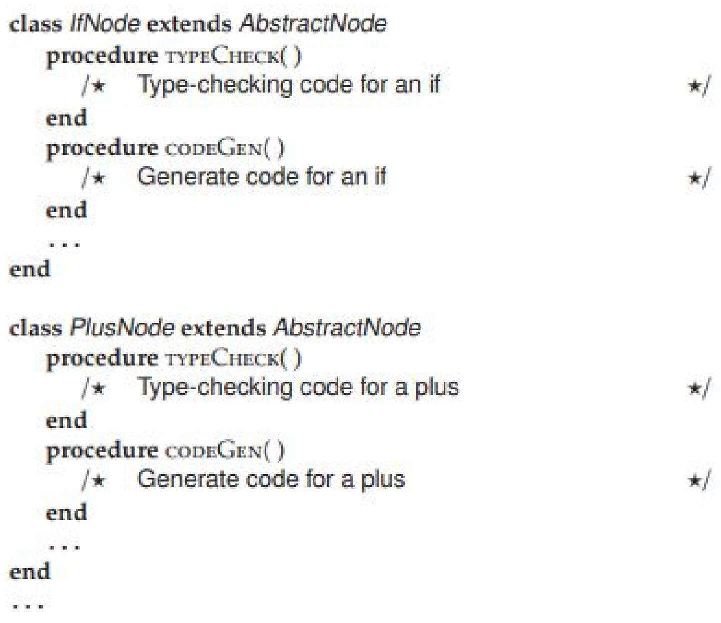
Declare local variable Type = LeftMostChild.getType()
Check all the other children have the same (or convertible) types.

Group Exercises - Lecture 9

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Do Fischer et al exercise 20, 19, 21 on page 272-278 (exercise 14, 19 20 on pages 304-310 in GE) (you may use the language you are defining in your project when designing ast nodes in exercise 19)

   20. In contrast to the approach discussed in Section 7.7.2, Figure 7.25 shows the partial results of a design in which each phase contributes code to each node type.



```
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.

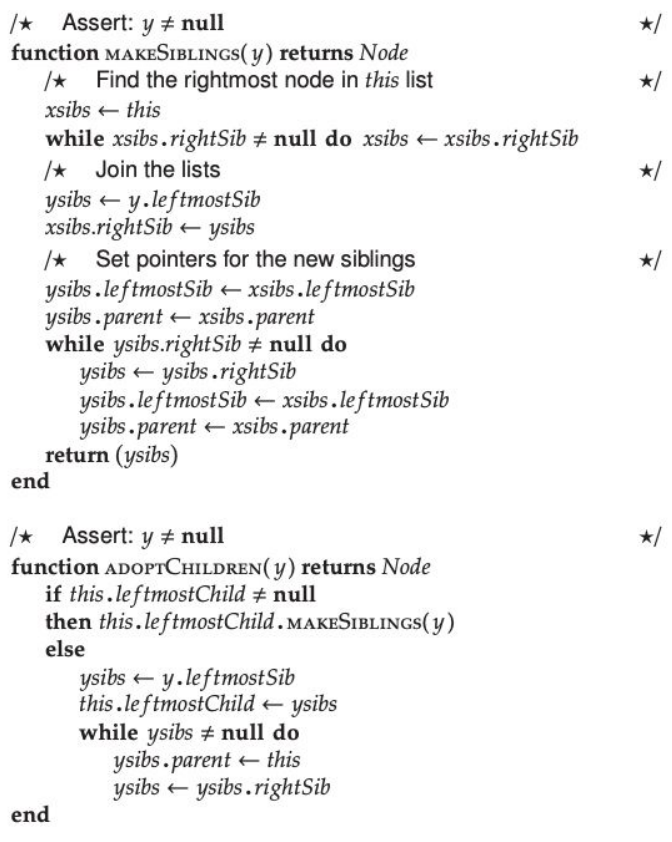
   (a) What are the advantages and disadvantages of the approach used in Figure 7.25?
   This approach might be more manageable initially when the amount of phases is low. Inside each node class you can easily follow what happens to that node type during the different compiler phases.

   A problem with this approach is that many modern compilers have up to a couple of hundred phases(mostly optimization steps). If the node classes themselves contained the methods for all phases, they would very quickly get very bloated. Consider software design principles such as the 'Single responsibility principle'. If you use the approach in 7.25, each Node class will end up being responsible for many different steps of the compiler. If you change a phase, or add a new one, you now have to go through all the node classes to implement this change.

(b) How does the visitor pattern discussed in Section 7.7.2 address the disadvantages?

A separate class for each phase is created. This class then contains visitor methods for all types of nodes. This makes modifying and adding of phases much simpler. If we modify a phase, we only have to modify the visitor for that phase. If we add a phase, we simply add a new visitor and then dispatch it to the root.

19. Based on the discussion of Section 7.4 and using the pseudocode in Figure 7.13 as a guide, design a set of AST classes and methods to support AST construction in a real programming language.



```
/*  Assert: $y \neq$ null                                          */
function makeSiblings( $y$ ) returns Node
    /*  Find the rightmost node in this list                      */
    $xsibs \leftarrow this$
    while $xsibs.rightSib \neq$ null do $xsibs \leftarrow xsibs.rightSib$
    /*  Join the lists                                             */
    $ysibs \leftarrow y.leftmostSib$
    $xsibs.rightSib \leftarrow ysibs$
    /*  Set pointers for the new siblings                         */
    $ysibs.leftmostSib \leftarrow xsibs.leftmostSib$
    $ysibs.parent \leftarrow xsibs.parent$
    while $ysibs.rightSib \neq$ null do
        $ysibs \leftarrow ysibs.rightSib$
        $ysibs.leftmostSib \leftarrow xsibs.leftmostSib$
        $ysibs.parent \leftarrow xsibs.parent$
    return ($ysibs$)
end

/*  Assert: $y \neq$ null                                          */
function adoptChildren( $y$ ) returns Node
    if $this.leftmostChild \neq$ null
    then $this.leftmostChild$.makeSiblings( $y$ )
    else
        $ysibs \leftarrow y.leftmostSib$
        $this.leftmostChild \leftarrow ysibs$
        while $ysibs \neq$ null do
            $ysibs.parent \leftarrow this$
            $ysibs \leftarrow ysibs.rightSib$
end
```

Figure 7.13: Methods for building an AST.

Below is an example for an if statement very similar to the one in Java / C# / C. In the first code snippet the AST node for the if statement can be seen. In the second code snippet a visitor method for creating the AST node when visiting the non-terminal ifstmt node in the parse tree. The ListNode inherits from an abstract node class.

Other node classes could be created in a similar fashion. Usually an abstract class with methods like getChildren() and accept(visitor) is created as a foundation. This class can then be extended like in this case by a ListNode, but often BinaryNodes and LeafNodes are also included, since their behaviour when visited is slightly different. The IfStmtNode below is of type listnode, since it has more than 2 children.

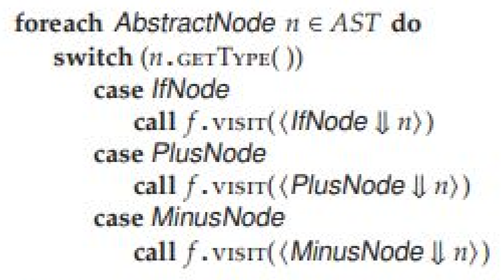
Node class for if statements

```java
public class IfStmtNode extends ListNode {
   public IfStmtNode(ArrayList<Node> children) {
       super(children);
   }
   public Node getLogicalExprNode(){
       return super.getChildren().get(0);
   }
   public Node getIfBlock(){
       return super.getChildren().get(1);
   }
   public Node getElseBlock(){
       // No else block present
       if(super.getChildren().size() < 3){
           return null;
       } else {
           return super.getChildren().get(2);
       }
   }
   // The generics here allows different visitors to have different
   // return types. For example the ASTBuilder visitor will probably
   // return objects of the type Node(to return a new modified tree),
   // while a pretty printer visitor might return strings.
   // Don’t fret if you don’t get this 100% right now
   public <T> T accept(ASTBaseVisitor<? extends T> astBaseVisitor) {
        return astBaseVisitor.visit(this);
   }
}
```

Visitor method for IfStmt inside the ASTBuilderVisitor<Node>(which extends the BaseVisitor)

```java
@Override
public Node visitIfstmt(ifStmtParseTreeNode node) {
   ArrayList<Node> children = new ArrayList<>();
   Node logicalExpr = visit(node.logical_expr());
   // Block of code inside if stmt
   Node ifBlock = visit(node.blck);
   children.add(logicalExpr);
   children.add(ifBlock);
   if(node.elseblck != null){
       Node elseBlocK = visit(node.elseblck);
       children.add(elseBlocK);
   }

   return new IfStmtNode(node, children);
}
```

21. In contrast with the approaches discussed in Sections 7.7.2 and 7.7.3, consider the idea sketched in Figure 7.26.



```
foreach AbstractNode $n \in AST$ do
    switch ($n$.GETTYPE( ))
        case IfNode
            call $f$.VISIT($\langle$IfNode $\Downarrow n\rangle$)
        case PlusNode
            call $f$.VISIT($\langle$PlusNode $\Downarrow n\rangle$)
        case MinusNode
            call $f$.VISIT($\langle$MinusNode $\Downarrow n\rangle$)
```

Figure 7.26: An alternative for achieving double dispatch.

(a) What are the advantages and disadvantages of the approach used in Figure 7.26?
The advantage is that it is easy to understand/implement and can give a good overview of the node types visited by the specific visitor.
The disadvantage is that it is essentially a lot of duplicated code and it can quickly become very big with 50+ node types. Also one node type can easily be forgotten in the switch case.
(b) How does the visitor pattern discussed in Sections 7.7.2 and 7.7.3 address the disadvantages?
Instead of switching on the type, each node has an accept function that takes the visitor and makes the visitor visit "this" (i.e. the node itself and it's children), which would invoke a visit method specific to that type. This eliminates the large switch statement, but still contains a lot of duplicated code (you have to have an accept() method in every node class).

The reflective visitor pattern eliminates the need for the accept() method through the dispatch() method. It looks through the available node classes and finds the "closest match".

3. (optional - but recommended) Follow the Lab associated with Crafting a Compiler Chapter 7
http://www.cs.wustl.edu/~cytron/cacweb/Chapters/7/lab.shtml
For this a virtual machine will be provided with the necessary tools pre installed.

Individual Exercises - Lecture 10

1. (optional - but recommended) Follow the studio associated with Crafting a Compiler Chapter 8 http://www.cs.wustl.edu/~cytron/cacweb/Chapters/8/studio.shtml
   For this a virtual machine will be provided with the necessary tools pre installed.

2. Do Fischer et al exercise 14 and 16 on pages 336-341 (exercise 15 and 17 on pages 370-373 in GE)
   14. Consider the following C program:

```
int func() {
   int x, y;
   x = 10;
   {
       int x;
       x = 20;
       y = x;
   }
   return(x * y);
}
```

   The identifier x is declared in the method's outer scope. Another declaration occurs within the nested scope that assigns y. In C, the nested declaration of x could be regarded as a declaration of some other name, provided that the inner scope's reference to is appropriately renamed x. Design a set of AST visitor methods that

   (a) Renames and moves nested variable declarations to the method's outermost scope
   You could assign a unique name to every scope.
   Then, for every variable declaration node you visit, prefix the variable name with the scope name. For example `int x = 1;` inside scope4 becomes `int scope4_x = 1;`
   Save this alias in a dictionary structure. Then move the declaration node to the outermost method scope.
   When your visitor exits out of a scope, remove all aliases that were created inside the scope.
   For every id node you visit, replace it by the most recent alias for that variable name.
   Example :

```
Int func() { // scope1
      int x, y;  // becomes scope1_x and scope1_y;
      x = 10; // scope1_x;
      { // scope 2
            x *= 2; // scope1_x
            int x; // scope2_x;
            x = 20; // scope2_x
            y = x; // scope1_y and scope2_x
      }
      return(x * y); // scope1_x, scope1_y
```

The following is a very extensive pseudo code version of the solution that was just proposed. It is okay if your solution is just an outline of a design.

In this approach we extend the symbol table to keep track of scope names. Each scope has its own dictionary to keep track of renamed variables.When we enter a new scope, it inherits all previous renamings. If a symbol is added that already exists in the dictionary, it is just overwritten with the new name:

```
public class SymbolTable {
  Stack<Scope> scopeStack = ...;
  private class Scope(String name, Dict<String, String> renamings){...}

  // Push new scope to stack passing on all renamings from current scope
  // Generate a new unique name for the new scope
  void enterScope() {...}
  // Pop scope from stack
  void exitScope() {...}

  // Add renaming to dictionary of current scope
  void addSymbol(String variableName) {
    String newName = scopeStack.peek().name + variableName;
    scopeStack.peek().renamings.put(variableName, newName);
  }
  // Look up variableName in current scope renaming dictionary
  String getSymbol(variableName){...}
}
```

We can now create a visitor for moving and renaming declarations according to the scope they are declared in:

```
public class ScopeLabelVisitor extends ASTVisitor {
  private SymbolTable symbolTable = new SymbolTable();
  private Node currentMethodScope;

  public void visit(MethodNode funcNode){
      currentMethodScope = funcNode;
      symbolTable.enterScope();
      // ... visit children
      // (Here we may visit a scopeNode)
      symbolTable.exitScope();
  };


// Continues on the next page!
```

```
   public void visit(ScopeNode scopeNode){
       symbolTable.enterScope();
       for (Node child : scopeNode.getChildren()) {
         child.accept(this); // Visit child
         // (The child may be of the type VarDeclNode)

         // Move renamed declaration to method node
         if (child instanceof VarDeclNode) {
             scopeNode.remove(child);
             currentMethodScope.insertChild(child);
         }
       }
       symbolTable.exitScope();
   };

   public void visit(VarDeclNode decl){
       symbolTable.addSymbol(decl.name);
       ...
   };
   ...
}
```

(b) Appropriately renames symbol references to preserve the meaning of the program

We can add the following method to the previous visitor:

```
   public void visit(VariableIdNode idNode){
       idNode.name = symbolTable.getRename(idNode.name);
   }
```

16. As mentioned in Section 8.4.2, C allows the same identifier to appear as a struct name, a label, and an ordinary variable name. Thus, the following is a valid C program:

```
main() {
    struct xxx {
        int a,b;
    } c;
    int xxx;

    xxx:
        c.a = 1;
}
```

In C, the structure name never appears without the terminal struct preceding it. The label can only be used as the target of a goto statement. Explain how to use the symbol table interface given in Section 8.1.2 to allow all three varieties of xxx to coexist in the same scope.

Section 8.1.2 Interface:

- **openScope()** opens a new scope in the symbol table. New symbols are en- tered in the resulting scope.

- **closeScope()** closes the most recently opened scope in the symbol table. Symbol references subsequently revert to outer scopes.

- **enterSymbol(name,type)** enters name in the symbol table's current scope. The parameter type conveys the data type and access attributes of name's declaration.

- **retrieveSymbol(name)** returns the symbol table's currently valid declaration for name. If no declaration for name is currently in effect, then a null pointer is returned.

- **declaredLocally(name)** tests whether name is present in the symbol table's current (innermost) scope. If it is, true is returned. If name is in an outer scope, or is not in the symbol table at all, false is returned.

If we want to preserve the current symbol table interface without changes, we could prefix every symbol name with its corresponding type. So to add the struct xxx to the symbol_table we would call enterSymbol("struct_xxx", "struct"). When retrieving the struct we would then call retrieveSymbol("struct_xxx). This way we can differentiate between symbols that have the same name, but different types.

Alternatively we can modify the interface in one of the following ways:
1. Since the struct and labels are semantically completely different to variable declarations, it would make sense to extend the symbol table with the following methods:

    retrieveStruct(name), which returns the struct for the given name or null
    retrieveLabel(name), which returns the label if it present, else null

Determining which of the two methods to call can be done based on the type of node the visitor finds. If the node is of type Struct, use retrieveStruct etc. The symbol table would then have to store all three types in a way that it can distinguish them from each other.

2. Another possible way to allow for the three different types in the symbol table is to simply add an extra column in the symbol table that holds the type of a declaration. In addition to this, the symbol table method for retrieving a symbol should then be modified to the following:

retrieveSymbol(name, type)

This method should return a valid declaration with the given name and type, if one exists.
E.g the struct declaration xxx would have a type parameter of struct, integer declaration of type integer and label declaration of type label. Thus when accessing the identifer xxx as a certain type, all the xxx identifiers will be distinguishable.

Group Exercises - Lecture 10

3. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

4. Do Fischer et al exercise 1, 2, 15 on pages 336-341 (exercise 3, 4, 14 on pages 370-373 in GE)

   1. The two data structures most commonly used to implement symbol tables in production compilers are binary search trees and hash tables. What are the advantages and disadvantages of using each of these data structures for symbol tables?
   Binary search tree:
   - Advantages:
     - Compact
     - Fast search times(O(log n))
   - Disadvantages:
     - Constant need to rebalance the tree

   Hash table:

   - Advantages:
     - Constant time lookup
   - Disadvantages:
     - Can be less memory efficient
     - Overkill for very small tables

2. Consider a program in which the variable is declared as a method’s parameter and as one of the method’s local variables. A programming language includes parameter hiding if the local variable’s declaration can mask the parameter’s declaration. Otherwise, the situation described in this exercise results in a multiply defined symbol. With regard to the symbol table interface presented in Section 8.1.2, explain the implicit scope-changing actions that must be taken if the language calls for

(a) Parameter hiding
(b) No parameter hiding

Section 8.1.2 Interface:

- **openScope()** opens a new scope in the symbol table. New symbols are en- tered in the resulting scope.

- **closeScope()** closes the most recently opened scope in the symbol table. Symbol references subsequently revert to outer scopes.

- **enterSymbol(name,type)** enters name in the symbol table’s current scope. The parameter type conveys the data type and access attributes of name’s declaration.

- **retrieveSymbol(name)** returns the symbol table's currently valid declaration for name. If no declaration for name is currently in effect, then a null pointer is returned.

- **declaredLocally(name)** tests whether name is present in the symbol table’s current (innermost) scope. If it is, true is returned. If name is in an outer scope, or is not in the symbol table at all, false is returned.

a) Parameter hiding

   We create two scopes for the method. The first scope is for the method parameters. All parameters symbols will go inside this scope. Inside this scope we open a new scope for the method body. The locally declared variable within the method body will be entered into this scope. When looking up the variable in the scope, we first check our immediate scope, and then the parent scope. If the variable is declared inside the method body scope as well as in the parameter scope, we will find the one in the method body scope first and hence shadow the one from the parameter list.

b) No parameter hiding

   We create one common scope for both the parameter symbols and the symbols in the method body. If a parameter and a variable declared in the method has the same name, the symbol table would identify this as an error.

15. Using the symbol table interface given in Section 8.1.2, describe how to implement structures (structs in C) under each of the following assumptions:

• All structures and fields are entered in a single symbol table.

• A structure is represented by its own symbol table, whose contents are the structure's subfields.

Section 8.1.2 Interface:

- **openScope()** opens a new scope in the symbol table. New symbols are en- tered in the resulting scope.

- **closeScope()** closes the most recently opened scope in the symbol table. Symbol references subsequently revert to outer scopes.

- **enterSymbol(name,type)** enters name in the symbol table's current scope. The parameter type conveys the data type and access attributes of name's declaration.

- **retrieveSymbol(name)** returns the symbol table's currently valid declaration for name. If no declaration for name is currently in effect, then a null pointer is returned.

- **declaredLocally(name)** tests whether name is present in the symbol table's current (innermost) scope. If it is, true is returned. If name is in an outer scope, or is not in the symbol table at all, false is returned.

We will use the same solutions used in Section 8.2.2 on page 285 to handle block-structured symbol tables. If all structures are to share the same symbol table, we assign each structure a unique *scope number*. Scope numbers should be distinguishable from the numbers used for nested scopes. Very large integers or perhaps negative values might be used. The scope number assigned is stored with the type information for a structure. When a reference like a.b is processed, a is first looked up like an ordinary identifier.

We verify that a is a structure, with scope number *s*. When field b is looked up, *s* is used to restrict lookup to the fields of the appropriate structure. When the scope of a structure is closed, its fields *are not* purged from the symbol table since later qualified lookups of its fields are possible.

If fields are given their own symbol table, lookup is easier still. When processing a structure, a new symbol table is created and populated with all the fields declared within the structure. A pointer to this table is stored with the type information for the structure. When a reference like a.b is processed, a is first looked up like an ordinary identifier. We verify that a is a structure, with its fields contained in symbol table *st*. Then field b is looked up in *st*.

5. (optional - but recommended) Follow the Lab associated with Crafting a Compiler Chapter 8 http://www.cs.wustl.edu/~cytron/cacweb/Chapters/8/lab.shtml
For this a virtual machine will be provided with the necessary tools pre installed.

6. Do exercise 3 in Chapter 5 of Sebesta on page 252.

3. Write a simple assignment statement with one arithmetic operator in some language you know. For each component of the statement, list the various bindings that are required to determine the semantics when the statement is executed. For each binding, indicate the binding time used for the language.

You may consider the following Java code:

```java
public static int increment(int x) {
    return x + 1;
}
```

What is the binding time of
– Value of argument x?
– Set of values of argument x?
– Type of argument x?
– Set of types of argument x?
– Properties of operator +?

Value of argument x: Run Time

Set of values of argument x: Language design time (Note that for languages like c/c++ the size of primitive values can vary depending on the machine you are running it on, so in that case it would be bound at language implementation time)

Type of argument x: Compile Time

Set of types of argument x: Language Design Time. We can for example design the language to accept float values into int parameters by implicitly casting them.

Properties of operator +: Compile Time (Not possible to dynamically overload operators in Java)

7. What is the scope of the various x's in the following C program (with static scoping)?

```c
// Static scoping example
#include <stdio.h>
int x;
void Proc1(){
   int x;
   x = 3;
   printf("In Proc1: x = %d\n", x);
}

void Proc2(){
   x = 9;
   printf("In Proc2, x = %d\n", x);
}

int main(){
   x = 1;
   printf("Before Proc1, x = %d\n", x);
   Proc1();
   printf("After Proc1, x = %d\n", x);
   Proc2();
   printf("After Proc2, x = %d\n", x);
   return 0;
}
```

Output:
```
Before Proc1, x = 1
In Proc1, x = 3
After Proc1, x = 1
In Proc2, x = 9
After Proc2, x = 9
```

"In lexical scoping (and if you're the interpreter), you search in the local function (the function which is running now), then you search in the function (or scope) in which that function was defined, then you search in the function (scope) in which that function was defined, and so forth. "Lexical" here refers to text, in that you can find out what variable is being referred to by looking at the nesting of scopes in the program text."

See this link for a comparison between static and dynamic scoping:
https://wiki.c2.com/?DynamicScoping

8. What is the scope of the various x's in the following C Program (with dynamic scoping)?

```c
// Dynamic Scoping Example
#include <stdio.h>
int x;

void Proc1(){
   x = 1;
}
void Proc2(){
   int x;
   x = 2;
   printf("In Proc2 before Proc1, x = %d\n", x);
   Proc1();
   printf("In Proc2 after Proc1, x = %d\n", x);
}
int main(){
   x = 3;
   printf("Before Proc2, x = %d\n", x);
   Proc2();
   printf("After Proc2, x = %d\n", x);
   Proc1();
   printf("After Proc1, x = %d\n", x);
   return 0;
}
```

Result:
Before Proc2, x = 3
In Proc2 before Proc1, x = 2
In Proc2 after Proc1 x = 1
After proc 2, x = 3
After proc 1, x = 1

"In *dynamic* scoping you search in the local function first, then you search in the function that *called* the local function, then you search in the function that called *that* function, and so on, up the call stack. "Dynamic" refers to *change,* in that the call stack can be different every time a given function is called, and so the function might hit different variables depending on where it is called from."
See this link for a more thorough explanation:
https://wiki.c2.com/?DynamicScoping

Note: In this exercise you are supposed to imagine c has dynamic scoping. If this program is run, it will yield a different result.

Individual Exercises - Lecture 11

1. (optional - but recommended) Do the exercises associated with the Tutorial on the Visitor Pattern (if you haven't already done so)
   You can consider looking at the ANTLR example included in the virtual machine. You can expand the grammar, and investigate the visitor classes that ANTLR generates from it. Try doing a simple visitor that just prints something for every node in the tree.

2. For the little language defined in Figure 7.14 design and implement in pseudo code AST classes based on the AST structure in Figure 7.15

```
 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.

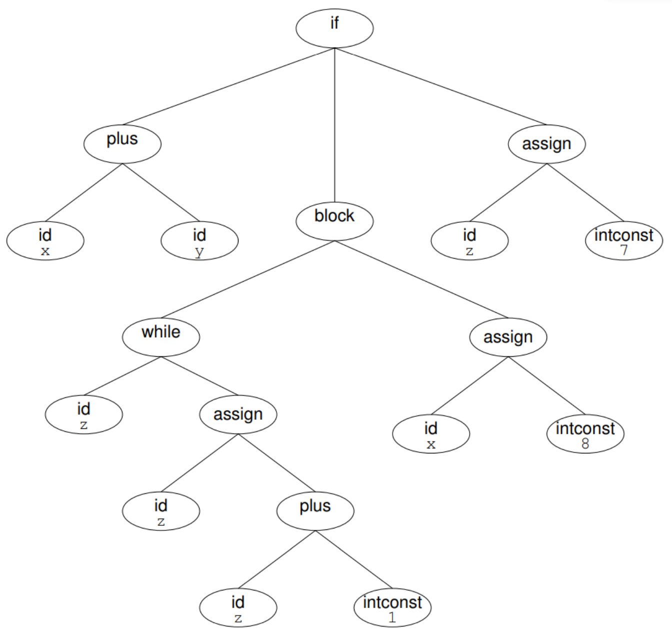
Example implementations for *AssigmentNode* and *WhileNode* in java.
Note that the classes could include an accept(); method to support use of the visitor pattern in java.

```java
class AssignmentNode implements Node{
      private VariableNode variable;
      private ExpressionNode expr;

      public AssignmentNode(VariableNode variable, ExpressionNode expr){
            this.variable = variable;
            this.expression = expr;
      }
      @Override
      List<Node> getChildren(){
            return new ArrayList<>({variable, expr})
      }
}
```

```
class WhileNode implements Node {
      Private PredicateNode predicate;
      Private LoopBodyNode loopBody;

      Public WhileNode(PredicateNode pred, LoopBodyNode loop) {
            this.predicate = pred;
            this.loopBody = loop;
      }
      @Override
      List<Node> getChildren() {
            return new ArrayList<>({predicate, loopBody});
      }
}
```

3. Construct an AST for the program in figure 7.18 (you may cheat by creating the AST based on figure 7.19)

   The program: If (x + y) { while (z) z = z + 1 od; x = 8 } else z = 7 fi $
   ('od' is just the symbol that marks the end of the while loop body in this grammar)



   Important points:

   - Structural symbols (such as '{', '(' ';' etc.) can be omitted as their meaning is represented implicitly in the structure of the tree.
   - Several nodes can often be compacted into a single node (because we do not need information about how they were derived using the grammar)
     For example a series of nested Stmt nodes may be compacted into a single block node containing as many children as there are statements.
   - You can use existing AST designs from other languages as inspiration for your own. For instance: https://astexplorer.net/

4. Add a pretty-printing method to each class in your AST structure from exercise 2 and pretty print the program from exercise 3

   One approach to pretty-printing is indenting nodes based on how deep into the tree they are nested. To print a tree in this fashion, one might pass an indentation string that increases in length for nested calls:

```
class WhileNode implements Node {
      ...
      public void prettyPrint(String indentation){
             printTabs(indentation);
             System.out.println("While Node : ")
             this.predicate.prettyPrint(indentation + "     ")
             System.out.println(); // New line
             this.loopBody.prettyPrint(indentation + "     ")
      }
}
```

5. Design a Visitor for pretty-printing for the little language defined in Figure 7.14 using the AST structure in Figure 7.15 and use it on the program from Figure 7.18

```
 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.

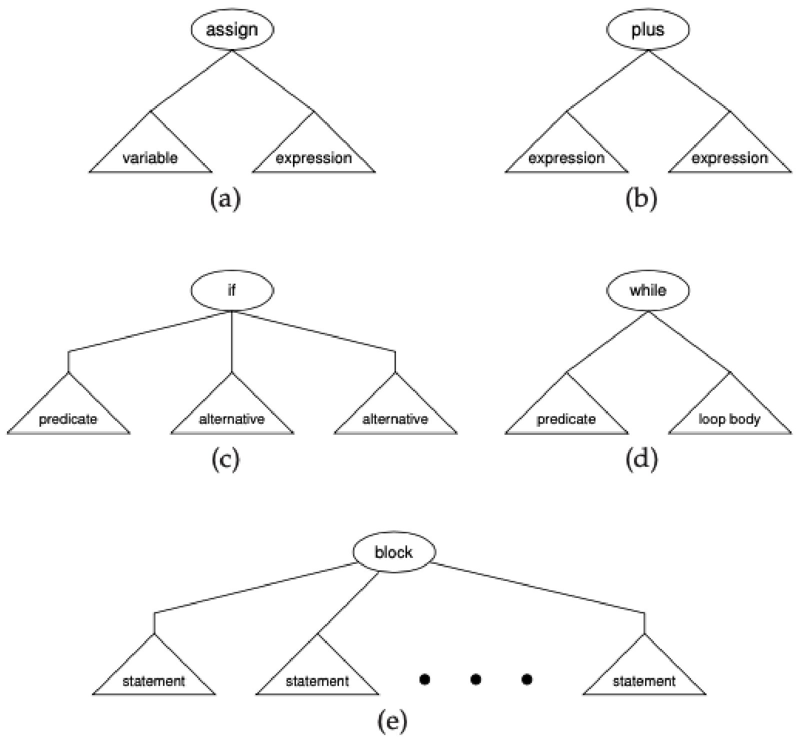


Figure 7.15: AST structures: A specific node is designated by an ellipse. Tree structure of arbitrary complexity is designated by a triangle.

This visitor might include a global variable to keep track of indentation. Upon visiting a node the indentationCounter is incremented, and upon finishing a visit method, the count is decremented. Below is a partial implementation of such a visitor:

```
class PrettyPrintVisitor extends ASTVisitor{
       private int indentation = 0;
       public void visit(IfNode ifNode){
              printTabs(indentation);
              System.out.println(“IfNode”);
              indentation++;
              visit(whileNode.predicate); // Visits
              visit(whileNode.loopBody); // Visits block node
              indentation--;
       }
       public void visit(WhileNode whileNode){
              printTabs(indentation);
              System.out.println(“WhileNode”);
              indentation++;
              visit(whileNode.predicate); // Visits
              visit(whileNode.loopBody); // Visits block node
              indentation--;
       }
       public void visit(BlockNode block){
             printTabs(indentation);
             System.out.println("BlockNode");
             indentation++;
             for(Node n : block.getChildren())
                 visit(n);
             indentation--;
       }
       ...
}
```

The print result should be something like this:

```
IfNode
      PlusNode
            x
            y
      BlockNode
            WhileNode
                  IdNode   (This is the while predicate)
                  AssignNode (This is the while body)
                        IdNode (z)
                        PlusNode
                              IdNode(z)
                              Integernode(1)
            AssignNode
                  IdNode(x)
                  IntegerNode(8)
      AssignNode
            IdNode(z)
            Integer(7)
```

6. Do Fischer et al exercise 18, 21 on pages 339-340 Note the word VISIT is missing! (exercise 18, 19 on pages 371-373 in GE)

   18. Write a visit method for handling the array type definitions represented by the AST in Figure 8.19(b), with upper and lower bounds included in array specification. Note that some type checking on the bounds expressions will be required in this method.

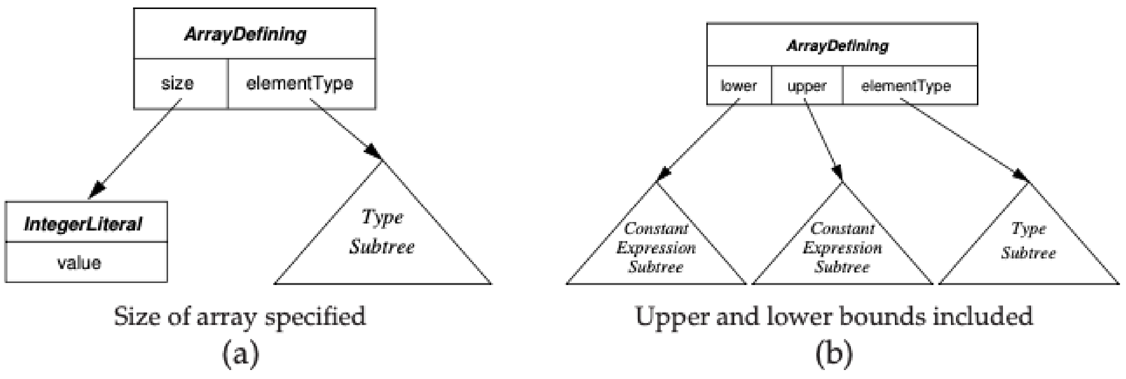
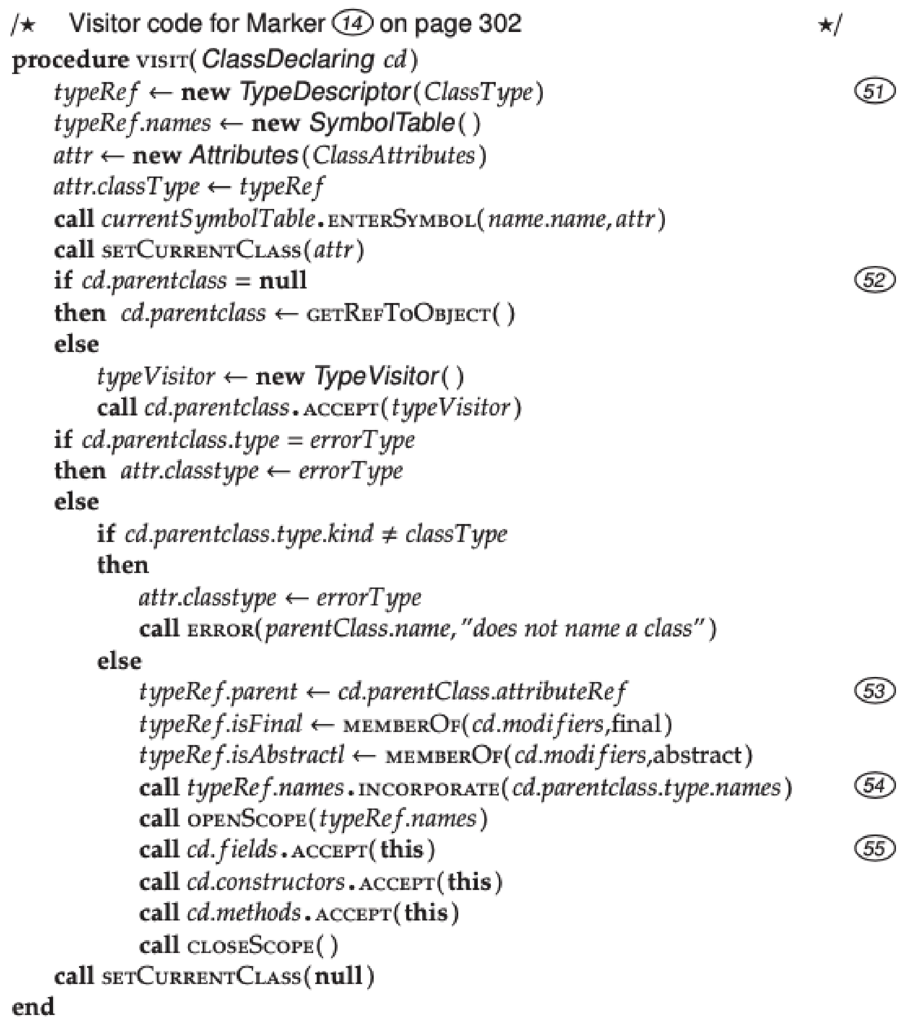


Figure 8.19: Abstract Syntax Trees for Array Definitions

```
class TypeVisitor extends ASTVisitor{
      ...
      Type visit(ArrayDefinition arrayDef){
            lowerBound = visit(arrayDef.lowerBound)
            upperBound = visit(arrayDef.upperBound)
            if(lower.type != integer || upper.type != integer)
                  throw new InvalidBoundTypeException();
            if(lower.value < 0 || upper.value < lower.value)
                  throw new InvalidBoundValueException();
            elementType = visit(arrayDef.elementNode)
            return new ArrayType(elemenType)
      }
      ...
}
```

21. Extend the visit method in Section 8.7.1 to handle the feature in Java that allows a list of interfaces implemented as part of a class declaration (as described at the end of that section). Interface declarations themselves are similar to class declarations. Write a visit method for processing interface declarations.



```
/*   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

Extension to visit(ClassDeclaring cd):

```
...
cd.interfaces ← getRefToInterfaces()
for interface in cd.interfaces:
      if interface.type.kind != interfaceType
      then
         attr.classType ← errorType
      if not cd.methods.containsAll(interface.methods())
      then
         call ERROR("Class does not implement all methods from " +
         interface.name)
...
```

The visit(interfaceDecl) method is very similar to the visit method for class declaration. But recall that interfaces can only extend other interfaces. The following method skips the code duplicated from the visit(ClassDeclaring) Method, and shows only the check to verify that “parent classes” are interfaces:

```
procedure visit(InterfaceDecl iDecl)
      ...
      for parent in iDecl.parentClasses:
           if iDecl.parentclass.type.kind != interfaceType
           then
               attr.classtype ← errorType
               call Error(parentclass.name, "is not an interface")
       ...
```

7.Do Fischer et al exercise 1, 2 on page 385 (exercise 1, 2 on pages 417-421 in GE)

1. *Extend the semantic analysis, reachability, and throw visitors for if statements (Section 9.1.2) to handle the special case in which the condition expression can be evaluated, at compile time, to true or false.*

A detailed answer to this exercise is presented in Solutions Manual for Crafting a Compiler (can be found in general course materials on moodle):

```
procedure      (IfTesting ifn)
    ifn.thenPart.isReachable ← true
    ifn.elsePart.isReachable ← true
    call    C        (ifn)
    constExprVisitor ← new ConstExprVisitor( )
    call ifn.condition.        (constExprVisitor)
    conditionValue ← ifn.condition.exprValue
    if conditionValue = true
    then
        ifn.elsePart.isReachable ← false
        ifn.terminatesNormally ← ifn.thenPart.terminatesNormally
    else
        if conditionValue = false
        then
            ifn.ifPart.isReachable ← false
            ifn.terminatesNormally ← ifn.elsePart.terminatesNormally
        else
            thenNormal ← ifn.thenPart.terminatesNormally
            elseNormal ← ifn.elsePart.terminatesNormally
            ifn.terminatesNormally ← thenNormal or elseNormal
end
```

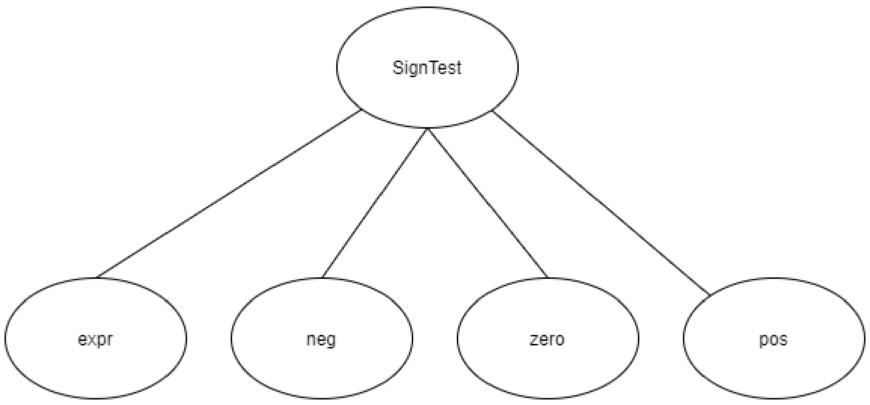
In summary, check if the boolean expression inside the if statement can be computed at compile time. For instance if the expressions consist only of constant values or literals. If that is the case branching is not required and can be optimized out.

2. Assume that we add a new kind of conditional statement to C or Java, the signtest. Its structure is:

```
Signtest (exp) {
        neg:   stmts
        zero:  stmts
        pos:   stmts
}
```

The integer expression exp is evaluated. If it is negative, the statements following neg are executed. If it is zero, the statements following zero are executed. If it is positive, the statements following pos are executed.

Show the AST you would use for this construct. Revise the semantic analysis, reachability, and throws visitors for if statements (Section 9.1.2) to handle the signtest.



A simple implementation of SignTest contains four children. One for integer sign expression and three for the stmts. It's possible to optimize this. For instance if the expression is a value we can compute at compile time, then we can leave out the 2 unused statement branches.

The SignTest is a special case of an if statement. Firstly, we must ensure that the expression, expr, evaluates to an integer. If the result of the expr is a signed integer, then we can mark all 3 statements paths as reachable. If the expr evaluates to an unsigned integer, we can mark the neg path as unreachable. Finally, we have the special case where expr can be evaluated at compile time, In this case we could determine which path would be taken at compile and mark the 2 others as unreachable.

The throws visitor remains the same as for an if statement.

Group Exercises - Lecture 11

1. Discuss the outcome of the individual exercises
   Did you all agree on the results?

2. (optional) Do Fischer et al exercise 26 on page 341 (exercise 27 on pages 373 in GE)
   (use the language you are defining in your project)

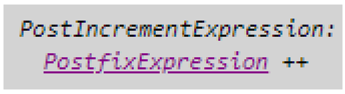
   26. Recall from Section 8.8 that special checking must be done for names that are used on the left-hand side of an assignment. The visitor class LHSSemanticVisitor was introduced for this purpose. Section 8.8.1 noted that since a parser will not allow a literal to appear as the target of an assignment, no visit method for handling literals was included in LHSSemanticVisitor.
   This example uses Java, you should use your own project language!

   a) *For a language of your choice, identify any other contexts in a program where an L-value is required.*

      Increment operators in Java, where for example 5++; is not a legal expression.

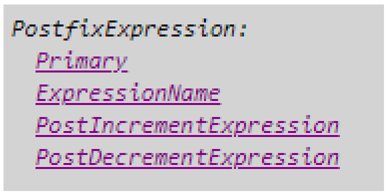
   b) *Do the syntactic rules of the language guarantee that a literal can not appear in these contexts?*

      No, the grammar actually allows literals(which are rvalues) in a postfix increment. The syntax rules specify that a postfix increment has the form:



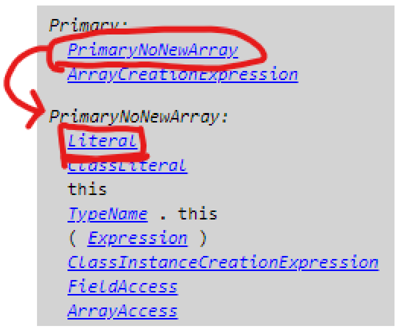
```
PostIncrementExpression:
  PostfixExpression ++
```

      Where a PostFixExpression is defined as:



```
PostfixExpression:
  Primary
  ExpressionName
  PostIncrementExpression
  PostDecrementExpression
```

      And Primary is defined as:



```
Primary:
  PrimaryNoNewArray
  ArrayCreationExpression

PrimaryNoNewArray:
  Literal
  ClassLiteral
  this
  TypeName . this
  ( Expression )
  ClassInstanceCreationExpression
  FieldAccess
  ArrayAccess
```

      ^ From the java syntax specification found on:
      https://docs.oracle.com/javase/specs/jls/se15/html/jls-15.html#jls-PostIncrementExpression

c) *Write appropriate LHSSemanticVisitor visit methods for the AST nodes corresponding to any literals allowed by the language.*

In the **SemanticsVisitor** we create a method for visiting postfix increment nodes:

```
public void visit(PostFixIncrementNode incNode) {
      LHSSemanticVisitor lhsVisitor = new LHSSemanticVisitor();
      incNode.incrementTarget.accept(lhsVisitor);
}
```

Inside the LHSSemanticVisitor we create methods for the possible increment targets (considering only the increment operator):

```
public void visit(IdNode idNode){
      if (symbolTable.get(idNode.name).isFinal()) {
            throw new ConstantModifciationException(...);
      } else if (!symbolTable.get(idNode.name).supportsIncrement()) {
            throw new UnsupportedOperationException(...);
      }
      // You could also add more semantic checks,
      // fx. to check that the variable has been declared
}

public void visit(IntegerLiteralNode intLiteralNode){
      throw new InvalidLHSValueException(...);
}
// Repeat for other literal types such as float, doubles and strings
...
```

3. Do Fischer et al exercise 3, 7, 10, 13, 12 on pages 385-389 (exercise 3, 7, 11, 13, 14 on pages 417-421 in GE)

   3. Assume we add a new kind of looping statement, the exit-when loop to C or Java. This loop is of the form:

   loop

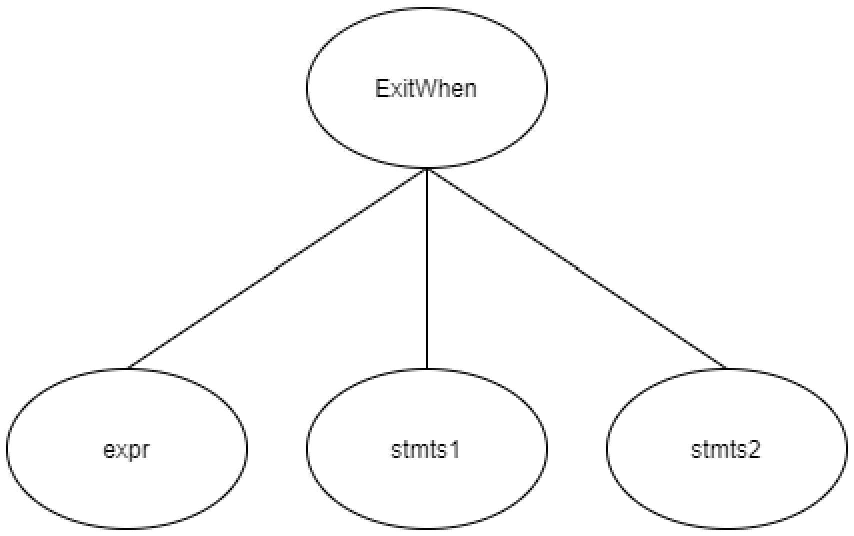
   statements1

   exit when expression

   statements2

   end

   First, the statements in statements1 are executed. Then, expression is evaluated. If it is true, then the loop is exited. Otherwise, the statements in statements2 and statements1 are executed. Next expression is reevaluated and the loop is conditionally exited. This process repeats until expression eventually becomes true (or else the loop iterates forever). Show the AST you would use for this construct. Revise the semantic analysis, *reachability, and throws visitors for while loops (Section 9.1.3) to handle this form of loop. Be sure the special case of expression being constant-valued is handled properly*



Once again we look at the special case where the expression can be evaluated at compile compile time. In this case where it evaluates to true, we can mark stmts2 as unreachable since it will always exit beforehand. If it is false at compile time, then both statements will be reachable.

7. Some programming languages, such as Ada, require that within a for loop, the loop index should be treated like a constant. That is, the only way that a loop index can change is via the loop update mechanism listed in the loop header. Thus (using Java syntax):

for (i=1;i<100;i++) print(i)

is legal, but:

for (i=1;i<100;i++) print(--i)

is not legal. Explain how to change the semantic analysis visitor of Section 9.1.4 to enforce read-only access to a loop index within a loop body.

Give the loop index variable a different type or flag in the symbol table (for example you can mark it as ‘read-only’). With such a flag it is easy to verify that no symbol marked ‘read-only’ is used on the left hand side. The semantic check has to make an exception by not checking the ‘read-only’ for the assignment operation to a loop index using the loop update mechanism in the loop where a specific loop index is declared.

10. One of the problems with the class structure used by Java and C is that field and method declarations (which are terse) are intermixed with method implementations (which can be lengthy and detailed). As a result, it can be hard to casually ”browse” a class definition. As an alternative, assume we modify the structure of a class to separate declarations and implementations. A class begins with class declarations. These are variable and constant declarations (completely unchanged) as well as method headers (without method bodies). An ”implemented as” section follows, which contains the bodies of each method declared in the class. Each method declared in the class must have a body defined in this section, and no body may be defined unless it has been previously declared. Here is a simple example of this revised class structure:

```
class demo {
     char skip = '\n';
     int f();
     void main();
implemented as
     f:    { return 10; }
     main: { print("Ans =",f(),skip); }
}
```

What changes are needed in the semantic analysis of classes and methods to implement this new class structure?

A detailed answer to this exercise is presented in Solutions Manual for Crafting a Compiler. (Can be found in general course materials on moodle)

In summary, the headers must be processed to create the symbol tables entries. then, the bodies can be processed with checks added to ensure the implementations match the definitions, that all declared methods are implemented, and that all implemented methods are declared.

13. As mentioned in Section 9.1.6, C, C++, and Java allow non-null cases in a switch statement to ”fall through” to the next case if they do not end with a break statement. This option is occasionally useful, but far more often leads to unexpected errors. Suggest how the semantic analysis of switch statements (Figure 9.21) can be extended to issue a warning for non-null cases that do not end in a break. (The very last case never needs a break since there are no further cases to ”fall into.”)

The SemanticsVisitor already visits all statements in each case by calling VisitChildren in visit(CaseItem cn). The method visit(CaseItem cn) could be extended to check if this is a non-null case (i.e. it contains statements) and if the more variable in the CaseItem AST node (Figure 9.20) is not null (i.e. this is not the last case). If both of these checks return true and the last statement in the case is not a break, a warning should be emitted that this is neither a none-null case nor is it the last case and as such should probably end with a break.

12. Most programming languages, including C, C++, C, and Java pass parameters positionally. That is, the first value in the argument list is the first parameter, the next value is the second parameter, and so on.

For long parameter lists this approach is tedious and error prone. It is easy to forget the exact order in which parameters must be passed. An alternative to positional parameters is keyword parameters. Each parameter value is labeled with the name of the formal parameter it represents. The order in which parameters are passed is now unimportant.

For example, assume method M is declared with four parameters, a to d. The call M(1,2,3,4), using ordinary positional form, can be rewritten as M(d:4,a:1,c:3,b:2). The two calls have identical effects; only the notation used to match actual parameters to formal parameters differs. What changes would be needed in the semantic analysis of calls, as defined in Figure 9.34, to allow keyword parameters?

A detailed answer to this exercise is presented in Solutions Manual for Crafting a Compiler. (Can be found in general course materials on moodle)

In summary, the primary task is to determine what method is being called given the method name and signature. For calls to methods with positional parameters only, their types and order is required, but for keyword parameters their names must also be taken into account. So to ensure a method can be called with a set of keyword arguments the semantic analysis must find all relevant methods (e.g., overloading allows multiple methods with the same name), and verify that the number of arguments, their types, and their names match a unique method. Python is an example of a language that supports both positional and keyword parameters:

```python
def pow(x, y):
    ...

positional = pow(2, 8)  # 256
keyword = pow(y=8, x=2) # 256
```

Individual Exercises - Lecture 12

1. Do the following for Java, C, PhP and SML: (You may replace the languages with other languages you are familiar with)

   a. *Describe the set of values that data objects of type Integer (int) may contain*

      C: Ints are either 2 or 4 bytes big, in order to see the size of an int on your local compiler load the <limits.h> package and print INT_MAX

      2^16 or 2^32 signed.

      -32,768 to 32,767 or -2,147,483,648 to 2,147,483,647

   b. *Determine the storage representation for values of Integer type*

      C: Stores integers as the binary representation in memory.

      Other languages like Java have the concept of “boxing” of primitive data types like ints and objects. See: https://stackoverflow.com/questions/27647407/why-do-we-use-autoboxing-and-unboxing-in-java

   c. *Determine the syntactic representation used for constants of that type*

      C uses the “const” keyword as part of the type qualifier. Const prohibits modifying an int beyond its initialization.

      Int const a = 1;

   d. *Determine the set of operations defined for Integers and give the signature for these operations and syntactic representation*

      Arithmetic Operators: +, -, *, /, %, ++, --

      Relational Operators: ==, !=, <=, >=, <, >

      Logical Operators: &&, ||, !

      Bitwise Operators: &, ^, |, ~, <<, >>

      Assignment Operators: =, +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=

      Misc Operators: sizeof, &, *, ? :

      C automatically coerces types in some operations such as addition. Adding an int to a float will first convert the int to a float. Same is true for different integer sizes.

e. *Determine if any of the operation symbols defined for integers may be overloaded*

   C does not support operator overloading

   Consider looking at C++ operator or C# operator overloading:
   https://en.cppreference.com/w/cpp/language/operators
   https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/operator-overloading

f. *Determine whether static or dynamic type checking is used to determine the validity of each operation.*

   C utilises static type checking based on the types of the operands.

2. Do exercise 1 for floating point data type

   Like for exercise 1 this information is usually available in the documentation of the language/implementation, its standard, or specified by the implementation of the language.

   For instance for C#:
   https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types

   Most languages map integers directly to their binary representation. Floating point presents new challenges in this format. Different standards exist (as mentioned in today's chapter) for storing floats in binary. These representations are often a compromise between simplicity of the implementation, size, speed of translation/operation, and precision of the value.

3. Consider the languages C and Java. For each language determine what the language implementation does with respect to constructs that cannot be checked statically, e.g. arrays bounds check, access to pointers/references and division by 0.

a) *Does the language provide dynamic checking or leave it unchecked at runtime?*

   Java provides runtime bounds checking. For array access it throws an `ArrayIndexOutOfBoundsException` when trying to access indexes outside the specified array size. Additionally, when accessing variables that have not yet been assigned a value a `NullPointerException` is thrown (not the case for primitive types as they have default values).

   In C, however, there is no runtime checking for array indexes and non-initialized values.This means that it is possible to lookup data located outside of the array in memory. Additionally, no checks are made when accessing uninitialized data structures, and they may contain garbage data. Welcome to the land of undefined behaviour.

Java provides runtime checks for division by 0 and throws an instance of ArithmeticException when it occurs. In C division by zero is undefined behaviour but does not trigger any exceptions.

a. *Are there other constructs you can think of?*
Legal values for types. Fx. both C and Java exhibit rollover behaviour when encountering an integer overflow. Neither language provides runtime checking for this. Some versions of the Ada language trigger an exception in case of overflow.

a. *What about the language you are designing?*

What data types are in your language? Do you perform any runtime or compile checks on them?

4. The language SIMSCRIPT allows data elements of different types to be stored as elements of an array. One reason for this is that arrays in SIMSCRIPT are implemented as vectors of pointers to the data elements thus each element in an array has a uniform representation. Both Java and SML use the same implementation technique, but do not allow data elements of mixed types in arrays. Can you explain why? Can you give a work-around in Java that enables you to store mixed elements in an array? Can you explain why this works in Java? What about your own language – will it allow mixed types in arrays/compound data types – why/why not?

Java can provide stronger type checking at compile time by only allowing one type of element in a collection.

To work around this, collections, e.g. `ArrayList`, can be declared with the type `Object`, which enables all types deriving from `Object(every non-primitive type)` to be added to the list regardless of their type. Additionally, the collections can be declared using interfaces or super classes(like Object), which both also enable different subtypes in the same array.

5. Do Sebesta exercise 9 on page 322.

9. Multidimensional arrays can be stored in row major order, as in C++, or in column major order, as in Fortran. Develop the access functions for both of these arrangements for three-dimensional arrays.
See the following resources:

- https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

- https://stackoverflow.com/questions/14015556/how-to-map-the-indexes-of-a-matrix-to-a-1-dimensional-array-c

Group Exercises - Lecture 12

1. Discuss the outcome of the individual exercises

   Did you all agree on the results?

2. Do the individual exercise 1 and 2 for your own language

   If you do not deal with the low level details of how integers and floats are represented in your language, then consider how they are represented in the language that you are compiling to.

3. Do Sebesta exercise 13 on page 323 (rather than write, discuss with your group) or more to the point discuss what was lost and what was gained by excluding pointers from the design of Java

   What was gained:
   - Simplification of the language. No need to worry about dereferencing and pointer arithmetic (which can often lead to program errors for unexperienced programmers)

   What was lost:
   - Low level control of memory. I.e no/limited control of stack/heap allocation (possible through `sun.misc.Unsafe`) and no option to pass parameters by reference(parameters are always passed by value in Java).

   See Sebesta - 6.11.7 Implementation of Pointer and Reference Types for more Information.

4. What are the arguments for and against heap management with single-sized elements and variable-sized elements?
   Single-sized cells:
      For:
      - Simple
      - Allocation is easy (just follow the linked list of pointers to available cells and take the amount you need)

      Against:
      - Not as powerful, most languages need variable-sized cells.
      - Garbage collection is difficult
         - Potential for multiple pointers to same cell (Can be solved using reference counting or the mark sweep approach)

   Variable-size cells:
      For:
      - Heap space can be utilized more effectively with objects of variable size

      Against:

        - Has all the sames difficulties of managing a heap of single-size cells
        - Scanning is problematic, where does the next cell start, if they have different sizes? (This can be solved by having each cell store its own size, but this uses more space)
        - Finding an available (big enough) space is a complex problem

    For more information see Sebesta Section 6.11.7.3 Heap Management

5. Do Sebesta exercise 16 on page 323.

    16. What would you expect to be the level of use of pointers in C#? How often will they be used when it is not absolutely necessary?

    Pointers are rarely used in C# as constructs are added to replace them, specifically: references, out parameters, and the Marshal Class. As such, pointers are generally only used when a managed solution adds an unacceptable overhead, and their use is discouraged by the language designers by requiring the `unsafe` keyword .

    The principal argument for the more restrictive constructs used in C# is better error detection, with which better reliability is obtained. A perhaps nebulus argument for the use of pointers would be the loss of freedom when, e.g. using references. Languages such as C and C++ put a stronger 'faith' in the programmer, that they know what they are doing and will not make mistakes.

Individual Exercises - Lecture 13

1. Read the articles under additional references.
   The articles will be used as a basis for the discussions in the group exercises.

2. Sebesta chapter 7, p 350-351, exercises 9,13, 19
   9. Assume the following rules of associativity and precedence for expressions:

Precedence	*Highest*	*, /, **not**
		+, −, &, **mod**
		- (unary)
		=, / =, <, <=, >=, >
		and
	Lowest	**or**, **xor**
Associativity	*Left to right*	

Show the order of evaluation of the following expressions by parenthe- sizing all subexpressions and placing a superscript on the right parenthesis to indicate order. For example, for the expression
a + b * c +d
the order of evaluation would be represented as
$((a + (b * c)^1)^2 + d)^3$

Exercise	Original Expression	Expression with Explicit Evaluation Order
A	a * b - 1 + c	$(((a * b)^1 - 1)^2 + c)^3$
B	a * (b - 1) / c mod d	$(((a * (b - 1)^1)^2 / c)^3 mod\, d)^4$
C	(a - b) / c & (d * e / a - 3)	$(((a - b)^1 / c)^2 \& (((d * e)^3 / a)^4 - 3$
D	-a or c = d and e	$((- a)^1 or\, ((c = d)^2 and\, e)^3)^4$
E	a > b xor c or d <= 17	$(((a > b)^1 xor\, c)^3 or\, (d <= 17)^2)^4$
F	-a + b	$(- (a + b)^1)^2$

13. Let the function fun be defined as

```
int fun (int* k) {
    *k += 4;
    return 3 * (*k) - 1;
}
```

Suppose fun is used in a program as follows:

```
void main() {
    int i = 10, j = 10, sum1, sum2;
    sum1 = (i / 2) + fun(&i);
    sum2 = fun(&j) + (j / 2);
}
```

This exercise does not specify which operator precedence and associativity to use. In our solution and the one provided by Sebesta we assume the program to be written in C (as the syntax indicates).

For more information on C: https://en.cppreference.com/w/c/language/eval_order

a. operands in the expressions are evaluated left to right? Sum1: 46, Sum2: 48

b. operands in the expressions are evaluated right to left? Sum1: 48, Sum2: 46

19. Consider the following C program:

```
int fun (int* i) {
    *i += 5;
    return 4;
}

void main(){
    int x = 3;
    x = x + fun(&x);
}
```

What is the value of x after the assignment statement in main, assuming
a. operands are evaluated left to right? X: 7
b. operands are evaluated right to left? X: 12

3. Sebesta chapter 7, p 352, programming exercises 1 and 2

1. Run the code given in Problem 13 (in the Problem Set) on some system that supports C to determine the values of sum1 and sum2. Explain the results.

C program:

```c
#include <stdio.h>
int fun(int* k) {
    *k += 4;
    return 3 * (*k) -1;
}
void main() {
    int i = 10, j = 10, sum1, sum2;
    sum1 = (i / 2) + fun(&i);
    sum2 = fun(&j) + (j / 2);
    printf("Sum1: %d, Sum2: %d \n", sum1, sum2);
}
```

Prints: Sum1: 46, Sum2: 48

The result indicates that the implementation of C used evaluated the expressions from left to right. By evaluating the expression from left to right `i` is divided by 2 before four is added to it so `sum1` is 46. In contrast `sum2` is 48 as 4 is added to `j` before j is divided by 2.

Again for more information on C: https://en.cppreference.com/w/c/language/eval_order

2. Rewrite the program of Programming Exercise 1 in C++, Java, and C#, run them, and compare the results.

C++ Program: the result for C++ is the same as for C.

Java Program: Java uses pass-by-value exclusively. To emulate the behavior of the c program a wrapper class is created and used in place of the integers i and j. This simulates pass-by-reference semantics.

```java
public class JavaVersion {

    private static class IntWrapper {
        int val;

        public IntWrapper(int val) {
            this.val = val;
        }
    }

    static int fun(IntWrapper wrappedInt){
        wrappedInt.val += 4;
        return 3 * wrappedInt.val - 1;
    }

    public static void main(String[] args) {
        IntWrapper i = new IntWrapper(10), j = new IntWrapper(10);
        int sum1, sum2;
        sum1 = (i.val / 2) + fun(i);
        sum2 = fun(j) + (j.val / 2);
        System.out.println(String.format("Sum1: %d, Sum2: %d", sum1, sum2));
    }
}
```

Prints: sum1 = 46, sum2 = 48

```
class Program { // C# version
    static int fun(ref int k) {
        k += 4;
        return 3 * (k) - 1;
    }
    static void Main(string[] args) {
        int i = 10, j = 10, sum1, sum2;
        sum1 = (i / 2) + fun(ref i);
        sum2 = fun(ref j) + (j / 2);
        Console.WriteLine($"Sum1: {sum1}, Sum2: {sum2}");
    }
}
```

Prints: Sum1 = 46, Sum2 = 48

4. Sebesta chapter 8, p 386, programming exercises 1.a

1.a Rewrite the pseudocode segment using a loop structure in C, C++, Java, or C#

```
  k = (j + 13) / 27
loop:
  if k > 10 then goto  out
  k = k + 1
  i = 3 * k - 1
  goto loop
out: . . .
```

```
// Java code example
int i, j, k;
for (k = (j + 13) / 27; k > 10; k++) {
    i = 3 * k - 1;
}
```

Group Exercises - Lecture 13

1. Discuss the outcome of the individual exercises

   Did you all agree on the answers?

2. Discuss the article “Python & Java - A Side-by-Side Comparison”

   Find it here:
   https://pythonconquerstheuniverse.wordpress.com/2009/10/03/python-java-a-side-by-side-comparison/
   Consider topics such as productivity and expressiveness, i.e. how much code do you need to write to express something?

3. Discuss the article “go to statement considered harmful” and “Structured Programming with go to Statements”

   Find them here:
   http://doi.acm.org/10.1145/362929.362947
   and
   http://doi.acm.org/10.1145/356635.356640
   Do you consider gotos harmful? Why do you have that opinion?

4. Sebesta chapter 7, p 350, exercises 4

   4. Would it be a good idea to eliminate all operator precedence rules and require parentheses to show the desired precedence in expressions? Why or why not?
   As input to the discussion it is worth looking into how the Lisp languages (such as Clojure, Common Lisp, and Scheme) are designed as these languages operate in that way. In Lisp operators are functions that are applied to arguments, e.g, 2 + 2 becomes (+ 2 2) which is very similar to the function syntax used by imperative languages like C plus(2, 2). So if the goal is to create a language that focuses on functions (or lists) as much as possible, then eliminating operator precedence could be a good idea.

   It is worth noting that an expression with operators can easily be ambiguous if the programmer isn’t familiar with the precedence, but that this isn’t possible in the same way when users are forced to add parentheses. However requiring parentheses can of course be a burden, e.g. by requiring more typing, and because users cannot rely on their knowledge of mathematics to get an intuitive understanding of how expressions work when learning the language. Extensive use of parentheses could also have a negative impact on readability.

5. Should C's assignment operations such as += be included in other languages? Why or why not?

The need to write an assignment that is based on the original value comes up often in imperative languages, e.g. as an index variable when iterating over data structures.

Writing:

```
    A = A + B;
```

Is simply more verbose than:

```
    A += B;
```

On the other hand, with a new operator not known from mathematics it can be argued that the operation becomes harder to read as users might not know what '+=' means? And as such, that the inclusion of assignment operations trade writability for readability.
This example is of course relatively mild, but as expressivity goes to extremes, readability goes down, so if the goal was to make an extremely readable language (perhaps a beginner's first imperative language?) compounded assignment operators like those featured in C would be harder to defend. For an example of a programming language with very concise code, arguably at the cost of readability, see APL: https://en.wikipedia.org/wiki/APL_(programming_language)

6. Should C's single-operand assignment form, such as ++, be included in other languages? Why or why not?

See Exercise 5 above, as the arguments for and against adding the ++ operator are quite similar to the arguments for and against adding the += operator.

7. Sebesta chapter 8, p 385-386, exercises 5 and 9

5. What are the arguments, pros and cons, for Python's use of indentation to specify compound statements in control statements?

In Python:

```python
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

The scopes of control statements are marked with levels of indentation which makes them more readable. However, the lack of explicit end markers also make it harder to understand what scope a statement is part of if many nested scopes are used, and simply changing the indentation of a line can cause either parsing error or change the meaning of the program in the case of the last line. The latter might be more serious, since this error can go unnoticed by the programmer if the compiler does not express an error.

9. What are the arguments both for and against the exclusive use of Boolean expressions in the control statements in Java (as opposed to also allowing arithmetic expressions, as in C++)?

It can be argued that allowing arithmetic expressions to exist in control statements (where 0 would evaluate to false and any other number to true) grants a greater degree of expressivity.

An argument for exclusively using boolean values is that it can increase readability and allows the compiler to catch certain type errors at compile time, e.g, x = 1 is not a boolean expression but an assignment(which returns a value is some languages). This means that ‘if (x = 1)’ can be detected as an error. Furthermore, it is very easy to emulate the behavior of arithmetic expressions as booleans by adding a comparison operator e.g, aritmeticExpr != 0.

Individual Exercises - Lecture 14

1. Read the articles under additional references
   They can be found here:
   http://zorac.aub.aau.dk/login?url=http://link.springer.com/book/10.1007%2F978-0-85729-829-4 (chapter 4, p97-106)
   Preprint:
   http://people.cs.aau.dk/~bt/PLATEAU2016/Preprint-PLATEAU2016-KurtevChristensenThomsen.pdf
   Sample sheet: http://people.cs.aau.dk/~bt/PLATEAU2016/samplesheet.pdf
   Task sheet: http://people.cs.aau.dk/~bt/PLATEAU2016/tasksheet.pdf

2. Go through the review questions (Sebesta) 1-37, p. 436-437

   1. What are the three general characteristics of subprograms?

   There is a caller and a callee

   Callee returns control to the caller upon completion

   The subprogram has **1** entry point.

   While the callee is called, the caller is suspended, awaiting return of control from the callee.

   2. What is a subprogram call?

   Request for execution of a specific subprogram (Sebesta 391)

   3. What is a subprogram definition?

   A subprogram definition consists of a header (describes the interface of the subprogram) and a body (describing the actions of the subprogram) (Sebesta 391)

   4. What characteristic of Python subprograms sets them apart from those of other languages?

   Function definitions (keyword 'def') are executable statements that assign the function name to the function body. (Sebesta 391)

   5. How do Ruby methods differ from the subprograms of other programming languages?

   In Ruby methods can be defined outside class definitions, in which case they are considered methods of the root 'Object'(Sebesta 392)

   6. What is the feature of Lua functions?

   All Lua functions are anonymous (they don't have names) (Sebesta 392)

   7. What are function declarations called in C and C++? Where are the declarations often placed?

   Prototypes. They are often placed near the top of the program or in header files.

8. Name one pure functional programming language that does not have mutable data.

Haskell (Sebesta 393)

9. What are positional parameters?

For example a function in Java, that has the following parameters (int i, float f, String s), could not take the following arguments (“string”, 5.2, 1), even though each of the arguments’ types is represented. They have to follow the order of the parameters.

10. Give an example of a language that allows positional parameters in addition to keyword parameters.

Python, C#

11. What is the use of a default value in a formal parameter?

Providing a default value for a parameter makes it optional during invocation. If a value is not provided for the parameter it will set to its default value. Some languages, like C#, requires all non-optional parameters to precede optional ones. This is to ensure each argument is matched to the correct parameter. An alternative to this is using named parameters.

See:https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/named-and-optional-arguments

12. What is the rule of using a default parameter in C++?

Formal parameters with default values must appear after all other parameters because C++ use positional parameters

13. What is the rule for accepting variable parameters in C# methods?

Use the params keyword. All parameters must be of the same type

14. What language allows array formal parameters?

Ruby

15. What is an ellipsis?

Java ellipsis syntax (...) is used to accept a variable number of parameters

16. What are the modes, the conceptual models of transfer, the advantages, and the disadvantages of pass-by-value, pass-by-result, pass-by-value- result, and pass-by-reference parameter-passing methods?

Pass-by-value: In-mode. Often implemented by copy.

One advantage is that the callee can freely modify the passed value, and the caller can know that the passed variable will not be affected (this is complicated slightly by boxing). A disadvantage is that it can be expensive if the passed value is large.

Pass-by-result: Out-mode. Typically by copy.

Similar advantages/disadvantages of pass-by-value. Has the problem of parameter collision. See Sebesta 9.5.2.2.

Pass-by-value-result: Inout. By copy.
Has the same problem of memory/cpu cost associated with copying. Also has the problem of parameter collision.

Pass-by-reference: Inout. By access path.
Passing the parameter is effective in terms of memory and computation time. Access to the formal parameters is, however, slightly slower due to extra level of indirection. A disadvantage is that the subprogram may unexpectedly alter the variable you pass to it, or even store the reference and alter the value later. This can make debugging more challenging.

17. Describe the ways that aliases can occur with pass-by-reference parameters.
If the parameter of a function has a different name, than the argument being passed to the function. Both variables will point to the same object / place in memory, but they are referred to by two different names.
18. What is the difference between the way original C and C89 deal with an actual parameter whose type is not identical to that of the corresponding formal parameter?
In original C neither the number of parameters and their types were not checked. In C89 the type is included for the parameters in the function prototype.
19. Name some languages that support procedures.
C++, C, Python
20. Describe the problem of passing multidimensional arrays as parameters.
The problem relates to storing arrays of arrays. For instance in C, a function can only occupy a fixed size of memory and so a function must be written for each length of array that needs support, ie to pass an array with 100 columns you need the following function definition in C:

```
void fun(int matrix[][100])
```

..and to pass an array with 101 columns you need a function with definition:

```
void fun(int matrix[][101])
```

This problem can be avoided if the array is passed as a pointer, but then you will also need to pass the lengths of the array (rows and columns) and calculate the index according to row major order (See Sebesta 9.5.6).

21. What is the name of the parameter-passing method used in Ruby?

Ruby uses pass-by-assignment. (see page 410 in Sebesta)

22. What are the two issues that arise when subprogram names are parameters?

Type checking the parameters of the passed subprogram and which environment should be used to execute the passed subprogram

23. Define shallow and deep binding for referencing environments of subprograms that have been passed as parameters.

Shallow: The environment of the call statement that enacts the passed subprogram. That is, the passed subprogram runs within the environment of the subprogram that calls it.

Deep: The environment of the definition of the passed subprogram. That is, the passed subprogram runs within the environment of where it was defined.

24. What is a generic subprogram?

Functions that work regardless of the type of the argument given. This saves time, since some algorithms can now be implemented only once.

25. What is ad hoc binding?

Ad-hoc binding refers to a method for determining which environment to reference when executing nested subprograms. In ad-hoc binding we use the environment of the call statement that passed the subprogram as an actual parameter. For a more thorough explanation see Sebesta Section 9.6 pages 418-419.

26. What causes a C++ template function to be instantiated?

Naming the function in a call, or taking its address with the '&' operator

27. In what fundamental ways do the generic methods of Java 5.0 differ from those of C# 2005?

C# does not support wildcard types (indicated by '?' in Java) (Sebesta 428)

28. If a Java 5.0 method returns a generic type, what type of object is actually returned?

During execution the raw method works on objects of the Object class. Upon returning the value it is cast to the appropriate type.

29. If a Java 5.0 generic method is called with three different generic parameters, how many versions of the method will be generated by the compiler?

Only one copy of the code is built (because, as stated above, the raw method operates on objects of the Object class).

30. When does a variable have unlimited extent?

A variable has unlimited extent, if the variable's lifetime is the same as the lifetime of the entire program.

31. What is subtype polymorphism?

This is a concept from object oriented programming, where a variable of type A can be assigned a value of type A or any of the subclasses derived from A.

32. What is a multicast delegate?

A delegate class that can store a list of delegates, which can then be called in order and the value of the last delegate is returned.

For definition of a delegate see Sebesta 9.7 on page 420.

33. What is the main drawback of generic functions in F#?

F# performs type inference for generic functions. So some generic functions like adding to parameters, will have their types inferred to int and the function becomes less useful.

34. What is a coroutine?

- A normal function will run to completion before returning.
- A coroutine is a function that has the ability to pause execution, and return control to the caller for a time, before picking it up again where it left off.

35. What are the language characteristics that make closures useful?

Statically scoped, allow nested subprograms and allows subprograms as parameters

36. What languages allow the user to overload operators?

Python, C#, C++, Haskell… (see https://en.wikipedia.org/wiki/Operator_overloading for a longer list)

37. What is a symmetric unit control model?

The coroutine control mechanism (Sebesta 9.13, p. 432)

3. Do Sebesta chapter 9, p 438, exercise 4

4. Suppose you want to write a method that prints a heading on a new output page, along with a page number that is 1 in the first activation and that increases by 1 with each subsequent activation. Can this be done without parameters and without reference to nonlocal variables in Java? Can it be done in C#?

The Java program below implements a counter that does not make use of method arguments nor nonlocal references:

```
public static void main(String[] args) {
   Runnable incrementer = createIncrementer();
   incrementer.run(); // Prints 0
   incrementer.run(); // Prints 1
   incrementer.run(); // Prints 2
}

public static Runnable createIncrementer(){
   AtomicInteger i = new AtomicInteger();
   return () -> System.out.println(i.getAndIncrement());
}
```

And here is a C# version

```
class Program {
    static void Main(string[] args) {
        var pageCounter = GetPageCounter();
        Console.WriteLine($"Page: {pageCounter()}");
        Console.WriteLine($"Page: {pageCounter()}");
        Console.WriteLine($"Page: {pageCounter()}");
    }

    static Func<int> GetPageCounter() {
        int page = 0;
        return () => page++;
    }
}
```

4. Argue for or against Java's design not to include operator overloading.

Allowing operator overloading makes operations with mathematical entities like complex number and matrices simpler (e.g. `BigDecimal` in Java uses an add method instead of +). On the other hand by allowing users to overload operators, types can be created where + and - have completely different meaning making programs much harder to understand. However, this negative aspect is similar to the problem of classes and method being poorly named. Consider for example the addition of two lists list1 + list2, does this refer to a pairwise addition of the elements, or concatenation of list1 and list2?

5. Go through Sebesta chapter 14, p 652-653, review questions 1-27

1. Define exception, exception handler, raising an exception, continuation, finalization, and built-in exception.

Exception: The state of the program has somehow deteriorated to a point where the intended computation is no longer possible and instead of continuing an exception is raised as the mechanism to handle this state, via the exception handler.
Exception handler: A component responsible for deciding how to proceed in the event of a specific type of exception.
Raising an exception: An event occurs, that triggers an exception. For example, some invariant no longer holds.
Continuation: The question of whether to terminate the program or resume exection.
Finalization: Code that is always executed, regardless of whether an exception is thorwn/raised. For example the finally block in java.
Build-in exception: The exception types built into the language (as opposed to programmer-defined exceptions)

2. How did the early programming languages support the exception-handling mechanism?
In early programming languages the occurrence of errors simply caused the program to be terminated and control to be transferred to the operating system. Alternatively, some languages, e.g. C, use integer values to represent at exit status of a program. This integer value may represent some exception/error.
3. What are the advantages of having support for exception handling built into a language?
It allows for programmers to create code that to some extent attempts to handle exceptions that could occur as a construction that isn't directly written along with the rest of the program's statements.

It makes it possible to handle exceptions externally by raising errors up though the call stack to an eventual point that has a solution.
4. What is the reason that some languages do not include exception handling?
One reason some languages do not include exception handling is the complexity it adds to the language. Some methods of exception handling could be considered a special case of the goto, a jump to another point in the program.
5. Give an example of an error that is not detectable by hardware but can be detected by the code generated by the compiler in Java.
Checking that arrays are indexed within their bounds.
6. What is the use of label parameters in Fortran?
This allows the callee to return to a different point in the caller if an exception is thrown.
7. How does exception propagation work in Java?
An exception is propagated through the call-stack until an appropriate `try/catch` catches the exception. If the exception is not caught before the main method, the program terminates.
8. How does a throw statement without an operand work in C++?
An empty `throw` either rethrows the exception if used inside a handler or terminates the program. For more information see (2):
https://en.cppreference.com/w/cpp/language/throw
9. How can an exception handler be written in C++ so that it handles any exception?
C++ has a special construct for catching all exceptions:
catch(...)
10. What constraint will a function overriding a function that has a throws clause face?
The overriding function cannot have a more general exception in its own throws clause.
11. Does Java include built-in exceptions?
Yes. Here is a list
(https://www.geeksforgeeks.org/types-of-exception-in-java-with-examples/)
12. Which library functions in C++ support the overflow_error exception?
Some third party math libraries like boost.math

13. What are the two system-defined direct descendants of the Throwable class in Java?
Error and Exception.
14. What package in Java contains the Array Index Out Of Bounds Exception?
Java.lang.Exception
15. How can an exception handler be written in Java so that it handles any Exception?
catch(Exception e) will catch all exceptions.
catch(Exception e), however, catches exceptions and serious runtime errors that most programs should not catch (e.g., VirtualMachineError - the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating.).
16. What is the difference between C++ exception handlers and Java exception handlers?
C++ can throw anything as an exception, Java can only throw 'throwables'
C++ has a catch all exception
17. What is the difference between Java exception handlers and Python exception handlers?
Python's handlers allow for an `else` clause to be defined which is called if no exception is raised.
18. How can a method deal with exceptions it receives from another method that lists a particular checked exception in its throw clause?
It can call the other method inside a try statement, with a catch clause, that handles the case, where the exception is thrown. In Java, methods with a checked exception can only be called inside a try-clause with a catch-clause catching the checked exception.
19. In Java, when does a try clause only have a finally clause and no exception handlers?
This was a result of the explicit lock interface introduced in Java 5.0. Now a section that acquired a lock will be implemented in a try clause, with the lock being released in a finally clause.
20. In Java, what clause should a method include to propagate any checked exception?
A `throws` clause must be added to the methods definition.
21. Explain what an else block in Python does.
This is a block of code that only executes in the absence of an exception thrown in the try clause.
22. What advantages do assert statements have?
They can help in debugging by telling exactly where in the program a function started having unexpected values. These assertions can also be disabled for the deployment build. This saves time for later program maintenance.
23. What happens when an assert statement evaluates to false?
In Java the AssertionError exception is thrown.
24. What does the message method of Ruby's StandardError class do?
Returns the result of invoking `exception.to_s`. Normally this returns the exception's human-readable error message or its name, as `to_s` by default returns the exception's message or the name of the exception if no message is set.

25. What exactly happens when a raise statement with a string parameter is Executed?
In Ruby this causes an `RuntimeError` to be raised with the string as its message.
26. What does the Python _debug_ flag do?
A constant that can be disabled by running the python program with the -o flag. Means that unnecessary calls, such as messages written to the console during design time can be removed elegantly.
27. In what ways are exception handling and event handling related?
In both cases a pre-registered handler is called when something occurs (an exception or an event). However, while exceptions are either explicitly raised by user code or implicitly raised by the runtime environment, events are created by external actions such as the user clicking a button in a gui.

For more information see Sebesta Section 14.5 - 14.7

6. (optional but recommended) In exercise 2 for Lecture 11 you created an AST for the little language defined in Figure 7.14. Design and implement a recursive interpreter for the language based on the Visitor Pattern (clue: look at the Visitor for pretty printing you implemented in exercise 4 in Lecture 11)

```
 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.

```
// Visitor for IfExpression in the main interpreter class
public void visit(ifExprNode ifNode) {
      boolean predicateResult;
      predicateResult = ifNode.predicateNode.accept(new BooleanVisitor());
      if (predicateResult) {
            // Execute statements inside the if statement body
            ifNode.Body.accept(this);
      } else if (ifNode.hasElseClause()) {
            // Execute statements inside the else body
            ifNode.elseBody.accept(this);
      }
}
```

This language does not feature boolean expressions/operators, so we are assuming the c-style approach to boolean values, where 0 represents *false* and all other numbers represent *true*. The ExpressionVisitor recursively calls down depth first into the expression until literal values or ids are found. If an id is found, the value of said variable is looked up in the symbol table, and the expression is then evaluated using the values from the variables and the literals values of the expression when returning up the tree.

```
public boolean visit(PlusExpression plusNode) {
      int leftValue = plusNode.leftChild.accept(new ExpressionVisitor());
      int rightValue = plusNode.rightChild.accept(new ExpressionVisitor());
      int sum = leftValue + rightValue;
      return sum != 0;
}
```

Group Exercises - Lecture 14

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Do Sebesta chapter 9, p 438, exercise 5
   5. Consider the following program written in C syntax:

```
void swap(int a, int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void main(){
    int value = 2, list[5] = {1, 3, 5, 7, 9};
    swap(value, list[0]);
    swap(list[0], list[1]);
    swap(value, list[value]);
}
```

   For each of the following parameter-passing methods, what are all of the values of the variables value and list after each of the three calls to swap?

   A) Passed by value,
   swap(value, list[0])   => value: 2 and list: 1, 3, 5, 7, 9
   swap(list[0], list[1])   => value: 2 and list: 1, 3, 5, 7, 9
   swap(value, list[value]) => value: 2 and list: 1, 3, 5, 7, 9

   B) Passed by reference
   swap(value, list[0])   => Value: 1 and List: 2 3 5 7 9
   swap(list[0], list[1])   => Value: 1 and List: 3 2 5 7 9
   swap(value, list[value]) => Value: 2 and List: 3 1 5 7 9

   C) Passed by value-result
   As stated in Sebesta Section 9.5.2.2 when using variables that are passed-by-result for indexing arrays the effect depends on when the parameter is bound to an address. If the address is bound at the time of the call the result will match B, otherwise the result is:

   swap(value, list[0])   => Value: 1 and List: 2 3 5 7 9
   swap(list[0], list[1])   => Value: 1 and List: 3 2 5 7 9
   swap(value, list[value]) => Value: 2 and List: 3 2 1 7 9

   For more information about parameter passing methods see Sebesta Section 9.5.2

3. What mechanism can be used for detecting errors in programming languages without exception handling?

Examples include error codes (e.g., C, C++, Go), returning types that can be either the result or an empty value like Option in Rust (e.g., Rust, OCaml, Haskell), or jumping to different labels depending on if the operation fails or succeeds (e.g. Fortran see Sebesta Section 14.1).

4. What is the difference between exception handling in C# and Java?

Java supports both checked and unchecked exceptions, C# only has unchecked exceptions.
A checked exception is an exception the caller is forced to handle if the callee throws it.

5. (optional) Write a recursive interpreter for (a subset) of the language you are designing in your project

Only do a very small subset of your language.

Individual Exercises - Lecture 15

1. Read the articles under additional references.
   The lean, mean, virtual machine:
   https://www.infoworld.com/article/2077184/the-lean--mean--virtual-machine.html
   The Jasmin User Guide:
   http://jasmin.sourceforge.net/guide.html
   Technical Overview of the Common Language Runtime (or why the JVM is not my favorite execution environment):
   https://www.semanticscholar.org/paper/Technical-Overview-of-the-Common-Language-Runtime-Meijer-Miller/60d90adf79acd100704ccc5d476982569272251c

2. Do Fischer et al exercise 4, 7, 8, 9 on pages 414-415 (exercise 4, 5, 8, 10 on pages 446-447 in GE)

   4. Why are there two instructions (ldc and ldc_w) for pushing a constant value on top of the stack?
   In summary, ldc selects what to push onto the stack using one byte for the index while ldc_w uses two bytes (wide index). The wide index is constructed as indexbyte1 << 8 + indexbyte2.

   7. The JVM has two instructions for unconditional jumps: goto and goto_w. Determine the appropriate conditions for using each instruction.
   The goto instruction forms a jump offset using two bytes, while the goto_w forms the jump offset using four bytes. Thus, if the intended target of the jump can be reached by an offset o, $-32768 \leq o \leq 32767$, then the goto instruction suffices. Otherwise, the goto_w instruction must be used. It turns out, however, that the size of a JVM method cannot exceed 65535 bytes, so the goto_w instruction is rarely needed.

   8. The ifeq instruction provisions only 2 bytes for the branch offset. Unlike goto, there is no corresponding goto_w for instruction ifeq. Explain how you would generate code so that a successful outcome of ifeq could reach a target that is too far to be reached by a 16 bit offset.
   The ifeq instruction is a conditional jump using two bytes for the offset, to jump further one can simply jump to another jump instruction that supports a larger offset like goto_w.

9. Consider the C or Java ternary expression: (a > b) ? c : d, which leaves either c or d on TOS depending on the outcome of the comparison. Assume all variables are type int, but do not assume that any of them are 0. Explain how you would generate code for the comparison that uses only the ifne instruction for branching (no other if or goto instructions are allowed).

The ifne instruction can be used to compute a > b by checking that all other conditions are not True, for example checking for equality by subtracting the two numbers and checking if the result is not 0.
If a > b then a - b would be positive. Then we can use ifne to check the sign bit by shifting right 31 bits (Given that the int is represented as 32 bit signed).
For a more thorough explanation see Solutions Manual for Crafting a Compiler available on Moodle (Page 35)

3. (optional) Follow the studio associated with Crafting a Compiler Chapter 10: Intermediate Representations
The materials for this exercise can be found in the virtual machine available on Moodle.

4. (optional) Follow the Lab associated with Crafting a Compiler Chapter 10: Intermediate Representations
The materials for this exercise can be found in the virtual machine available on Moodle.

Group Exercises - Lecture 15

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Do Fischer et al exercise 5, 6, 10 on pages 414-415 (exercise 6, 7, 9 on pages 446-447 in GE)

   5. The form of the getstatic instruction is described in Section 10.2.3 as having both a *name* and a *type*. While the name is certainly necessary to access the desired static field, is the type information really necessary? Recall that the accessed field field has declared type in the class defining the field
   The JVM is designed to support separate compilation of Java classes. A getstatic instruction may be issued for a static element of a class that has been modified and recompiled out-of-view of the code that is fetching the static element.
   We therefore could have two different views of the type of some static element: one at the instruction fetching the value, created when the fetching code was compiled, and another at the class providing the value, created when that code was compiled. The type of the static field is included in the getstatic instruction so that the actual and supposed type of the static element can be checked for compatibility.

   6. The getstatic instruction described in Section 10.2.3 requires that the static field's name be specified as an immediate operand of the instruction.

   Suppose the JVM instead had a getarbitrary instruction that could load a value from some arbitrary location. Instead of specifying the field by its name in an immediate operand, the location's address would be found at TOS. What are the implications of such an instruction on the performance and security of the JVM?
   If JVM had a getarbitrary instruction without any checks it could be used like a pointer in C and, e.g., read parts of a value such as the second half of a 64-bit long as a 32-bit int. But if the instruction has to be made secure JVM would need to check that the address is the start of a value, that the value matches the expected type, and that value is allowed to be read. So to keep guarantees to the getstatic instruction, the implementation of getarbitrary would have a higher complexity.

   10. A constructor call consumes the TOS reference to the class instance it should initialize. All constructors are void, so they do not return any kind of result. However, Java programs expect to obtain the result of the constructor call on TOS after the constructor has finished. By what JVM instruction sequence can this be accomplished?
   As constructors consume the reference put on the top of the stack by *new instruction,* the reference should be duplicated by dup before the constructor is executed.
   For more information see Stack Operations in Fisher Section 10.2.

3. Discuss the advantages and disadvantages of types in the JVM instruction set.
   Might be easier to implement statically typed languages on top of the JVM, since the JVM bytecode verifier provides type checking before executing code. This means that certain program errors that could otherwise lead to unintended behavior can be detected by the JVM at runtime. Implementing type-safe interpreters are made simpler by having typed instructions.

   On the other hand the JVM provides no way of encoding type-unsafe features of typical programming languages, such as pointers, immediate descriptors (tagged pointers), and unsafe type conversions. As such, it may be slightly more difficult to implement dynamically typed languages in the JVM(although "Da vinci Machine" project extended the JVM to better support dynamic languages. This fx. added instructions that allowed method invocation without type checking).
   https://www.cin.ufpe.br/~haskell/papers/Technical_Overview_of_the_Common_Language_Runtime-Meijer&Miller.pdf
   https://openjdk.java.net/projects/mlvm/

   Typed instructions introduce a large number of instructions that have identical behavior and differ only in their type. Furthermore, the JVM uses only one byte for opcodes. If every typed operation supported every type in jvm, the amount of instructions there would exceed what we could represent in a single byte. Instead the JVM instructions have a reduced level of support for certain types(less orthogonality). For a more thorough explanation, and a table showing what combination of operations and types are available see:
   https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.11.1

4. MS IL has similar properties to JVM instruction, but they are not all typed. Discuss why that is the case and what are the consequences.
   Having a single untyped instruction (e.g., add) instead of one instruction per type (e.g., iadd ladd fadd dadd) significantly reduces the number of instructions in the instruction set. However, having untyped instructions makes the implementation of an interpreter more difficult, as it must determine what operation to perform from its operands. This is not a problem for the CLR as all methods are compiled when executed, however, Oracle's HotSpot JVM starts by interpreting the bytecode and then compiles code executed repeatedly.
   For more information see:
   https://blog.overops.com/clr-vs-jvm-how-the-battle-between-net-and-java-extends-to-the-vm-level/
   https://www.oracle.com/technical-resources/articles/java/architect-evans-pt1.html
   https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process

Individual Exercises - Lecture 16

1. Read the articles under additional references.
   Can be found here:
   http://www.techrepublic.com/article/examine-class-files-with-the-javap-command/5815354
   http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javap.html

2. Do Fischer et al exercise 3, 9, 11 on pages 441-443 (exercise 3, 10, 11 on pages 473-476 in GE)

   3. Design an AST node that implements the CondTesting interface to accommodate an if statement with no else clause.

   (a) How do you satisfy the CondTesting interface while indicating that the if statement has no else clause?
   The simplest solution is to create a node with an empty else clause that generates no instructions for the else branch. A downside of this design is that it creates an unnecessary endLabel, since it should always skip the else clause.

   (b) How is the code-generation strategy presented in Section 11.3.9 affected by your design?
   A benefit of the above design is that the existing code-generation strategy can be used without any changes.

   9. Some languages offer iteration constructs like C's for statement, which aggregates the loop's initialization, termination, and repetition constructs into a single construct. Investigate the syntax and semantics of C's for statement. Develop an AST node and suitable interface to represent its components. Design and implement its code-generation visitor.

```
for (initializationStatement; testExpression; updateStatement)
{
   // statements inside the body of loop
}
```

   AST Node:
   ForLoopNode:
   children: initStatement, predicate, updateStatement, loopBody

Code Generation Procedure:

```
procedure visit(forLoopNode n) {
    doneLabel ← GenLabel()
    loopLabel ← GenLabel()

    n.getInitStatement().accept(this)
    call emitLabelDef(looplabel)

    n.predicate().accept(this)
    predicate ← n.predicate.getResultLocal()
    call emitBranchIfFalse(testExpressionResult, doneLabel)

    n.getLoopBody.accept(this)
    n.getUpdateStatement.accept(this)
    call emitBranch(loopLabel)

    call EmitLabelDef(doneLabel)
}
```

11. Section 11.3.10 generates code that emits the predicate test as instructions that occur before the loop body. Write a visitor method that generates the loop body's instructions first, but preserves the semantics of the WhileTesting node (the predicate must execute first).

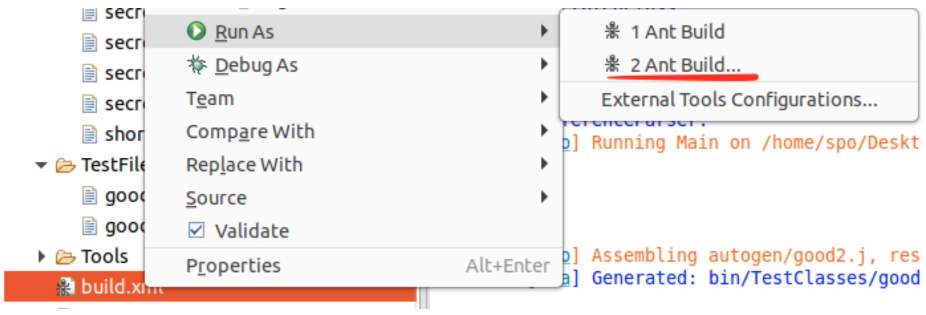
```
procedure visit(While n){
      loopBodyLabel ← GenLabel()
      predicateLabel ← GenLabel()

      call emitBranch(predicateLabel) // Jump to predicate
      call EmitLabelDef(loopBodyLabel)
      n.getLoopBody.accept(this)

      call EmitLabelDef(predicateLabel)
      n.getPredicate().accept(this)
      predicateResult ← n.getPredicate().getResultLocal()
      // If true, jump back to body
      emitBranchIfTrue(predicateResult, loopBodyLabel)
}
```

3. Investigate how if-then-else, while, switch and other constructs in Java are implemented by the Java compiler javac using javap
   Compiling a .java file using “javac filename.java” and then running “javap -v filename.class” gives you some information about how the java compiler handles your code.
   The table below shows the result from a java file with a simple while loop.

(Part of) Compiled Code	Original Java Code

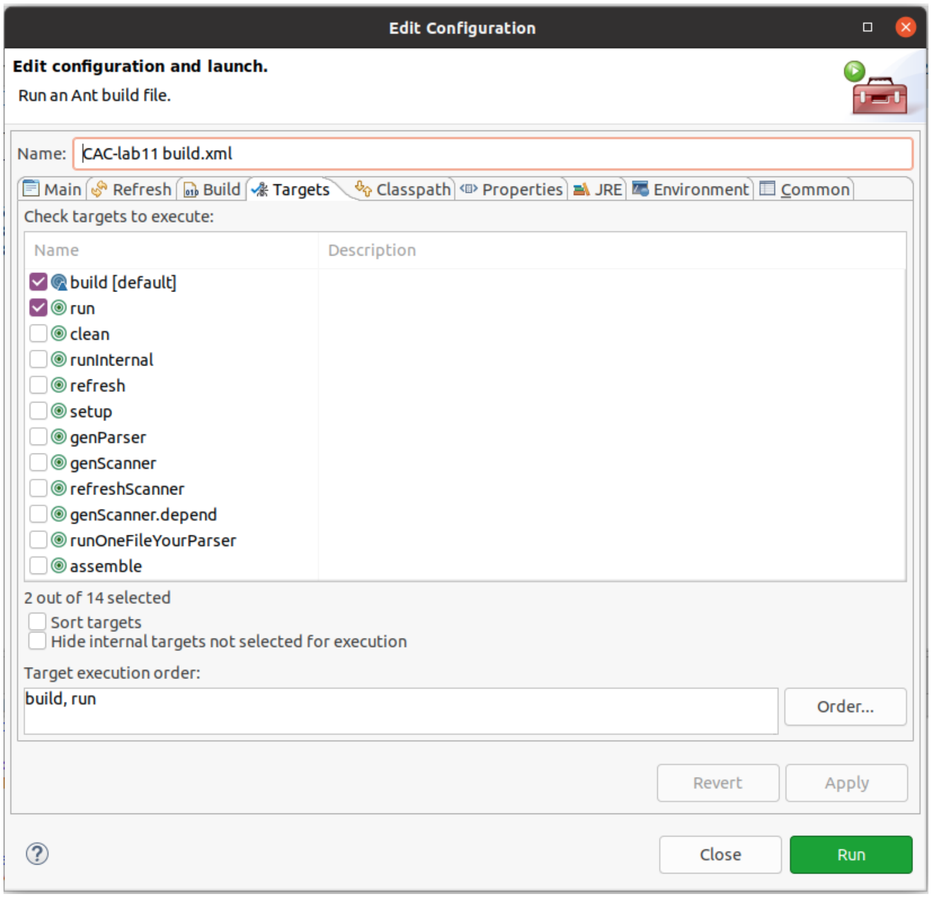
...
 0: iconst_0
 1: istore_0
 2: iconst_0
 3: istore_1
 4: iload_0
 5: iconst_5
 6: if_icmpge   19
 9: iload_1
 10: iload_0
 11: iadd
 12: istore_1
 13: iinc   0, 1
 16: goto   4
 19: iinc   1, 2
 22: return
...
	public class Program{
	    public static void main(){
	        int i = 0;
	        int a = 0;
	        while(i < 5){
	            a += i;
	            i += 1;
	        }
	        a += 2;
	    }
	}

   Note that in the bytecode the loop predicate check is inverted (‘>=’ instead of ‘<’). If the inverted predicate evaluates to true, the loop is exited by jumping to **19.** (Can you see why this inversion might be useful?)
4. (optional but recommended) Follow the Lab associated with Crafting a CompilerChapter 11: Code Generation for a Virtual Machine
   The materials for this exercise are provided in the virtual machine found on moodle.
   Running the main:
   Right click the ANT build and click the second Ant build…:



```
] Running Main on /home/spo/Deskt
] Assembling autogen/good2.j, res
] Generated: bin/TestClasses/good
```

   Check both ‘build’ and ‘run’ in the options. This will make sure that the code generator is first built and then run using the two files in the /TestFiles folder. Then click run.


Edit Configuration

Edit configuration and launch.

Run an Ant build file.

Name: CAC-lab11 build.xml

Main | Refresh | Build | Targets | Classpath | Properties | JRE | Environment | Common

Check targets to execute:

Name	Description
✓ build [default]	
✓ run	
☐ clean	
☐ runInternal	
☐ refresh	
☐ setup	
☐ genParser	
☐ genScanner	
☐ refreshScanner	
☐ genScanner.depend	
☐ runOneFileYourParser	
☐ assemble	

2 out of 14 selected

☐ Sort targets

☐ Hide internal targets not selected for execution

Target execution order:

build, run

The rest is explained in comments inside the ...src/submit/CodeGenVisitor.java.

NOTE: There is an error in the code provided by Fischer for this exercise. This prevents us from finding the main method. It can, however, be fixed by doing the following:

Replace the following code (inside /CAC-lab11/src/submit/CodeGenVisitor.java)

```
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, we don't handle
arrays");
    skip(2);
    emit(".method public static main([Ljava/lang/String;)V\n"
    + ".limit locals 1\n" + ".limit stack3\n"
    + "invokestatic " + c.getName() + "/main431()V\n"
    + "return\n" + ".end method\n\n");
    visitChildren((AbstractNode) c);
}
```

With this code:

```
public void visit(ClassDeclaring c) {
    emitComment("CSE431 automatically generated code file");
    emitComment("");
    emit(c, ".class public TestClasses/" + 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, we don't handle
arrays");
    skip(2);
    emit(".method public static main([Ljava/lang/String;)V\n"
    + ".limit locals 1\n" + ".limit stack3\n"
    + "invokestatic TestClasses/" + c.getName() +
    "/main431()V\n"
    + "return\n" + ".end method\n\n");
    visitChildren((AbstractNode) c);
}
```

5. (optional) In exercise 2 for Lecture 11 you created an AST for the little language defined in Figure 7.14 and in exercise 5, for lecture 14 you designed and implement a recursive interpreter for the language based on the Visitor Pattern. Design and implement a code generator visitor for the language generation JVM instructions (or rather Jasmin textual JVM programs)
Example of a generation method for the E + T case of the Expression rule:

```
visit(PlusExprNode plusNode) {
      // Emit code that will push value of expression to TOS
      plusNode.getLeftExpression().accept(this);
      // Emit code that will push the value the right term to TOS
      plusNode.getRightTern.accept(this);
      emit("iadd");
```

Group Exercises - Lecture 16

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Do Fischer et al exercise 5, 8, 4, 10 on pages 441-443 (exercise 4, 6, 7, 12 on pages 473-476 in GE)

   5. Languages like Java and C allow conditional execution among statements using the if construct. Those languages also allow conditional selection of an expression value using the ternary operator, which chooses one of two expressions based on the truth of a predicate. For example, the expression b ? 3 : 5 has value 3 if b is true; otherwise, it has value 5.

   (a) Can the ternary operator be represented by an CondTesting node in the AST? If not, design an appropriate AST node for representing the ternary operator.
   Yes, a ternary operator selects between two expressions similar to how if else statements selects between two block statements.

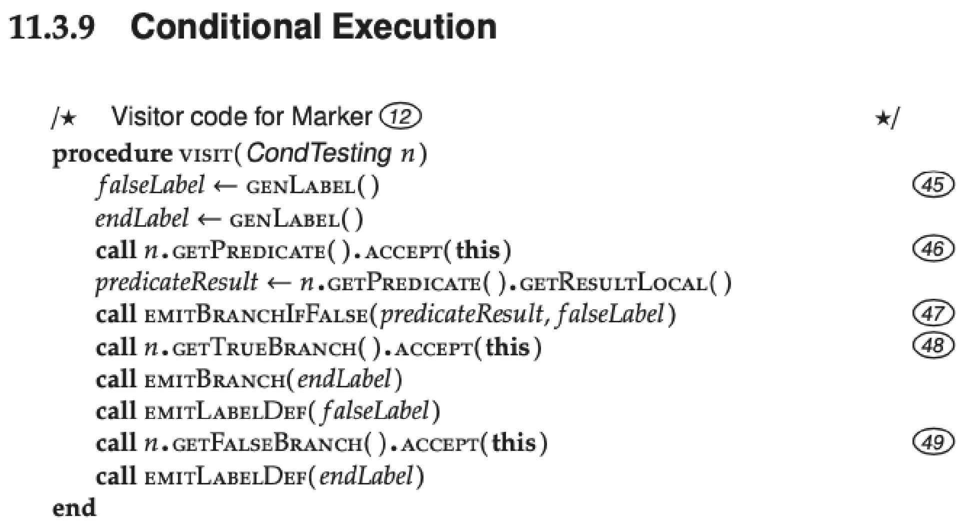
   (b) How does the treatment of the ternary operator differ from the treatment of an if statement during semantic analysis?
   In most languages the ternary operator yields a value (ie. it is an expression), while an if-statement is just a statement.
   For this reason we must do type checking on the ternary operator to verify that the two branches return the same type, or atleast, type that can be coerced to the same type.
   Be aware that some languages (e.g., Scala and Rust) also treat if-else as expressions.

   (c) How does code generation differ for the two constructs?
   The main difference is, that code generation for the ternary expression must leave the value of the executed expression on TOS, whereas the if/else statement does not.

8. The semantics of a WhileTesting node accommodate Java and C's while constructs. Languages like Java have other syntax for iteration, such as Java's do-while construct. The body of the loop is executed, and the predicate is then tested. If the predicate is true, then the body is executed again. Execution continues to loop in this manner until the predicate becomes false. The sense of the predicate is consistent, in that iteration ceases when the predicate test is false. However, the do-while construct executes the loop body before the test. The CondTesting node represents constructs where the predicate is tested before the loop body is executed.

How would you accommodate the do-while statement? What changes are necessary across a compiler's phases, from syntax analysis to code generation?
The node could be annotated with a ConditionTestTime flag that signifies if the test is supposed to happen before or after the loop iteration.
When generating the code, you emit the loop label at the start of the loop body, and the conditional jump to the loop label at the end of the loop body.

4. Consider the code-generation strategy presented in Section 11.3.9. A conditional jump to false Label is taken, when the predicate is false, to skip over code that should execute only when the predicate is true. Rewrite the code-generation strategy so that the conditional jump is taken when the predicate is true, to skip over code that should execute only when the predicate is false.



11.3.9 Conditional Execution

```
/★  Visitor code for Marker ⑫                                        ★/
procedure visit( CondTesting n )
    falseLabel ← genLabel( )                                          ㊺
    endLabel ← genLabel( )
    call n.getPredicate( ).accept( this )                             ㊻
    predicateResult ← n.getPredicate( ).getResultLocal( )
    call emitBranchIfFalse( predicateResult, falseLabel )             ㊼
    call n.getTrueBranch( ).accept( this )                            ㊽
    call emitBranch( endLabel )
    call emitLabelDef( falseLabel )
    call n.getFalseBranch( ).accept( this )                           ㊾
    call emitLabelDef( endLabel )
end
```

```
procedure visit(CondTesting n)
        trueLabel ← genLabel()
        endLabel ← genLabel()
        call n.getPredicate().accept(this)
        predicateResult ← n.getPredicate() .getResultLocal()
        call emitBranchIfTrue(predicateResult, trueLabel)
        call n.getFalseBranch() .accept(this)
        call emitBranch(endLabel)
        call emitLabelDef( trueLabel )
        call n.getTrueBranch().accept(this)
        call emitLabelDef(endLabel)
end
```

10. Later versions of Java offer the so-called “enhanced for” statement. Investigate the syntax and semantics of that statement. Develop an AST node and suitable interface to represent the statement. Design and implement its code-generation visitor.

An enhanced for loop is an adaptation of a normal for loop for cases where the conditional check is trivial, i.e. looping over a collection, so its a for loop where the condition is implicit and not user accessible. Example:

```
List<Integers> integers = ...
for (Integer number: integers) { System.out.println(number) }
```

A common implementation technique is for a type to implement a specific interface so that it can return an iterator over its elements (e.g., Java’s iterable) which allows multiple iterations. Using this approach, the AST node proposed for loops in Section 11.3.10 can be reused with the predicate being an iterable type. For code generation an iterator for the type is retrieved using the iterable interface at the start of the loop, then at each iteration it is verified that the iterator has more elements using the hasNext method before next is called to retrieve an element which is assigned to the symbol used in the foreach loop. At the code generation level, the predicate condition would be generated based on the implementation of hasNext() method in the iterator code.

Java Overview: https://www.journaldev.com/13460/java-iterator
Java Iterable: https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html
Java Iterator: https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

3. Discuss and sketch a code generator for your language for the JVM
   How do you translate your types into the ones for the JVM? How would some of your language constructs look in java bytecode, for example an if-stmt?

Individual Exercises - Lecture 17

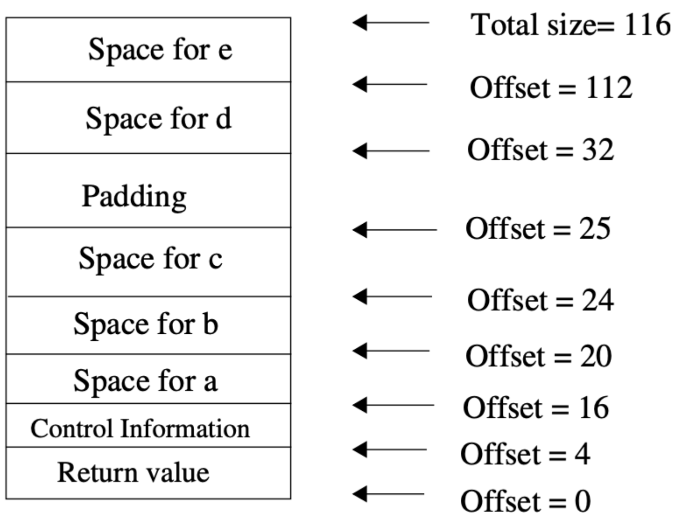
1. Read the articles under additional references
   Can be found here:
   http://www3.nd.edu/~dthain/courses/cse40243/spring2006/gc-survey.pdf
   www.memorymanagement.org
   https://www.memorymanagement.org/mmref/begin.html
   https://www.c-sharpcorner.com/article/memory-management-in-net/
   https://www.artima.com/insidejvm/applets/HeapOfFish.html
   https://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf

2. Do Fischer et al exercise 1, 3, 4, 5, 9, 12, 15, 20 on pages 482- 488 (exercise 2, 3, 4, 6, 11, 13, 16, 19 on pages 514-520 in GE)

   1. Show the frame layout corresponding to the following C function:

```
int f(int a, char *b) {
   char c;
   double d[10];
   float e;
   ...
}
```

   Assume control information requires 3 words and that f's return value is left on the stack. Be sure to show the offset of each local variable in the frame and be sure to provide for proper alignment (integers and floats on word boundaries and doubles on doubleword boundaries)



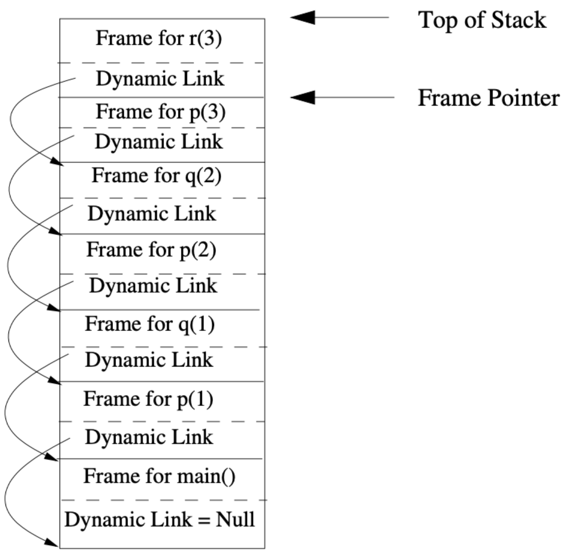
Note that there are seven bytes of padding following variable c because the array of doubles, d, requires double word alignment. The size of the frame could be reduced by placing the float e *before* d. Such reordering is rarely done because memory space currently is very cheap and programmers sometimes want to be able to anticipate where variables will be placed (for debugging and cache optimization).

On the other hand, being able to predict where variables will be placed in a frame makes it far easier to violate security rules. Hackers often access vital runtime data (like return address or heap pointers) by overwriting valid values from adjacent fields. Making data layout in a frame unpredictable enhances overall security.

3. Using the code below, show the sequence of frames, with dynamic links, on the stack when r(3) is executed assuming we start execution (as usual) with a call to main().

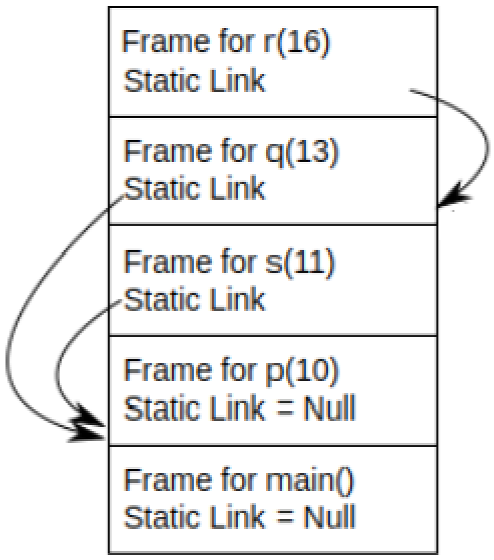
```
r(flag){
   printf("Here !!!\n"); }
q(flag){
   p(flag+1); }
p(int flag){
   switch(flag){
      case 1: q(flag);
      case 2: q(flag);
      case 3: r(flag); }

main(){
   p(1);
}
```

The snapshot of frames when r(3) executes is shown below:



4. Consider the following C-like program that allows subprograms to nest. Show the sequence of frames, with static links, on the stack when r(16) is executed assuming we start execution (as usual) with a call to main(). Explain how the values of a, b, and c are accessed in r's print statement.

```
p(int a){
   q(int b){
      r(int c){
         print(a+b+c);
      }
      r(b+3);
   }

   s(int d){
      q(d+2);
   }

   s(a+1);
}
main(){
   p(10);
}
```



The frame for r(16) has access to 'c' through itself. Access to 'b' can be obtained by following the static link to the frame of q(13). 'a' can be found by following static links to frame q(13) and from there to the frame of p(10).

5. Reconsider the C-like program shown in Exercise 4, this time assuming display registers are used to access frames (rather than static links). Explain how the values of a, b, and c are accessed in r's print statement.
Display registers are described in section 12.2.4. For this sequence of calls we would need a total of 3 display registers(because the deepest nesting level is three).

Again, accessing c is trivial as it is stored in the display of r(16) (d2).
Looking at d1 we get a reference to the display of method call q(13) which gives us access to the 'b' variable. Finally we look at d0, which references the display of the method call p(10). Here we have access to the 'a' variable.

9. Assume that in C we have the declaration int a[5][10][20], where a is allocated at address 1000. What is the address of a[i][j][k] assuming a is allocated in row-major order? What is the address of a[i][j][k] assuming a is allocated in column-major order?

Row major order:
Given a cube with the following dimension: i_length, j_length, k_length, then the offset of a given index can be calculated as the following:
$OffsetRowMajor(i, j, k) \ = \ i * j_{length} * k_{length} + j * k_{length} + k \ = (i * j_{length} + \ j) * k_{length} + k$

For this example we have that i_length = 5, j_length = 10, k_length = 20, then the offset of the index 2, 1, 2 (i = 2, j = 1, k = 2) can be calculated in the following way:
$Offset(2, 1, 2) \ = (2 * 10 + \ 1) * 20 + 2 \ = 422$
$MemoryAddressRowMajor(2, 1, 2) \ = \ 1000 \ + \ OffsetRowMajor(2, 1, 2) \ = \ 1422$

Column major order:
The offset using column major order can be calculated in the following way:
$OffsetColMajor(i, j, k) \ = i + j * i_{length} + k * i_{length} * j_{length}$

Using the same example as before (2,1,2 or i = 2, j = 1, k = 2) we get the following allocation address:
$OffsetColMajor(2, 1, 2) \ = 2 + 1 * 5 + 2 * 5 * 10 \ = \ 107$
$MemoryAddressColMajor(2, 1, 2) \ = \ 1000 \ + \ OffsetColMajor(2, 1, 2) \ = \ 1107$

For an in-depth explanation of row-major and column-major order for multi-dimensional arrays see:
https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

12. Assume we add a new option to C++ arrays that are heap-allocated, the flex option. A flex array is automatically expanded in size if an index beyond the array's current upper limit is accessed. Thus we might see:

```
ar = new flex int[10];
ar[20] = 10;
```

The assignment to position 20 in ar forces an expansion of ar's heap allocation. Explain what changes would be needed in array accessing to implement flex arrays. What should happen if an array position beyond an array's current upper limit is read rather than written?

As the amount of memory currently allocated for the array can only hold 10 values, a new block of memory must be allocated from the heap. Then all elements from the original array must be copied to the new array and the new element 10 written to location 20. When allocating a block of memory for the new array it is often better to grow the array with a certain factor (e.g., 2x) instead of just making room for the new elements to reduce the number of times the array must be resized. In Java ArrayList implements an array that dynamically resizes when elements are added, the implementation used for OpenJDK 8 can be found at [1].

The semantics for reading values from outside the bounds of a flex array is much less clear. Two safe options would be to return a default value for the type stored in the array (e.g., return 0 for int) or raise an error to inform the programmer that the index provided is outside the bounds of the array. An unsafe options would be to resize the array and then return the "garbage" value at the provided index (if all values are initialized this is not a problem).

[1]
http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/ArrayList.java

15. Assume we organize a heap using reference counts. What operations must be done when a pointer to a heap object is assigned? What operations must be done when a scope is opened and closed?

When reassigning a variable, the reference count of the heap reference being overwritten is decreased by one and the reference count of the heap reference being assigned is increased by one. Source and target locations containing null are treated as special cases.

When a scope is opened, all heap references not explicitly initialized are set to null or some special "error value." When a scope is closed, all heap references that are lost are treated as if they had been assigned null (thereby decreasing reference counts to account for the lost heap references).

20. The second phase of a mark-sweep garbage collector is the sweep phase, in which all unmarked heap objects are returned to the free space list. Detail the actions

needed to step through the heap, examining each object and identifying those that have not been marked (and hence are garbage).

Assume that the first word of each object is used to hold the length of the object and the sign bit is used for marking. Then returning objects to the free list is a simple run through the heap from its start address looking at the sign bit to determine if the objects should go in the list. Next object is found by adding the length and the current object to the address of the current object.

3. Do Sebesta exercise 7 and 8 page 468

   7. It is stated in this chapter that when nonlocal variables are accessed in a dynamic-scoped language using the dynamic chain, variable names must be stored in the activation records with the values. If this were actually done, every nonlocal access would require a sequence of costly string comparisons on names. Design an alternative to these string comparisons that would be faster.
   Use string mapping to replace variable names with integer placeholders that can be accessed in a linear look up time, with a hashtable. This increases performance, since integer comparison is cheap.

   8. Pascal allows gotos with nonlocal targets. How could such statements be handled if static chains were used for nonlocal variable access? Hint: Consider the way the correct activation record instance of the static parent of a newly enacted procedure is found (see Section 10.4.2).

   Following the hint stated with the question, the target of every goto in a program could be represented as an address and a nesting_depth, where the nesting_depth is the difference between the nesting level of the procedure that contains the goto and that of the procedure containing the target. Then, when a goto is executed, the static chain is followed by the number of links indicated in the nesting_depth of the goto target. The stack top pointer is reset to the top of the activation record at the end of the chain.

Group Exercises - Lecture 17

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Do Fischer et al exercise 2, 7, 11, 10, 13, 16, 19, 25, 24 on pages 482-488 (exercise 1, 8, 9, 10, 12, 15, 18, 24, 25 on pages 514-520 in GE)

   2. Local variables are normally allocated within a frame, providing for automatic allocation and deallocation when a frame is pushed and popped. Under what circumstance must a local variable be dynamically allocated? Are there any advantages to allocating a local variable statically (i.e., giving it a single fixed address)? Under what circumstances is static allocation for a local permissible?

   The variable must be dynamically allocated whenever the corresponding function/method/procedure can be activated more than once, i.e. there is more than one activation record for the function/method/procedure on the stack (think e.g. of a recursive function).

   Some languages allow the programmer to explicitly specify that a local function variable should be static, which causes the variable to persist across all calls of the function.

   Local constants can also be statically allocated.

   7. Although the first release of Java did not allow classes to nest, subsequent releases did. This introduced problems of nested access to objects, similar to those found when subprograms are allowed to nest. Consider the following Java class definition:

```java
class Test {
   class Local {
      int b;
      int v(){ return a+b; }
      Local(int val){ b=val; }
   }

   int a = 456;

   void m(){
      Local temp = new Local(123);
      int c = temp.v();
   }
}
```

   Note that method v() of class Local has access to field a of class Test

as well as field b of class Local. However, when temp.v() is called, it
is given a direct reference only to temp. Suggest a variant of static links
that can be used to implement nested classes so that access to all visible
objects is provided.

The problem is that the frame produced from the call to .v() has only a static reference to temp (the instance of the 'Local' class). From this we have no reference to variable 'a' from the test class instance.

However, nested class instances can only exist if their outer class is also instantiated. So to solve our problem: when creating an instance of the nested class(using the new keyword), we simply include a pointer(static link) to the outer class.
Now when .v() is called, it will contain a static link to the instance of the Local class, which then again contains a static link to the 'Test' class instance.

11. In Java, subscript validity checking is mandatory. Explain what changes would be needed in C or C++ (your choice) to implement subscript validity checking. Be sure to address the fact that pointers are routinely used to access array elements. Thus you should be able to check array accesses that are done through pointers, including pointers that have been incremented or decremented.

This would require us to know how an array is represented in memory and being able to check that a pointer still points to the array elements, e.g. by going from the current element back to the start of the array, where the size is kept.

First, arrays in C should be made first-class types that consist of a pointer and a size. This would allow arrays to be both passed to/from functions and have their bounds checked as their size is always available. Both explicit and implicit coercion of arrays to pointers should not be allowed. Pointer arithmetic should also be disallowed in order to prevent data from being accessed outside the bounds of an array. However, many existing programs would have to be updated to account for these changes. Adding bounds-checking to C has been a focus for much research with approaches suggested that are backwards compatible [1] and some that extend C [2].

[1] Dinakar Dhurjati and Vikram Adve, Backwards-compatible array bounds checking for C with very low overhead, https://dl.acm.org/doi/10.1145/1134285.1134309

[2] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks, Achieving Safety Incrementally with Checked C, https://rd.springer.com/chapter/10.1007/978-3-030-17138-4_4

10. Most programming languages (including Pascal, Ada, C, and C++) allocate global aggregates (records, arrays, structs, and classes) statically, while local aggregates are allocated within a frame. Java, on the other hand, allocates all aggregates in the heap. Access to them is via object references allocated statically or within a frame. Is it less efficient to access an aggregate in Java because of its mandatory heap allocation? Are there any advantages to forcing all aggregates to be uniformly allocated in the heap?

Access to a heap object requires a heap validity check and an indirection. If the object is accessed frequently, these overheads can probably be optimized away (as redundant or loop-invariant operations). If the heap manager does not compact the heap, a program may find active objects spread among inactive heap allocations, increasing the program's "footprint" in memory.

An advantage of allocating all aggregates in the heap is that the size of the aggregate can be part of its value rather than its declaration. Java arrays can change their size during execution. This can be very useful when no fixed size bounds are apparent (e.g., a character buffer being edited, or an array being sorted). On the other hand, sometimes it is useful to fix the size of an array and allocate it as efficiently as possible (e.g., an array representing a chess board). C# has added fixed-size arrays (that may be allocated statically or within a frame) as a language extension. (Java-style arrays are of course retained.)

13. Fortran library subprograms are often called from other programming languages. Fortran assumes that multidimensional arrays are stored in column-major order; most other languages assume row-major order. What must be done if a C program (which uses row-major order) passes a multidimensional array to a Fortran subprogram. What if a Java method, which stores multidimensional arrays as arrays of array object references, passes such an array to a Fortran subprogram?

When calling a Fortran library from C the programmer must manually reorganize the multidimensional array so the values are stored in column-major as expected by Fortran. When combining C and Fortran code it is recommended to perform all of the array operations in either C or Fortran to remove the problem. When calling Fortran from Java a multidimensional array in column-major order must be manually constructed where the references are replaced with the actual data and passed it to Fortran.

16. Some languages, including C and C++, contain an operation that creates a pointer to a data object. That is, p = &x takes the address of object x, whose type is t, and assigns it to p, whose type is t*.
How is management of the runtime stack complicated if it is possible to create pointers to arbitrary data objects in frames? What restrictions on the creation and copying of pointers to data objects suffice to guarantee the integrity of the runtime stack?

Creating a pointer to a location in a frame is very dangerous as the pointer may be used *after* the frame is popped from the stack. Hence modern successors to C and C++, like Java and C♯, disallow this operation.

There are two approaches to handling pointers to frame locations. We may add the rule that a pointer to a frame location may never be assigned to a variable with a longer lifetime than the frame itself. This is a form of *escape analysis* that tracks where a frame pointer may be assigned.

An alternative, taken by some functional languages like ML, is to allocate frames in the heap rather than on a stack. Now the rules of garbage collection apply; when no pointers to a frame remain, the frame is collected. As long as a pointer remains, the frame is protected from garbage collection. It may appear that using the heap to hold frames is needlessly inefficient, but this need not be the case. Modern *copying collectors* (Section 12.4.3 on page 472) have the property that their cost is controlled by the number of *live* heap objects. Dead objects, like frames for completed calls, are essentially "free."

19. In a strongly typed language such as Java, all variables and fields have a fixed type known at compile-time. What runtime data structures are needed in Java to implement the mark phase of a mark-sweep garbage collector in which all accessible ("live") heap objects are marked?

We must trace all pointers (references) found in global, stack and heap space. For global pointers we can simply have a list of global addresses known (from their declarations) to hold heap pointers. To save space, a *bit map* might be used. In a bit map, each bit represents one word of global data space. A one bit means the word is a pointer and a zero says it is not.

To find pointers within a frame, we start with the current frame pointer. Each frame, in its control information, will contain a method code identifying the method the frame corresponds to. Each method code will index a list (or bitmap) of frame offsets known to hold heap pointers.

Each heap allocation has a header word. This header contains a type code identifying the class the heap object implements. We use this type code to index a list of offsets within the object that contain pointers. For array objects that contain pointers, we can extract the size of the array and where within the object the array elements begin.

25. An unattractive aspect of both mark-sweep and copying garbage collection is that they are batch-oriented. That is, they assume that periodically a computation can be stopped while garbage is identified and collected. In interactive or real-time programs, pauses can be quite undesirable. An attractive alternative is concurrent garbage collection in which a garbage collection process runs concurrently with a program.

Consider both mark-sweep and copying garbage collectors. What phases of each can be run concurrently while a program is executing (that is, while the program is changing pointers and allocating heap objects)? What changes to garbage collection algorithms can facilitate concurrent garbage collection?

Mark-sweep: The Mark sweep method does not immediately allow for concurrent garbage collection. The releasing of heapvars to the freelist could, however, possibly be done concurrently in the following manner. First halt the program, mark all vars without any references. Then resume execution. Now a concurrent process could run, that cleans the marked variables and puts them back into the free list.

Copying garbage collectors: The problem with running the copying GC concurrently, is the possibility of copying memory that is being modified by the running program. Modifications to the GC algorithm are therefore required in order to safely run concurrently. Take a look at the additional references below.

For more information on GC in real time systems, take a look at:

https://en.wikipedia.org/wiki/Tracing_garbage_collection#Tri-color_marking

https://www.drdobbs.com/jvm/java-garbage-collection-for-real-time-sy/184410684

https://www.ibm.com/developerworks/library/j-rtj4/

24. Copying garbage collection can be improved by identifying long-lived heap objects and allocating them in an area of the heap that is not collected.

What compile-time analyses can be done to identify heap objects that will be long lived? At runtime, how can we efficiently estimate the “age” of a heap object (so that long-lived heap objects can be specially treated)

Compile Time: Possibly follow references from the Main class and somehow estimate how long each reference lives. For example, objects allocated and stored on the Main object will probably live for the majority of the program.

Runtime: copying garbage collectors can be extended with generations. Objects are allocated in the youngest generation and after they have survived garbage collection *n* times they are moved to the next generation. Later generations are garbage collected less frequently, and the oldest generation might not even be garbage collected. For more information see Generational Garbage Collection in Fisher Section 12.4.

3. What are the design issues for pointer types?
   The primary design issues particular to pointers are the following:
   1. What are the scope and lifetime of a pointer variable?
   2. What is the lifetime of a heap-dynamic variable (the value a pointer references)?
   3. Are pointers restricted as to the type of value to which they can point?
   4. Are pointers used for dynamic storage management, indirect addressing, or both?
   5. Should the language support pointer types, reference types, or both?

   For more detail see Sebesta Section 6.11

4. What are the two most common problems with pointers?
   Dangling Pointer: a pointer that is pointing to a location in memory that has been deallocated, using this pointer is dangerous as the memory might have been allocated for some other data, possibly even another data type.

   Lost Heap-Dynamic Variable: if data is allocated on the heap but the pointer to the data is lost, the program leaks memory as the data cannot be deallocated by the program.

   Also, Reading from uninitialized pointers provides either “garbage” values or causes a segmentation fault. Use of pointers can also make the program harder to read and comprehend.

   For more detail see Sebesta Section 6.11

5. Why are pointers in most languages restricted to pointing to a single type variable?
   A pointer points to a specific place in memory. When dereferencing the pointer to get the value, the compiler needs to know how many bytes to read from the starting address. Some types have different lengths i.e 32 and 64 bit integers. So restricting a pointer to a single type ensures that the correct memory amount of memory is read. Consider if there were no types in the declaration of a pointer in C. The compiler would be able to perform some type inference for some simple programs, however, it would not be able to detect all types behind a pointer. The compiler would therefore not know how much memory to return. A solution to this would be to force the programmer to decorate all dereferencing of pointers with a type or number of bytes to read but that would introduce a lot of other potential problems.

6. Sebesta review questions 35, 36 page 321
   35. Why are reference variables in C++ better than pointers for formal parameters?

   Reference variables used as formal parameters in C++ always reference a value and are implicitly dereferenced unlike pointers. Resulting in safer and more readable code.

   See also:
   https://en.wikipedia.org/wiki/Reference_(C%2B%2B)#Relationship_to_pointers

   36. What advantages do Java and C# reference type variables have over the pointers in other languages?

   Java class instances are referenced with reference variables. Since Java instances are implicitly deallocated, there cannot be dangling references in Java. C# follows a similar design, however it includes pointers when using the unsafe keyword. If a object is pointed to by a pointer it will not be implicitly deallocated in C#

7. Sebesta problem set 13, 14 page 323 (in 14 discuss rather than write)
   13. Write a short discussion of what was lost and what was gained in Java's designers' decision to not include the pointers of C++.

   Compared to C++, Java provides a more limited interface for reading from and writing to memory by not including pointers, e.g., you cannot read data from a Java array of longs at an arbitrary byte offset. As such, the lack of pointers limits what can be expressed in Java to a small degree, on the other hand, it is much more difficult for the programmer to make a mistake when reading and writing to memory. In summary, it is a tradeoff between expressibility (e.g, allowing optimizations) and safety.

   14. What are the arguments for and against Java's implicit heap storage recovery, when compared with the explicit heap storage recovery required in C++? Consider real-time systems.

   Like stated for Problem Set 13, Java prioriotizes memory safety to a higher degree than C++ and therefore uses garbage collection to manage heap memory while C++ requires that the programmer have to manually deallocate memory through RAII (Resource Acquisition Is Initialization) and smart pointers like std::unique_ptr and std::shared_ptr (reference counting). By automating memory management a large collection of memory related bugs can be removed, e.g, Microsoft have stated that ~70% of the vulnerabilities they assign a CVE (Common vulnerabilities and Exposures) each year are memory safety issues. However, garbage collection comes at a cost as the programmer has no control over when it runs. This uncertainty is a problem for real-time systems where guarantees on execution time is required, e.g., the embedded system controlling a car's airbag. However, research into using Java for such applications has been performed [2].

[1]
https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/

[2]
https://vbn.aau.dk/da/publications/hvmtp-a-time-predictable-and-portable-java-virtual-machine-for-ha

8. What is the expected level of use of pointers in C#? How often have you used pointers in C#?
   Raw pointers are almost never used in C#, e.g., they might be used for interfacing with existing C or C++ code, however, references to classes are pointers in themselves.

Individual Exercises - Lecture 18

1. Read the following additional references.
   Can be found here:
   http://www.itu.dk/~sestoft/papers/numericperformance.pdf

2. Fischer et. al. exercises 1 (you may want to read Seftoft's note first), 6, 9 (sketch rather than implement - NOTE CG should be TreeCG), 10, 19, 21 538- 546 on pages (exercises 2 (you may want to read Seftoft's note first), 6, 8 (sketch rather than implement), 10, 20, 21 on pages 570-578 in GE)

   1. Consider the following Java method:

```
public static int fact(int n){
   if (n == 0)
      return 1;
   else return n*fact(n-1); }
```

Using your favorite Java compiler, show the JVM bytecodes that would be generated for this method. Explain what each of the generated bytecodes contributes to the execution of the methods.

If the "public static" prefix is removed, the Java method becomes a valid C or C++ function. Compile it using your favorite compiler your favorite processor using no optimization. List the machine instructions generated and show which machine instructions correspond to each generated JVM bytecode.

The bytecodes generated by the Java compiler supplied as part of the Java Development Kit, JDK (version 1.6.0:20) are:

```
0:    iload_0
1:    ifne    6
4:    iconst_1
5:    ireturn
6:    iload_0
7:    iload_0
8:    iconst_1
9:    isub
10:   invokestatic    #2; //Method fact:(I)I
13:   imul
14:   ireturn
```

The integers in the left column are byte offsets in the method (used as target addresses for branch instructions). The iload at offset 0 pushes the contents of local variable 0 (n) onto the operand stack. If it is not equal to 0 (ifne), control is transferred to offset 6. Otherwise, 1 is pushed onto the operand stack, and an integer return (ireturn) is executed.

If n was not equal to 0 execution continues at offset 6. n is pushed twice, followed by the integer 1. The isub at offset 9 computes n-1. The invokestatic calls the method whose signature is at offset 2 in the constant pool. This is the string fact:(I)I which represents a method named fact that takes an integer parameter and returns an integer result.

The return value from the call is left at the top of the operand stack. The value of n is below it. The imul at offset 13 computes n*fact(n-1). The ireturn at offset 14 returns this value to the caller.

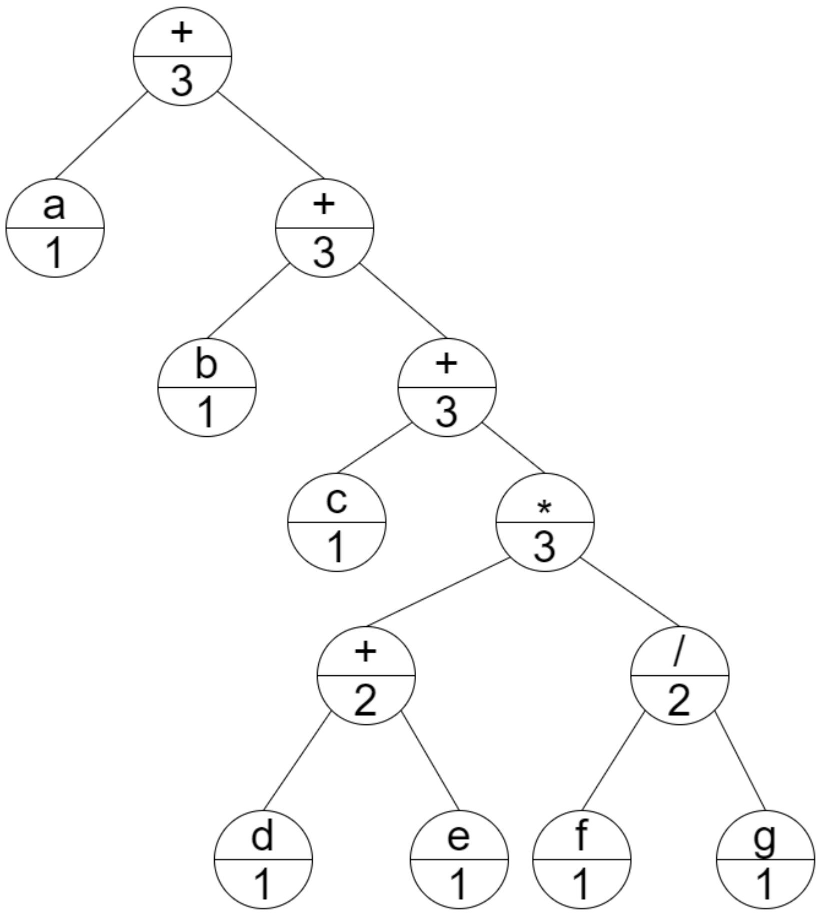
The MIPS code generated by gcc version 3.4.4 is:

```
fact:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
        move    $fp,$sp
        sw      $4,32($fp)
        lw      $2,32($fp)
        nop
        bne     $2,$0,$L2
        nop

        li      $2,1
        sw      $2,16($fp)
        j       $L1
        nop

$L2:
        lw      $2,32($fp)
        nop
        addiu   $2,$2,-1
        move    $4,$2
        jal     fact
        nop

        lw      $3,32($fp)
        nop
        mult    $2,$3
        mflo    $2
        sw      $2,16($fp)
$L1:
        lw      $2,16($fp)
        move    $sp,$fp
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
        j       $31
        nop
```

The initial instructions, from addiu to the store of register 4 (the parameter register) set up and link fact's frame. Hence they are actually part of the invokestatic code executed by fact's caller. The load into register 2 corresponds to the bytecode at offset 0. The bne corresponds to the ifne at offset 1. The li (load immediate) of 1 into register 2 corresponds to the iconst 1 at offset 4. The sw of register 2 into the frame, followed by the jump to $L1 and the code at $L1 implement the ireturn instruction at offset 5.

At $L2 the else part in fact is implemented. The lw loads n into register 2. The addiu computes n-1 into register 2, which is then moved into register 4 (the parameter register). The jal is a call instruction, corresponding to the invokestatic at offset 10. Next, n is loaded into register 3 (corresponding to the iload 0 at offset 6). The mult, mflo pair compute n*fact(n-1), corresponding to the imul at offset 13. Finally, the sw and the code at $L1 implement the ireturn instruction at offset 14.

6. Show the expression tree, with registerNeeds labeling, that corresponds to the expression a+(b+(c+((d+e)*(f/g)))).



Show the code that would be generated using the treeCG code generator.

```
a+(b+(c+((d+e)*(f/g))))
```

```
lw $10, f             # Load f into register 10
lw $11, g             # Load g into register 11
div $10, $10, $11     # Compute f / g into register 10
lw $11, d             # Load d into register 11
lw $12, e             # Load e into register 12
add $11, $11, $12     # Compute d + e into register 11
mul $10, $10, $11     # Compute (d + e) * (f / g) into register 10
lw $11, c             # Load c into register 11
add $10, $10, $11     # Compute c + ... into register 10
lw $11, b             # Load b into register 11
add $10, $10, $11     # Compute b + ... into register 10
lw $11, a             # Load a into register 11
add $10, $10, $11     # Compute a + ... into register 10
```

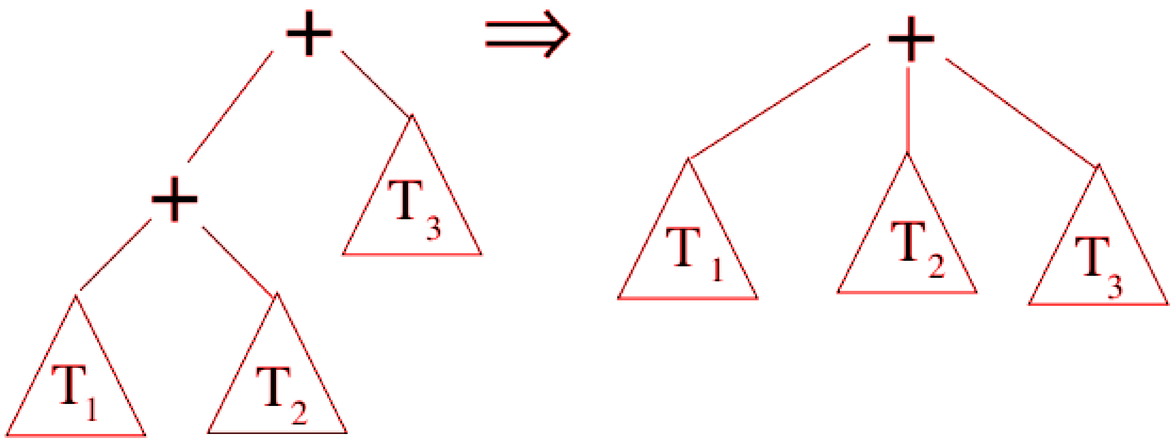
9. Sometimes the code generated for an expression tree can be improved if the associative property of operators like + and * is exploited. For example, if the following expression is translated using treeCG, four registers will be needed:

```
(a+b) * (c+d) * ((e+f) / (g-h))
```

Even if the commutativity of + and * is exploited, four registers are still required. However, if the associativity of multiplication is exploited to evaluate multiplicands from right to left, then only three registers are needed. First ((e+f)/(g-h)) is evaluated, then (c+d)*((e+f)/(g-h)), and finally (a+b)*(c+d)*((e+f)/(g-h)).

Write a routine associate that reorders the operands of associative operations to reduce register needs. (Hint: Allow associative operators to have more than two operands.)

We first restructure the expression tree so that all operands of an associative operator are visible. To do this, if an associative operator, op, has one or more subtrees also rooted by op, we move the children of the subtrees rooted by op up to the level of the parent operator. This is illustrated below:



Now we must handle the case in which an operator has more than 2 children. Assume the greatest number of registers needed by any subtree is n. Obviously we will need at least n registers to evaluate the operator and its children. Moreover, we will never need more than n + 1 registers to do the evaluation. We can choose any subtree and evaluate it, using no more than n registers. Its result can be held in an additional register. The next subtree is evaluated, using no more than n registers. It is combined, with the previous result, reusing that result's register. This process is continued until all subtrees are evaluated and combined.

How do we decide if n or n + 1 registers are required for a particular expression tree? The key is to select the two subtrees that require the greatest number of registers to evaluate. Say these values are n1 and n2, where n1 ≥ n2. If n1 > n2 then only n1 registers are needed. We evaluate the subtree needing n1 registers first, leaving its results in a single register. The remaining n1 − 1 registers are sufficient to evaluate all remaining subtrees, with partial results always held in the same result register.

If n1 = n2 then n1 +1 registers are needed. One or the other of the two most expensive subtrees must be evaluated first. Its result, perhaps combined with other subexpression values, must be held in a register. Thus when the other of the two most expensive subtrees is evaluated n1 + 1 registers will be needed.

In our example, after restructuring the root operator * will have three subtrees. ((e+f)/(g-h)) requires 3 registers, while (a+b) and (c+d) each require 2 registers. Thus n1 = 3 and n2 = 2, so we conclude 3 registers suffice if the associativity of * is exploited.

10. In Section 13.4 we saw that many modern architectures are delayed load. That is, a value loaded into a register may not be used in the next instruction; a delay of one or more instructions is imposed (to allow time to access the cache).

The treeCG routine of Section 13.2 is not designed to handle delayed loads. Hence, it almost always generates instruction sequences that stall at selected loads.

Show that if an instruction sequence (of length 4 or more) generated by treeCG is given an additional register, it is possible to reorder the generated instructions to avoid all stalls for a processor with a one instruction load delay. (It will be necessary to reassign the register used by some operands to utilize the extra register.)

Example Expression: a + (b + c)

```
lw $10, c          # Load c into register 10

lw $11, b          # Load b into register 11

lw $12, a          # Load a into register 12

add $10, $10, $11  # Compute b + c into register 10

add $10, $10, $12  # Compute a + (b + c) into register 10
```

By using the extra register to preemptively load another value into a register that is needed in the computation directly following the current computation we ensure that the value is fully loaded into the register instead of having to wait for the load to finish between the two add instructions.

For a more detailed solution see page 278 to 286 in: http://pages.cs.wisc.edu/~fischer/cs701.f14/lectures/L.All.pdf

19. It is sometimes the case that we need to schedule a small block of code that forms the body of a frequently executed loop. For example

```
for (i=2; a < 1000000; i++)
   a[i] = a[i-1]*a[i-2]/1000.0;
```

Operations like floating point multiplication and division sometimes have significant delays (5 or more cycles). If a loop body is small, code scheduling cannot do much; there are not enough instructions to cover all the delays. In such a situation loop unrolling may help. The body of loop is replicated n times, with loop indices and loop limits suitably modified. For example, with n = 2, the above loop would become

```
for (i=2; a <999999; i+=2){
   a[i] = a[i-1]*a[i-2]/1000.0;
   a[i+1] = a[i]*a[i-1]/1000.0; }
```

A larger loop body gives a code scheduler more instructions that can be placed after instructions that may stall. How can we determine the value of n (the loop unrolling factor) necessary to cover all (or most) of the delays in a loop body? What factors limit how large n (or an unrolled loop body) should be allowed to become?

Loop unrolling has much in common with **software pipelining**, which aims to schedule several loop iterations simultaneously. A popular variant of software pipelining first schedules only one iteration. If all pipeline stalls can't be covered, a second iteration is included, etc. As an upper limit to how many iterations to unroll, we can use the maximum delay possible for any instruction being scheduled. Thus if a multiply has a delay of 5 cycles, scheduling 5 iterations will certainly allow at least the first iteration's multiply to be fully covered. Note though that iterations near the bottom of the loop may still see some delays. That is, if we unroll 5 iterations of a loop, long delay instructions in the last of the 5 loop copies are helped least. They still can be moved upward, so some benefit can be expected.

In practice a limiting factor in unrolling is the availability of registers. If we schedule i iterations in an expanded loop body, we may see i copies of a long delay instruction (like a multiply or divide) executing concurrently. Each will require a different result register. In fact adding iterations until all delays are covered or until available registers run out is a reasonable approach when unrolling a loop.

21. Assume we extend the IR tree patterns defined in Figure 13.27 with the patterns for the MIPS add and load-immediate instructions shown in Figure 13.35.

Show how the IR tree of Figure 13.36, corresponding to:

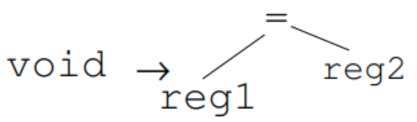
```
A[i+1] = (1+i)*1000;
```

Would be matched. What MIPS instructions would be generated?

In the first printing of Crafting a Compiler the instruction patterns listed in Figure 13.35 on page 544 are incomplete. Patterns for loading an address and storing the contents of a register into a memory location addressed by a register are needed:

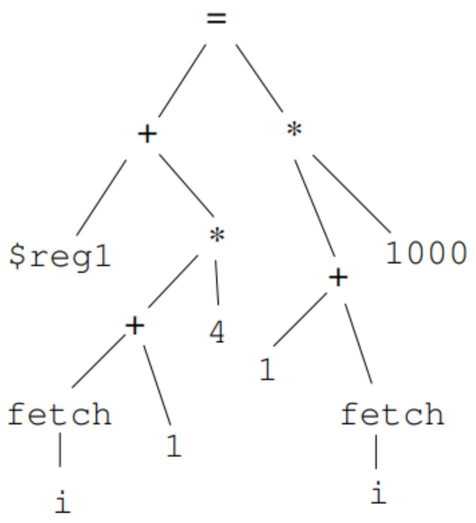
$$\text{reg} \rightarrow \text{adr}$$

```
li $reg,adr
```



```
sw $reg2,0($reg1)
```

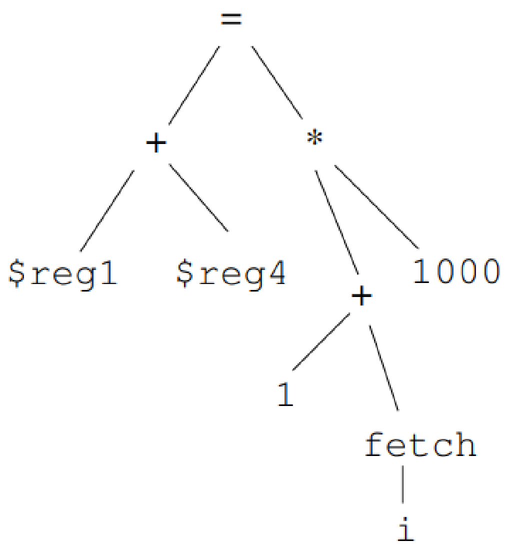
These patterns will be added to the figure in subsequent printings. The IR tree is matched bottom-up, so that code for operands is generated before the code for operators. In this example, register allocation is very simple—each time a register is assigned, a new register is allocated. A more realistic register assignment strategy is discussed in the next exercise. Our traversal will be bottom-up, left to right. So first the address of A is loaded into a register using a load immediate instruction. The IR tree is now:



Generated code:

```
li  $reg1,A
```

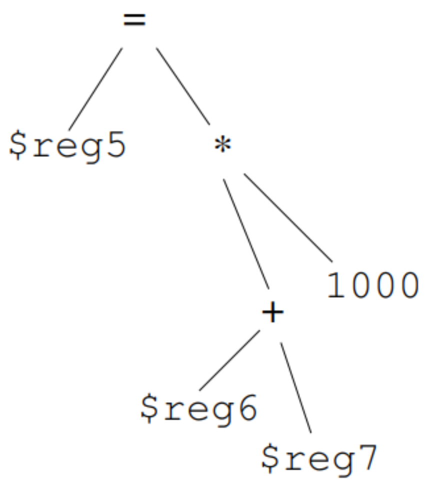
Next, the value held in location i is loaded into a register. One is added using a load immediate, and then i+1 is multiplied by 4 (using a shift left instruction) to compute the byte offset of i+1 in array A:



Generated code:

```
    li   $reg1,A
    lw   $reg2,i
    addi   %reg3,$reg2,1
    sll  $reg4,$reg3,2
```

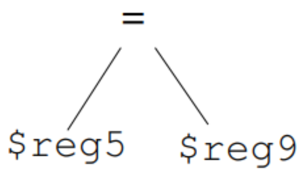
Adding $reg1 and $reg4 yields the address of A[i+1]. In translating 1+i we can't use an add immediate instruction because the pattern only allows an integer literal as the right operand. A richer set of instruction patterns might allow an integer literal as the left operand (since addition is commutative). We therefore load 1 in one register and load the value at location i into a second register:



Generated code:

```
    li   $reg1,A
    lw   $reg2,i
    addi   %reg3,$reg2,1
    sll  $reg4,$reg3,2
    add   %reg5,$reg1,$reg4
    li   $reg6,1
    lw   $reg7,i
```

We add registers $reg6 and $reg7 together to obtain 1+i, then multiply by 1000:



Generated code:

```
li   $reg1,A
lw   $reg2,i
addi  %reg3,$reg2,1
sll  $reg4,$reg3,2
add  %reg5,$reg1,$reg4
li   $reg6,1
lw   $reg7,i
add  %reg8,$reg6,$reg7
mul  %reg9,$reg8,1000
```

Finally, we store the value of (1+i)*1000, held in $reg9, into the memory location A[i+1], whose address is in $reg5:

```
void
```

Generated code:

```
li   $reg1,A
lw   $reg2,i
addi  %reg3,$reg2,1
sll  $reg4,$reg3,2
add  %reg5,$reg1,$reg4
li   $reg6,1
lw   $reg7,i
add  %reg8,$reg6,$reg7
mul  %reg9,$reg8,1000
sw   %reg9,0($reg5)
```

Group Exercises - Lecture 18

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Fischer et. al. exercises 5, 4, 15 (Note that GCR... skould be GCRegAlloc), 26, 28, 30 on pages 538-546 (exercises 4, 5, 14, 25, 29, 30 on pages 570-578 in GE)

   5. Array bounds checks are mandatory in Java and C#. They are very useful in catching errors, but are also fairly expensive, especially in loops. It is often the case that conditional branches provide information useful in optimizing or even eliminating unnecessary bounds checks. For example, in:

```
while (i < 10) {
   print(a[i++]); }
```

   We know i must be less than 10 whenever array a is indexed. Moreover, since i is never decreased in the loop, a single check that i is non-negative at loop entrance suffices.

   Suggest ways in which information provided by conditional branches (in conditionals and loops) can be exploited when code to index arrays is generated.

   This analysis can be implemented as a variant of constant propagation (Section 14.6 on page 623). Rather than track variables known to hold a constant value, we track the range of values a variable may hold. Thus in our example, at the start of each iteration, we know i must be in the range [-∞..9] (-∞ and ∞ represent the minimum and maximum possible integer values). Range information must be updated across assignments. Thus after i++ we conclude i must be in the range [(-∞+1)..10].

   At the end of conditionals and loops, range information must be merged. Thus if we know j is in the range [0..10] at the end of the then part of an if, and j is in the range [-5..5] at the end of the else part, then j is in the range [-5..10] after the if statement.

   In Java the size of an array is part of its value, but it is very often the case that an array variable is allocated only once, with an easily identifiable size. Note too that conditional constant propagation, in which propagated constant values can be used to evaluate and simplify boolean expressions, can be generalized to range analysis. Thus if we know k is in the range [0..10], then an expression like (k<0) can be simplified to false.

4. A common subprogram optimization is *inlining*. At the point of call, the body of the called method is substituted for the call, with actual parameter values used to initialize local variables that represent formal parameters.

Assume we have the bytecodes that represent the body of subprogram P that is marked private or final (and hence cannot be redefined in a subclass). Assume further that P takes n parameters and uses m local variables. Explain how we could substitute the bytecodes representing P's body for a call to P, prior to machine code generation. What changes in the body must be made to guarantee that the substituted bytecodes do not “clash” with other bytecodes in the context of call?

All references to the n parameters in P must be updated so P references the arguments given at the call site. Also, care must be taken to rename the m local variables so they do not conflict with the parameters and local variables currently in scope at the call site.

15. Assume we have n registers available to allocate to a subprogram. Explain how, using either GCRegAlloc or PriorityRegAlloc, we can estimate the total cost of register spills within the subprogram. How could this cost estimate be used in deciding how many registers to allocate to a subprogram?

The key is getting reasonable estimates of how often each basic block in a subprogram will execute. Profiling is commonly used. An important issue is how closely test data mimics actual user data. Lacking dynamic data provided by profiling, static estimates may be used. We already scale spill cost estimates by a factor of 10 per loop nesting level. With a bit more effort actual loop counts can be extracted for the common case of constant loop bounds. We reduce execution counts in conditionals, perhaps assuming each leg of an if has a 50% chance of executing.

With estimates of execution frequencies, we can compute added execution time whenever GCRegAlloc or PriorityRegAlloc elects to do a spill. We may total added instruction counts or perhaps focus on added memory reads (since cache misses and page faults can be very costly).

Given accurate spill costs estimates, we can now ask a key question — how much benefit do we obtain by giving a subprogram an extra register. Interprocedural register allocators ask this question repeatedly, deciding which of a number of subprograms can best use an additional register.

26. The following instruction sequence often appears in Java programs:

```
a[i] = ...
... = a[i];
```

That is, an element of an array is stored, then that same element is immediately reused. Suggest a peephole optimization rule, at the bytecode level, that would recognize this situation and optimize it using the dup bytecode.

In the first printing of Crafting a Compiler this exercise incorrectly suggested using a dup instruction instead of a dup x2. This error will be corrected in subsequent printings.

This optimization is akin to a machine-level optimization that removes a register load if the desired value is already a register resident. We will aim to make a copy of the value assigned to a[i] and use that value in the second assignment, avoiding a reload of a[i]. The patterns we use will depend on where a and i are stored. We'll cover the common case that both are local variables within a method. Related patterns will be needed in the cases that a or i are fields.

We must also consider the possibility that the instructions on the right hand side of the first assignment involve assignment or transfer of control. (We don't want i changed while evaluating the right-hand side.) Call an IR instruction that is not a label and does not transfer control or potentially change a local variable to be "pure." How many IR instructions should we expect in the right-hand side of the first assignment? We can write a number of related rules, each for a particular IR instruction count.

Alternatively, we can allow a pattern to match a range of instructions, with a fixed upper limit (so that pattern matching doesn't waste time trying to match unrealistically large right-hand sides). Let the pattern pure[1:n] match from 1 to n pure instructions. The pattern listed below will look for an array store (*iastore*) immediately followed by an array load (*iaload*) of the same element in the same array. If this is found, a dup x2 instruction duplicates the value being stored and places it below the three operands to the array store instruction. This makes the array load instruction unnecessary, as the desired value will be found at the top of the operand stack:

```
aload Ar                 aload Ar
iload Ind                iload Ind
pure[1:n]        ⇒       pure[1:n]
iastore                  dup_x2
aload Ar                 iastore
iload Ind
iaload
```

28. Many architectures include *load-negative* instruction that loads the negation of a value into a register. That is, the value, while being loaded, is subtracted from zero, with the difference stored into the register. Suggest two instruction-level peephole optimization patterns that can make use of a load-negative instruction.

The obvious pattern corresponds to ...=-b which will load b then negate it. A load negative of b is a clear improvement. For uniformity, we'll use the MIPS instruction set and we'll assume *lwn* is a load word negative:

```
lw $reg,loc        ⇒    lwn $reg,loc
negu $reg,$reg
```

A less obvious (and probably less common) replacement involves an expression like ...=-(a+b). This will generate two loads, an add, and a negation.

But this expression is equivalent to ...=(-a)-b which can be implemented using a load negative, a load and a subtraction:

```
lw $reg1,loc1                  lwn $reg1,loc1
lw $reg2,loc2             ⇒    lw $reg2,loc2
add $reg1,$reg1,$reg2          sub $reg1,$reg1,$reg2
negu $reg1,$reg1
```

30. Assume we have a peephole optimizer that has n replacement patterns. The most obvious approach to implementing such an optimizer is to try each pattern in turn, leading to an optimizer whose speed is proportional to n.

Suggest an alternative implementation, based on hashing, that is largely independent of n. That is, the number of patterns considered may be doubled without automatically doubling the optimizer's execution time.

The basic idea is to represent patterns in a way that allows us to hash them. We can then construct a map that will map those pattern hashes to an optimization.

For example: optimizationMap[“constant + constant”]=constantFoldingOptmization. We then traverse the tree and plug the node combinations we find into the map. If a node combination maps to an optimization, we apply it.

Individual Exercises - Lecture 19

1. Read the article under additional references.
   Can be found here (Is available for free through AAU AUB here https://www.aub.aau.dk, remember to login first!): https://link.springer.com/article/10.1007%2FBF00289243

2. Sebesta, programming exercises 5, page 511, do the exercise in C, and Java or C#.
   5. Write an abstract data type for complex numbers, including operations for addition, subtraction, multiplication, division, extraction of each of the parts of a complex number, and construction of a complex number from two floating-point constants, variables, or expressions. Use C++, Java, C#, or Ruby.
   Example implementation using C. As C does not provide classes or objects a struct is used to represent the data while operations on the data type are implemented as functions. While objects could be simulated using function pointers taking the struct itself as the first argument, this would make the program much more complicated and, as such, much harder to read and maintain.

```c
typedef struct Complex {
  const double real;
  const double imaginary;
} Complex;
Complex new(double real, double imaginary) {
  return (Complex) { real, imaginary };
}

Complex add(Complex a, Complex b) {
  return new(a.real + b.real, a.imaginary + b.imaginary);
}

Complex sub(Complex a, Complex b) {
  return new(a.real - b.real, a.imaginary - b.imaginary);
}

Complex mul(Complex a, Complex b) {
  double real = a.real * b.real - a.imaginary * b.imaginary;
  double imaginary = a.real * b.imaginary + a.imaginary * b.real;
  return new(real, imaginary);
}

Complex div(Complex a, Complex b) {
  double x = a.real * b.real + a.imaginary * b.imaginary;
  double y = a.imaginary * b.real - a.real * b.imaginary;
  double z = b.real * b.real + b.imaginary * b.imaginary;
  return new(x / z, y / z);
}

double real(Complex a) {
  return a.real;
}

double imaginary(Complex a) {
  return a.imaginary;
}
```

Example implementation using Java. Contrary to C, Java provides support for object-oriented programming through classes and objects. Using a class both the code and data can combine into a single entity. In addition, the data can be hidden from the user of the class with the `private` access modifier while the data is only made readonly in the C version of the program. As Java does not support operator overloading (unlike C#) equations with `Complex` must be written using method calls instead of mathematical operators.

```
class Complex {
    public Complex(double real, double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    public Complex add(Complex other) {
        return new Complex(this.real + other.real,
                           this.imaginary + other.imaginary);
    }

    public Complex sub(Complex other) {
        return new Complex(this.real - other.real,
                           this.imaginary - other.imaginary);
    }

    public Complex mul(Complex other) {
        double real = this.real * other.real -
            this.imaginary * other.imaginary;
        double imaginary = this.real * other.imaginary +
            this.imaginary * other.real;
        return new Complex(real, imaginary);
    }

```

```java
    public Complex div(Complex other) {
        double x = this.real * other.real +
            this.imaginary * other.imaginary;
        double y = this.imaginary * other.real -
            this.real * other.imaginary;
        double z = other.real * other.real +
            other.imaginary * other.imaginary;
        return new Complex(x / z, y / z);
    }

    @Override
    public String toString() {
        return this.real + " " +  this.imaginary + "i";
    }

    public double getReal() {
        return this.real;
    }

    public double getImaginary() {
        return this.imaginary;
    }

    private final double real;
    private final double imaginary;
}
```

3. Sebesta, exercise 14 and 16, page 565.
   14. Explain diamond inheritance.
   Diamond inheritance occurs when a class (D) is a subtype of two different classes (B and C) through multiple inheritance which themself are a subtype of a class (A). A significant problem with multiple inheritance is that both B and C can define a method with the same name and the same protocol. For example, a class that implements an interface must define all of the methods declared in the interface. So, if both the parent class and the interface include methods with the same name and protocol, the subclass must reimplement that method to avoid the name conflict.

   See Also: https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

16. What is the primary reason why all Java objects have a common ancestor?

One reason why all Java objects have a common ancestor is so they can all inherit a few universally useful methods, e.g. equals(Object other), toString(), hasCode(). Another reason is that code can be written that works for all objects without using generics: https://en.wikipedia.org/wiki/Generics_in_Java.

Group Exercises - Lecture 19

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. What are the language design requirements for a language that supports abstract data types?
   Sebesta lists three design issues for abstract data types:
   (i) The first design issue is whether abstract data types can be parameterized.
   (ii) The second design issue is what access controls are provided and how such controls are specified.
   (iii) Finally, the language designer must decide whether the specification of the type is physically separate from its implementation. For more information see Sebesta Section 11.3.

3. Sebesta, review questions, 21, 25, 33, 35, 36 page 509.
   21. Where are Java classes allocated?
   Instantiations are allocated from the heap
   25. Describe the fundamental differences between C# structs and its classes.
   Structs are value types, classes are references.
   Structs cannot be default constructed, inherit another struct, or implement an interface.
   33. What problems can occur using C to define abstract data types?
   C does not have complete support for abstract data types, but both abstract data types and multiple-type encapsulations can be simulated by wrapping them in a library file, i.e. a header file and then creating a .c file with the implementation of said functions.
   A problem with this solution could for example be, a user could simply cut and paste the definitions from the header file into the client program, rather than using #include. This would work, because #include simply copies the contents of its operand file into the file in which the #include appears. However, this introduces two problems. First, the documentation of the dependence of the client program on the library (and its header file) is lost. Second, suppose a user copies the header file into his or her program. Then the author of the library changes both the header and the implementation files. Following this, the user uses the new implementation file with the old header. For example, a variable x could have been defined to be **int** type in the old header file, which the client code still uses, although the implementation code has been recompiled with the new header file, which defines x to be **float**. So, the implementation code was compiled with x as an **int** but the client code was compiled with x as a **float**. The linker does not detect this error. Read more in section 11.6.2 in Sebestas book.

35. What is a Java package, and what is its purpose?
Packages provide naming encapsulation. Access modifiers (private, protected etc.) also allows restriction of access across packages.
36. Describe a .NET assembly.
An assembly is a file (.dll or .exe) that defines a module that can be separately developed. (See Sebesta Section 11.6.4)

4. Sebesta, problem set, exercise 2, page 510.
   2. Suppose someone designed a stack abstract data type in which the function top returned an access path (or pointer) rather than returning a copy of the top element. This is not a true data abstraction. Why? Give an example that illustrates the problem.
   The problems created by making an abstract data type a pointer are:
   (a) There are many inherent problems in dealing with pointers (see Chapter 6),
   (b) comparisons between objects do not work as expected, because they are only pointer comparisons (rather than pointed-to value comparisons),
   and (c) the implementation of the abstract type cannot control allocation and deallocation of objects of the type (the user can create a pointer to an object with a variable declaration and use it without creating an object).

5. Sebesta, review questions, 1 - 58, page 562-564
   1. Describe the three characteristic features of object-oriented languages.
   Abstract data types
   Inheritance
   Dynamic binding
   2. What is the difference between a class variable and an instance variable?
   A class variable (static in C#) belongs to the class and is accessible from all instances of the class. This is opposed to the instance variable that belongs to the individual instance (object) of the class and is accessible only through that instance.
   3. What is multiple inheritance?
   The concept of a class inheriting from more than one class.
   4. What is a polymorphic variable?
   A variable that can hold values of different types.
   5. What is an overriding method?
   In terms of inheritance, it is sometimes useful to replace a method inherited from a superclass, that is where overriding comes in.
   For example, in C# a method marked override explicitly replaces a method inherited from a superclass with the new method of the same signature. In C#, however, this is only possible if the method from the super class is marked as virtual.
   6. Describe a situation where dynamic binding is a great advantage over static binding.
   If you need to administrate objects of different classes, that all inherit from a common super class (base class), and call an inherited method on all of them.

7. What is a virtual method?
A dynamically bound method that can be overriding in a derived class. The keyword virtual is for example used in C# and C++.
8. What is an abstract method? What is an abstract class?
An abstract method has no implementation and must be defined in inherited derived classes.
An abstract class cannot be instantiated, and is instead meant as a base blueprint for derived classes to implement.
9. Describe briefly the seven design issues used in this chapter for object-oriented languages.

1. The exclusivity of objects: Should any other types be allowed?
2. Are subclasses subtypes?: Should two objects extending the same superclass be interchangeable?
3. Single and multiple inheritance: Can an object inherit from than one class?
4. Allocation and deallocation of objects: When and how? (Heap or stack?
5. Dynamic or static binding: Which to use?
6. Nested classes: Should we allow for classes only visible to specific classes?
7. Initialization of objects: How are objects initialized to values?

10. What is a nesting class?
A class defined within another class.
Useful if a class is only used in one place
11. What is the message protocol of an object?
The message protocol or message interface is all the methods defined on an object. A method call is equivalent to sending a message to the object.
12. From where are Smalltalk objects allocated?
All Smalltalk objects are allocated from the heap.
13. Explain how Smalltalk messages are bound to methods. When does this take place?
Smalltalk messages are dynamically bound to methods at runtime by traversing the inheritance hierarchy of the object receiving the message and executing the correct method when found. If the top of the hierarchy (Object) has been reached and no method matching the message is found, an error occurs. For more information see Sebesta Section 12.4.1.3 Dynamic Binding.
14. What type checking is done in Smalltalk? When does it take place?
Smalltalk only supports dynamic type checking. Type checking is performed at runtime when a message is sent to an object. For more information see Sebesta Section 12.4.1.3 Dynamic Binding.
15. What kind of inheritance, single or multiple, does Smalltalk support?
Smalltalk only supports single inheritance. All objects must derive from an existing object. For more information see Sebesta Section 12.4.1.2 Inheritance.
16. What are the two most important effects that Smalltalk has had on computing?
(i) The Smalltalk user-interface demonstrated how to integrate use of windows, mouse-pointing devices, and pop-up and pull-down menus.
(ii) The advancement of object-oriented programming. For more information see

Sebesta Section 12.4.1.4 Evaluation of Smalltalk.
17. In essence, all Smalltalk variables are of a single type. What is that type?
Everything is an object as stated in Sebesta Section 12.4.1.1 General Characteristics
18. From where can C++ objects be allocated?
Objects can be allocated statically, stack dynamic, and heap dynamic.
19. How are C++ heap-allocated objects deallocated?
Explicitly, with delete (or with smart pointers, such as shared_ptr, unique_ptr)
20. Are all C++ subclasses subtypes? If so, explain. If not, why not?
In the case of public derivation, yes.
In the case of private derivation, no.
21. Under what circumstances is a C++ method call statically bound to a method?
If it is not a pointer to the base class, or in other terms, if the class is determinable at compile time, not runtime. In general, method calls that are not defined as virtual are statically bound. See Sebesta Section: 12.4.2.3 Dynamic Binding.
22. What drawback is there to allowing designers to specify which methods can be statically bound?
Dynamic binding is slower than statically bound methods, so allowing users to define statically bound methods can increase the program's performance if a method does not need to be dynamically bound.
23. What are the differences between private and public derivations in C++?
In a private derivation public and protected members of the derived class are replaced by private copies.
24. What is a friend function in C++?
A function that has been granted access to private or protected fields of a class.
25. What is a pure virtual function in C++?
Roughly equivalent to an abstract function in C#
Has no implementation, written like this: virtual void foo() = 0;
Requires implementation in the deriving classes.
26. How are parameters sent to a superclass's constructor in C++?
Parameters are sent to the superclass constructor as part of the subclass constructor definition. Defining the inheritance requires the superclass's constructor parameters to be provided as well.
27. What is the single most important practical difference between Smalltalk and C++?
C++ uses static type checking which allows the compiler to detect errors at compile time which cannot be done in Smalltalk. C++ is generally also more efficient than Smalltalk
28. If an Objective-C method returns nothing, what return type is indicated in its header?
Void
29. Does Objective-C support multiple inheritance?
No
30. Can an Objective-C class not specify a parent class in its header?
Yes it can. As part of its interface declaration.

31. What is the root class in Objective-C?
The NSObject
32. In Objective-C, how can a method indicate that it cannot be overridden in descendant classes?
There is no way to prevent a subclass from overwriting an inherited method.
33. What is the purpose of an Objective-C category?
A category is a collection of methods that can be added to a class. No new instance variables can be added to this secondary interface. It can be used in some places to support multi inheritance without naming collisions on classes.
34. What is the purpose of an Objective-C protocol?
Similarly to an interface in Java/C#, it can provide some expected functionality of a class that implements the protocol.
35. What is the primary use of the id type in Objective-C?
It is a generic type that can refer to any object. Anything can be cast to an id type and an id type can be cast to anything.
36. How is the type system of Java different from that of C++?
Only primitive scalar types are not objects (bool, char, int, float etc.). Everything else is an object in Java. This is not the case for C++.
37. From where can Java objects be allocated?
All Java objects are dynamically allocated from the heap.
38. What is boxing?
Boxing is used in Java for having an object type of the primitive types (int, float, char etc.). The boxed versions har objects with names like Integer and Double. This was done, since the Java generics can only be used with objects. The compiler, however, autoboxes / unboxes.
The following code will box the i into an object of the type Integer and add it to the list.

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(i);
```

Due to boxing and type erasure the compiler changes the code above to approximately:

```
List<Object> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(Integer.valueOf(i));
```

39. How are Java objects deallocated?
In Java deallocation is done in the background by the Java garbage collector.
40. Are all Java subclasses subtypes?
Yes. The only class (which is not a subclass) that is not a subtype is the `Object` class.
41. How are superclass constructors called in Java?
`super(args)` from inside the subclass constructor.

42. Under what circumstances is a Java method call statically bound to a method?
Static binding is used if it is defined as `final`, `static`, or `private`.
43. In what way do overriding methods in C# syntactically differ from their counterparts in C++?
C# has a different abstract implementation.
In C# it is possible to mark a method `sealed` if they should be marked as explicitly not overridable.
44. How can the parent version of an inherited method that is overridden in a subclass be called in that subclass in C#?
Using the `base` keyword.
`base.[MethodName]([Parameters]);`
45. How does Ruby implement primitive types, such as those for integer and floating-point data?
Every value is an object. (unlike for example Java where an `int` is a primitive type but an `Integer` is an object)
46. How are getter methods defined in a Ruby class?
Multiple getter methods can be defined in one line using the following syntax:
`Attr_reader :variable_1, :variable_2`
Where `Attr_reader` is a built-in method that takes the variable names as arguments.
47. What access controls does Ruby support for instance variables?
Variables are effectively private, and constants public.
48. What access controls does Ruby support for methods?
Public, protected, and private.
49. Are all Ruby subclasses subtypes?
No.
50. Does Ruby support multiple inheritance?
No.
51. What does reflection allow a program to do?
To access its metadata directly, to use it, or to intercede in execution. In java you can for example read the name of a class and its fields into strings.
52. In the context of reflection, what is metadata?
The types, names and structures of a program.
53. What is introspection?
The process of a program examining its own metadata.
54. What is intercession?
The process of interceding in the execution of a program.
55. What class in Java stores information about classes in a program?
java.lang.Class
56. For what is the Java name extension .class used?
As a way to obtain the Class object of that class.
57. What does the Java getMethods method do?
Gets a method by name.
58. For what is the C# namespace System.Reflection.Emit used?
Tools for the emission of MSIL and other metadata.

6. Sebesta, problem set, 2, 6, 19 page 564-565.

   2. Explain the principle of abstraction.

   Abstraction is the process of creating a representation of an object or system that omits unnecessary details or attributes.

   6. Compare the multiple inheritance of C++ with that provided by interfaces in Java.

   Earlier versions of Java did not allow for interfaces to provide an implementation for a method, only a definition, so the problems caused by multiple inheritance in C++ could not occur. However, with the addition of default methods to Java interfaces, two different interfaces can provide different implementations of a method with the same signature. If this occurs the programmer must manually resolve this conflict by implementing the method on the class. More information about default methods: https://dzone.com/articles/interface-default-methods-java

   19. What are the differences between a C++ abstract class and a Java interface?

   Both instance variables, method definitions, and method implementations can be part of a C++ abstract class, however, a Java interface can only contain method definitions and method implementations but not instance variables.

7. Discuss how Tennent's principle may be used in your own language design

   Tennent's correspondence principle is a powerful programming language design guideline. It says that the meaning of an expression or statement should not change when an extra level of block structure is added around it.

   You can read more about Tennet's Principle here: https://fanf.livejournal.com/118421.html

   If your language contains block structures or other wrapping statements, e.g. closures, consider if this affects the meaning of an expression or statement.

Example Solutions - Lecture 20

1. The GNU Compiler Collection (gcc) compiler suite offers many levels of optimizations. Investigate the various levels that are available for a given language (e.g. C++). Hint: look at gcc flags controlling optimizations (Fischer et. al. exercise 1 on page 668 in GE)
   Resources about the optimizations available in GCC and LLVM:
   a. GCC 2.95.3: http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_2.html#SEC10
   b. GCC 9.3.0: https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/Optimize-Options.html
   c. Clang 11: https://clang.llvm.org/docs/UsersManual.html
   d. LLVM's Passes: https://llvm.org/docs/Passes.html

   How different combinations of compilers, flags, and platforms change the instructions produced for a specific program is easy to investigate using the Compiler Explorer at: https://godbolt.org/

Group Exercises - Lecture 20

1. Discuss the outcome of the individual exercises
   Did you all agree on the answer?

2. The GCC compiler suite can compile Java programs from either source or bytecode form. What are the advantages and disadvantages of compiling from source as compared to compiling from bytecode form? (Fischer et. al. 3, on page 668 in GE)
   While source code is more difficult to parse than bytecode, it often contains more information about the program that the compiler can use for optimizations. For example, at the bytecode level different constructs in a programming language are often represented by similar sequences of bytecodes, making it harder to distinguish between them if an optimization only applies to one of the constructs. In addition, some concepts might not even exist at the bytecode level as the source-to-bytecode compiler removes them, e.g., through the use of type erasure.