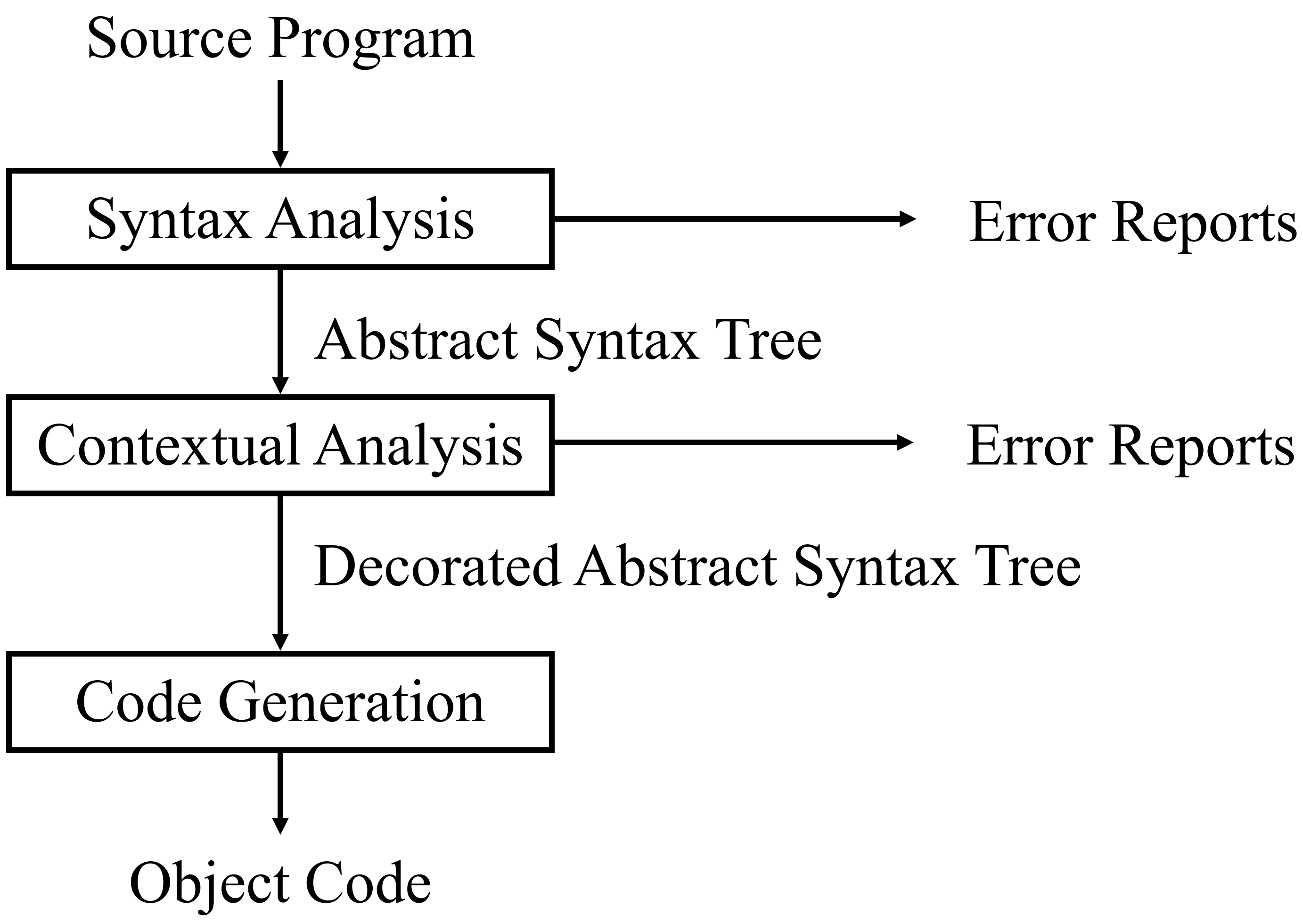
Languages and Compilers (SProg og Oversættere)

Structure of the compiler

Structure of the compiler

a) Describe the phases of the compiler
b) Give an overall description of the purpose of each phase and how the phases interface
c) Explain b) in more detail using the ac language
d) Single pass vs. multi pass compiler
   i. Issues in language design
   ii. Issues in code generation

The "Phases" of a Compiler



Different Phases of a Compiler

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

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

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

An Informal Definition of the ac Language

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

Syntax Specification

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

Figure 2.1: Context-free grammar for ac.

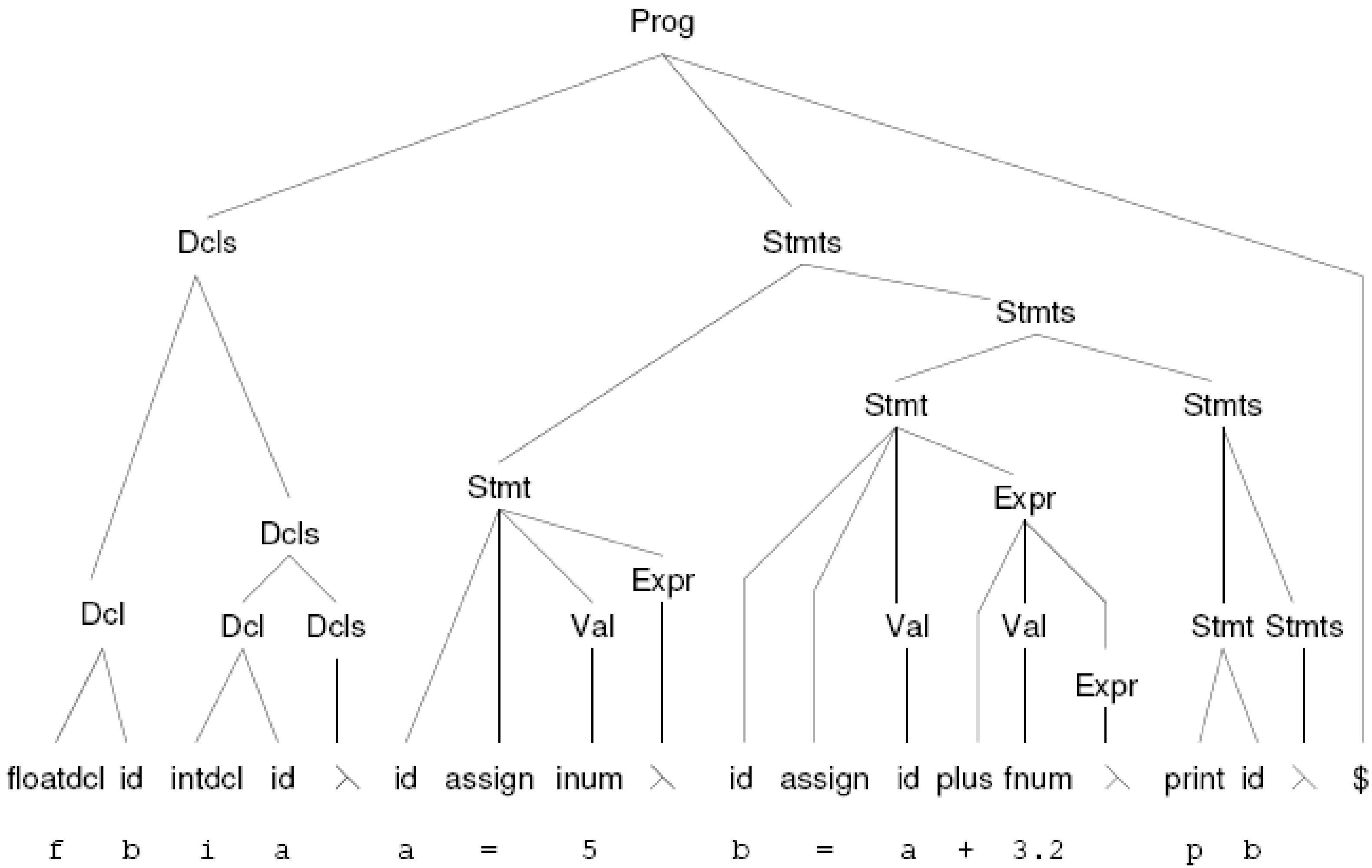


Figure 2.4: An ac program and its parse tree.

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

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

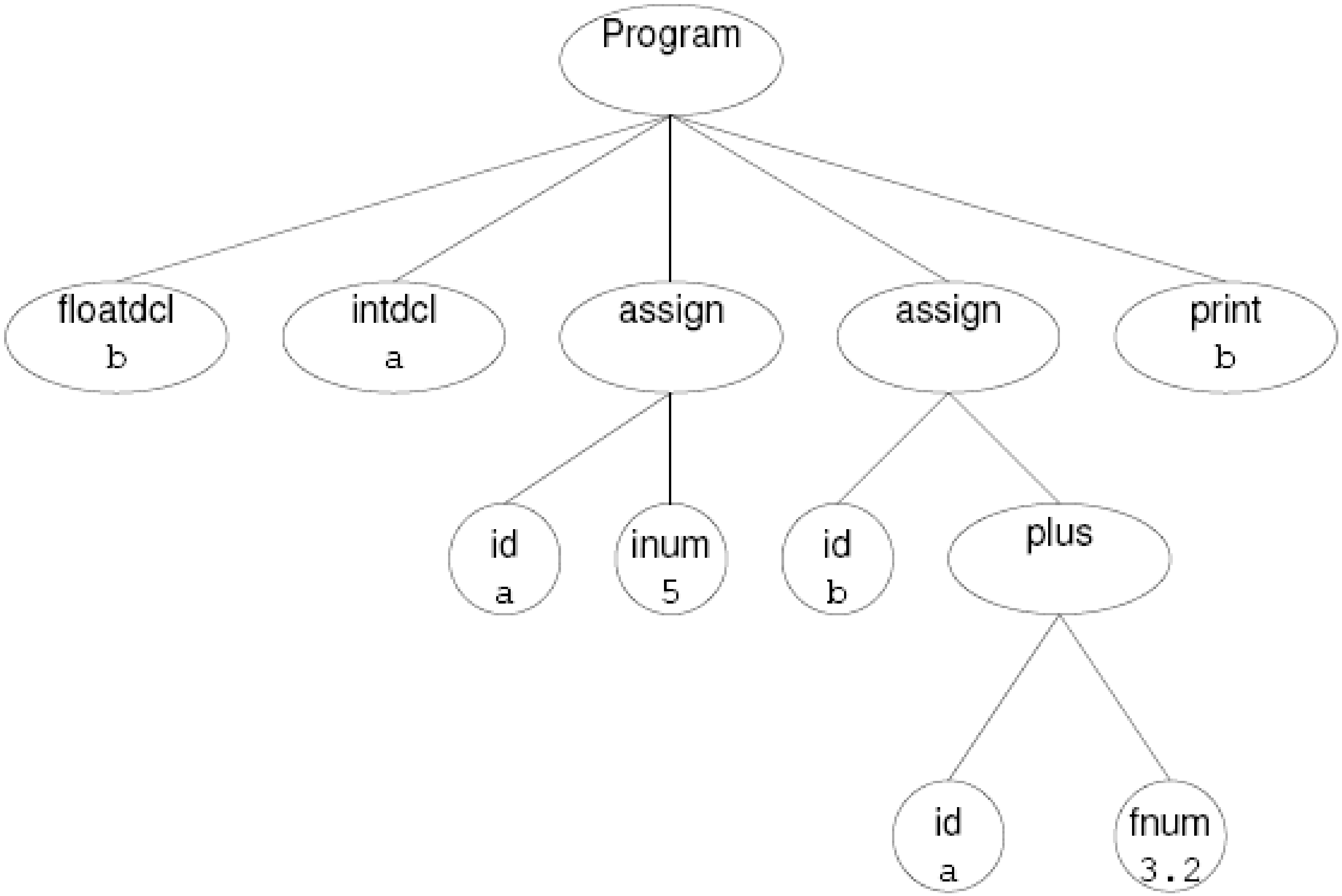
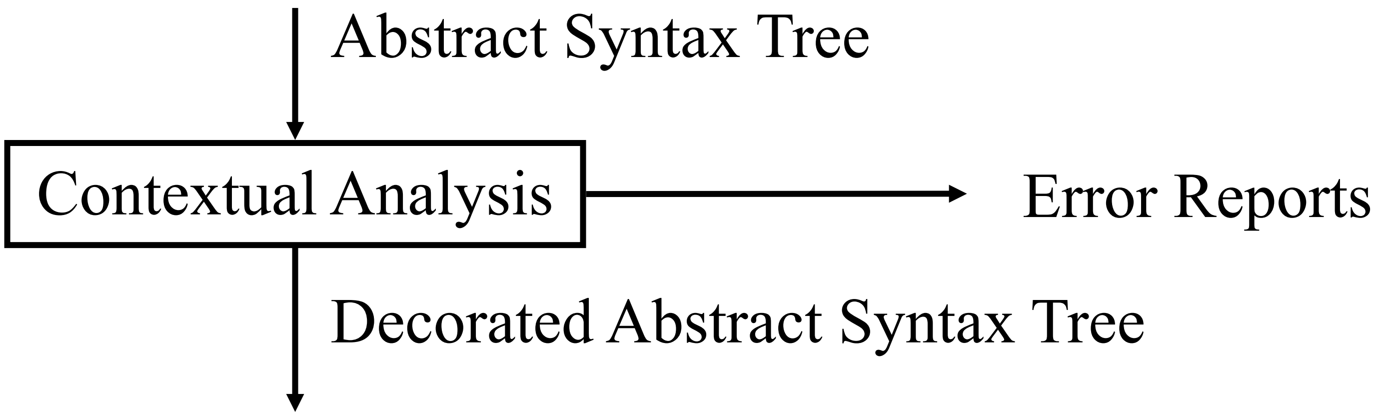


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

2) Contextual Analysis -> Decorated AST



Contextual analysis:

- Scope checking: verify that all applied occurrences of identifiers are declared
- Type checking: verify that all operations in the program are used according to their type rules.

Annotate AST:

- Applied identifier occurrences => declaration
- Expressions => Type

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

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

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

Figure 2.10: Symbol table construction for ac.

Type Checking

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

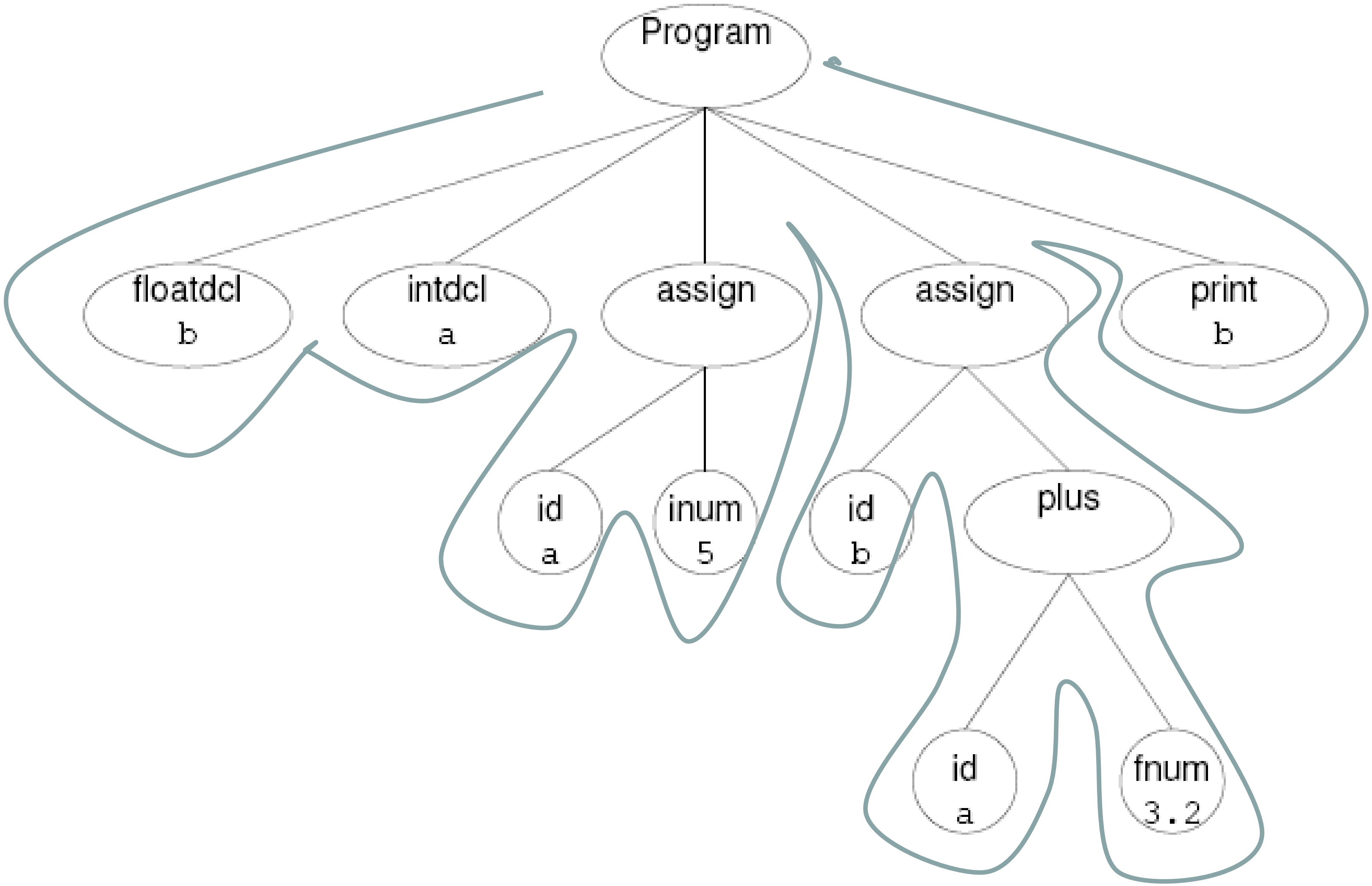


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

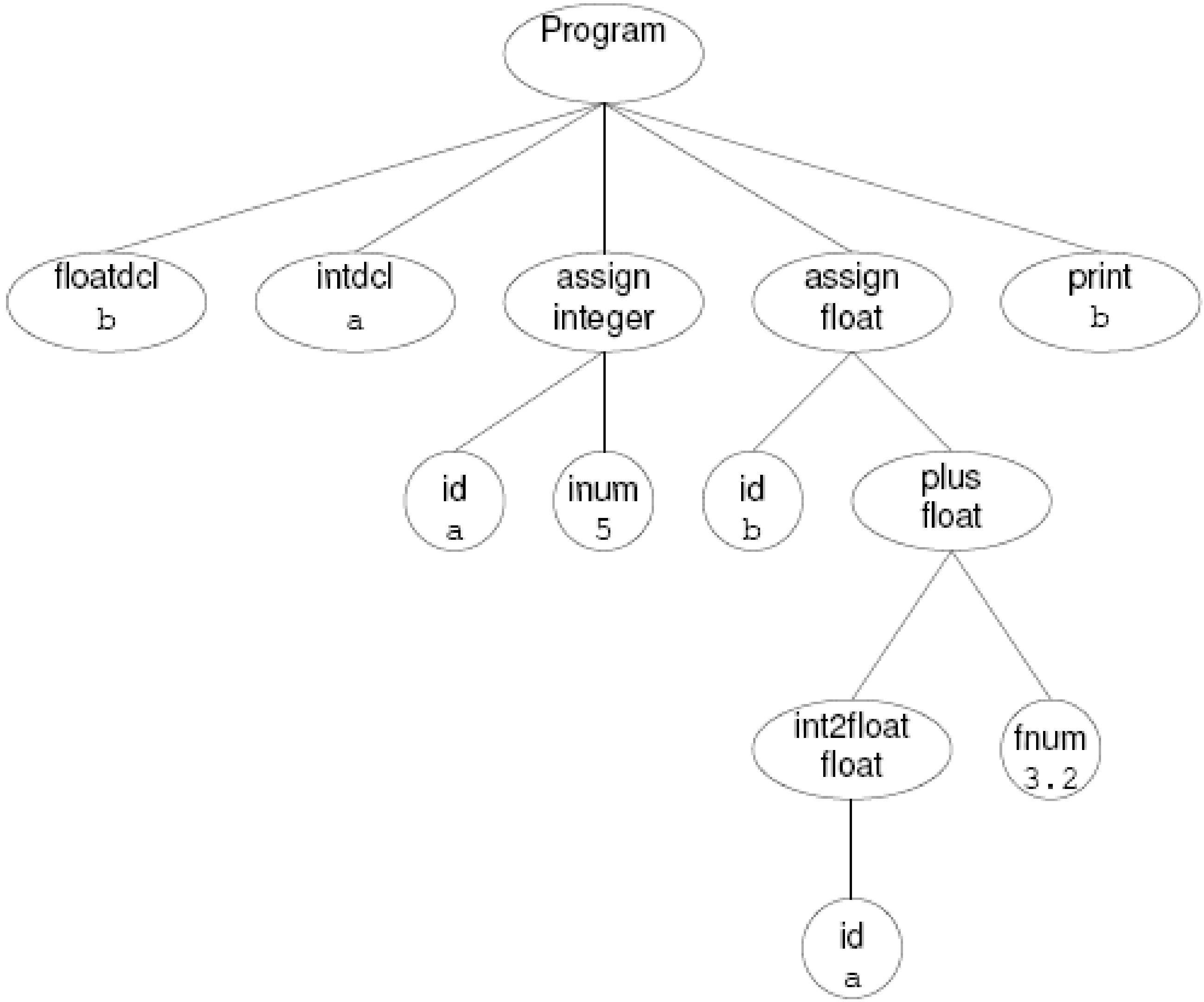


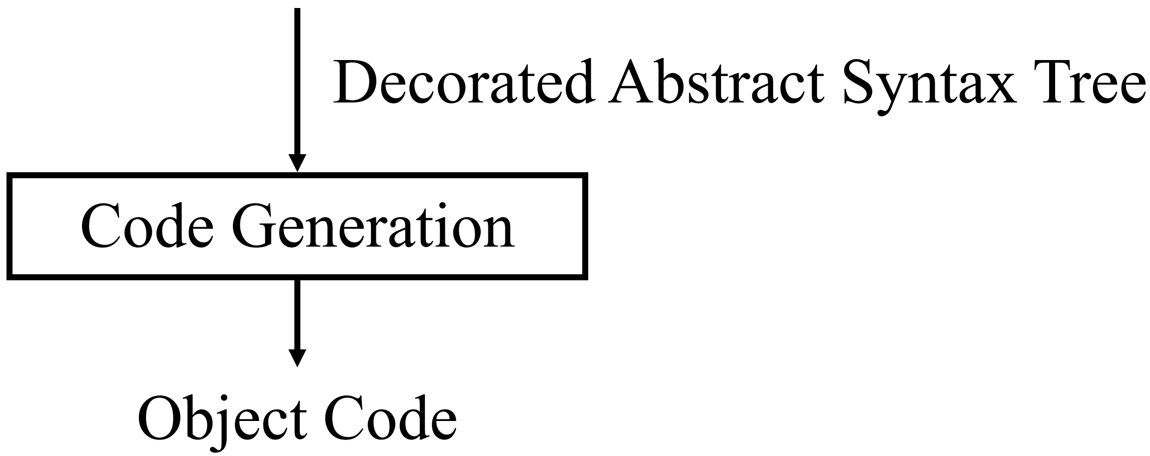
Figure 2.13: AST after semantic analysis.

Type Checking

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

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

3) Code Generation



- Assumes that program has been thoroughly checked and is well formed (scope & type rules)
- Takes into account semantics of the source language as well as the target language.
- Transforms source program into target code.

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

Figure 2.14: Code generation for ac

An Example ac Program

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

Organization of a Compiler

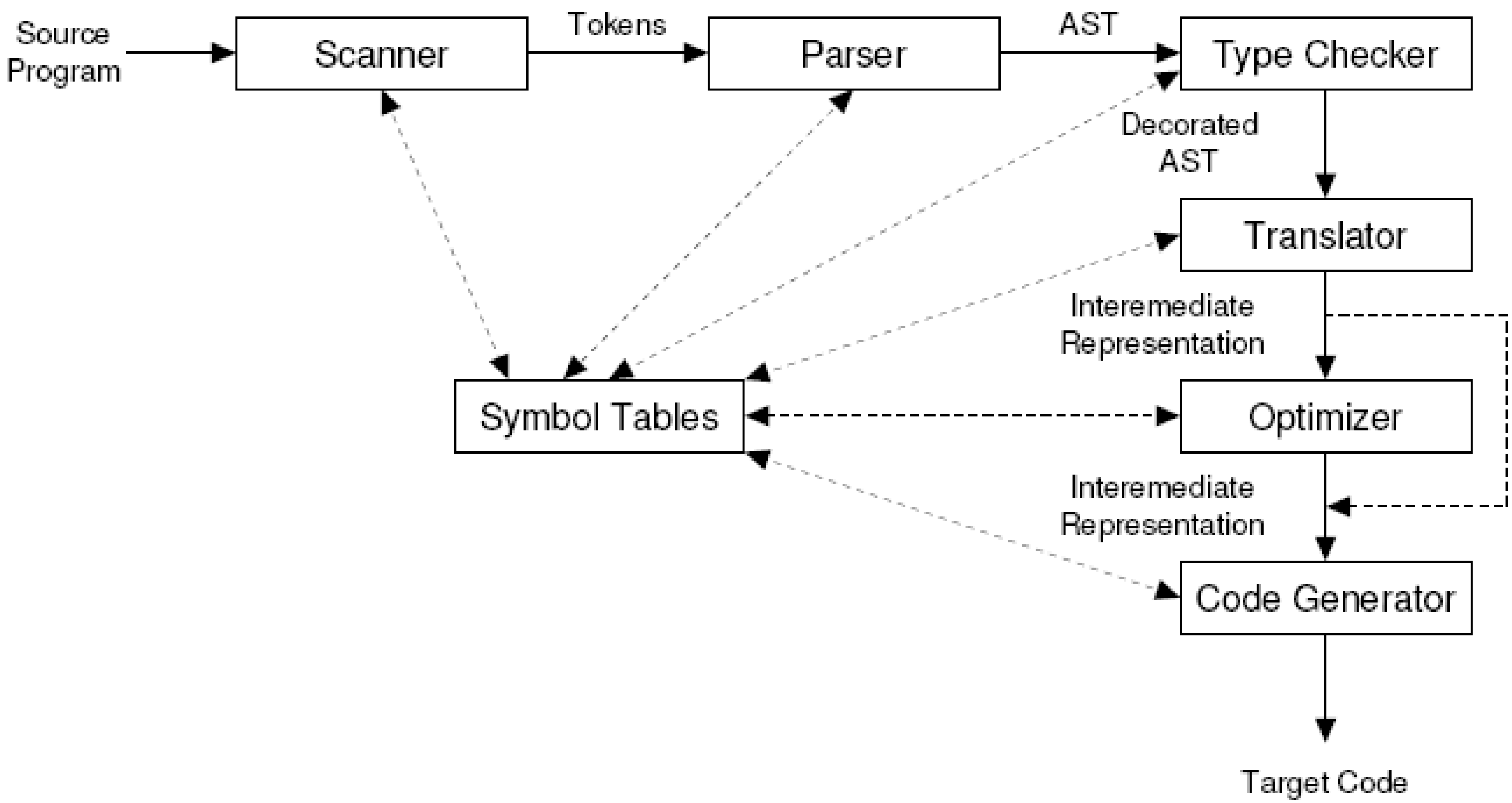


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

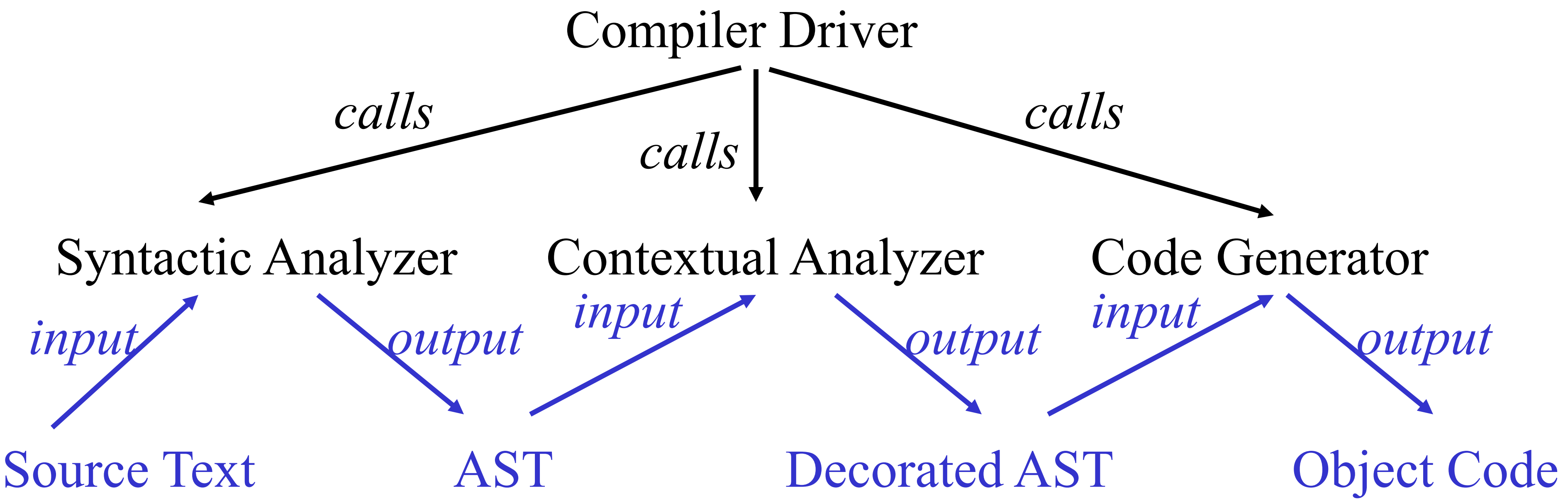
Implementing Tree Traversal

- "Traditional" OO approach
- Visitor approach
  - GOF
  - Using static overloading
  - Reflective
  - (dynamic)
  - (SableCC style)
- "Functional" approach
- Active patterns in Scala (or F#)
- (Aspect oriented approach)

Multi Pass Compiler

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

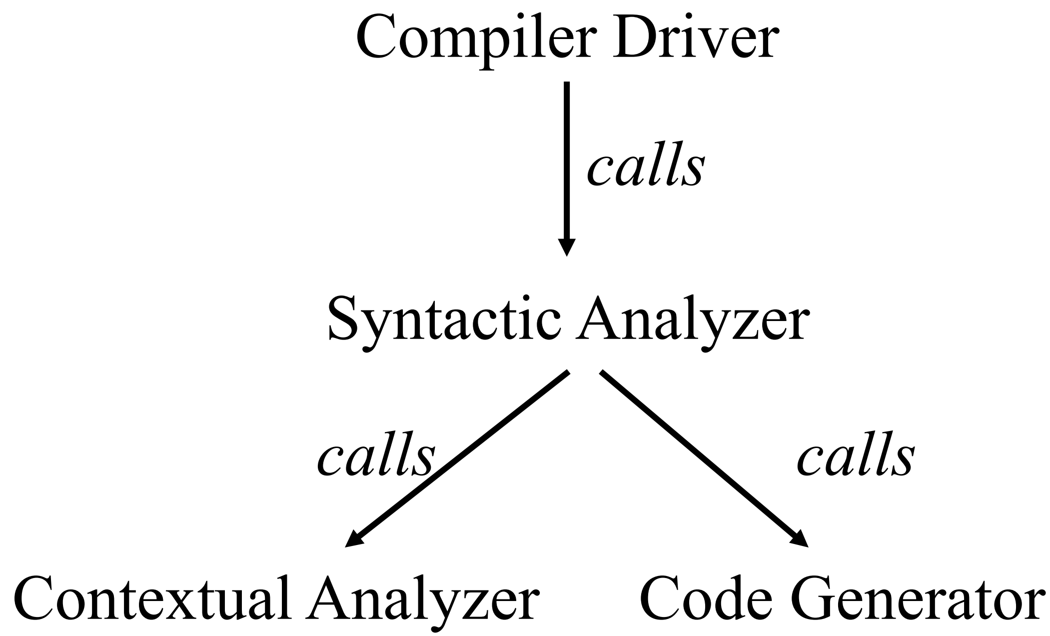
Dependency diagram of a typical Multi Pass Compiler:



Single Pass Compiler

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

Dependency diagram of a typical Single Pass Compiler:



Compiler Design Issues

	Single Pass	Multi Pass
Speed	**better**	**worse**
Memory	**better for large programs**	**(potentially) better for small programs**
Modularity	**worse**	**better**
Flexibility	**worse**	**better**
“Global” optimization	**impossible**	**possible**
Source Language	**single pass compilers are not possible for many programming languages**	

Language Issues

Example **Pascal:**

Pascal was explicitly designed to be easy to implement with a single pass compiler:

- Every **identifier** must be **declared before** it is first **use.**

```
var n:integer;

procedure inc;
begin
   n:=n+1
end
```

?

```
procedure inc;
begin
   n:=n+1
end;

var n:integer;
```

Undeclared Variable!

Language Issues

Example **Pascal:**

- Every **identifier** must be **declared before** it is **used.**
- How to handle mutual recursion then?

```
procedure ping(x:integer)
begin
   ... pong(x-1); ...
end;

procedure pong(x:integer)
begin
   ... ping(x); ...
end;
```

Language Issues

Example **Pascal:**

- Every **identifier** must be **declared before** it is **used.**
- How to handle mutual recursion then?

```
forward procedure pong(x:integer)

procedure ping(x:integer)
begin
   ... pong(x-1); ...
end;

procedure pong(x:integer)
begin
   ... ping(x); ...
end;
```

OK!

Language Issues

Example **Java:**

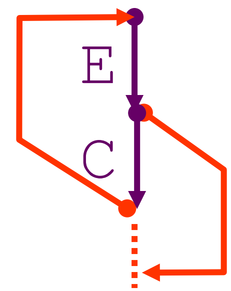
- **identifiers** can be **declared before** they are **used.**
- thus a Java compiler need at least two passes

```
Class Example {
      void inc() { n = n + 1; }
      int n;
      void use() { n = 0 ; inc(); }
}
```

Code Templates

While Command:

```
visit [while E do C] =
    l: visit [E]
        JUMPIFFALSE d
       visit[C]
       JUMP l
    d:
```




```
/*  Visitor code for Marker (13)
procedure VISIT( WhileTesting n )
    doneLabel ← GENLABEL( )
    loopLabel ← GENLABEL( )
    call EMITLABELDEF( loopLabel )
    n.GETPREDICATE( ).ACCEPT( this )
    predicateResult ← n.GETPREDICATE( ).GETRESULTLOCAL( )
    call EMITBRANCHIFFALSE( predicateResult, doneLabel )
    n.GETLOOPBODY( ).ACCEPT( this )
    call EMITBRANCH( loopLabel )
    call EMITLABELDEF( doneLabel )
end
```

Alternative While Command code template:

```
visit [while E do C] =
        JUMP h
    l: visit [C]
    h: visit[E]
        JUMPIFTRUE l
```