Languages and Compilers
(SProg og Oversættere)

Lecture 3
The ac language and compiler

Bent Thomsen

Department of Computer Science

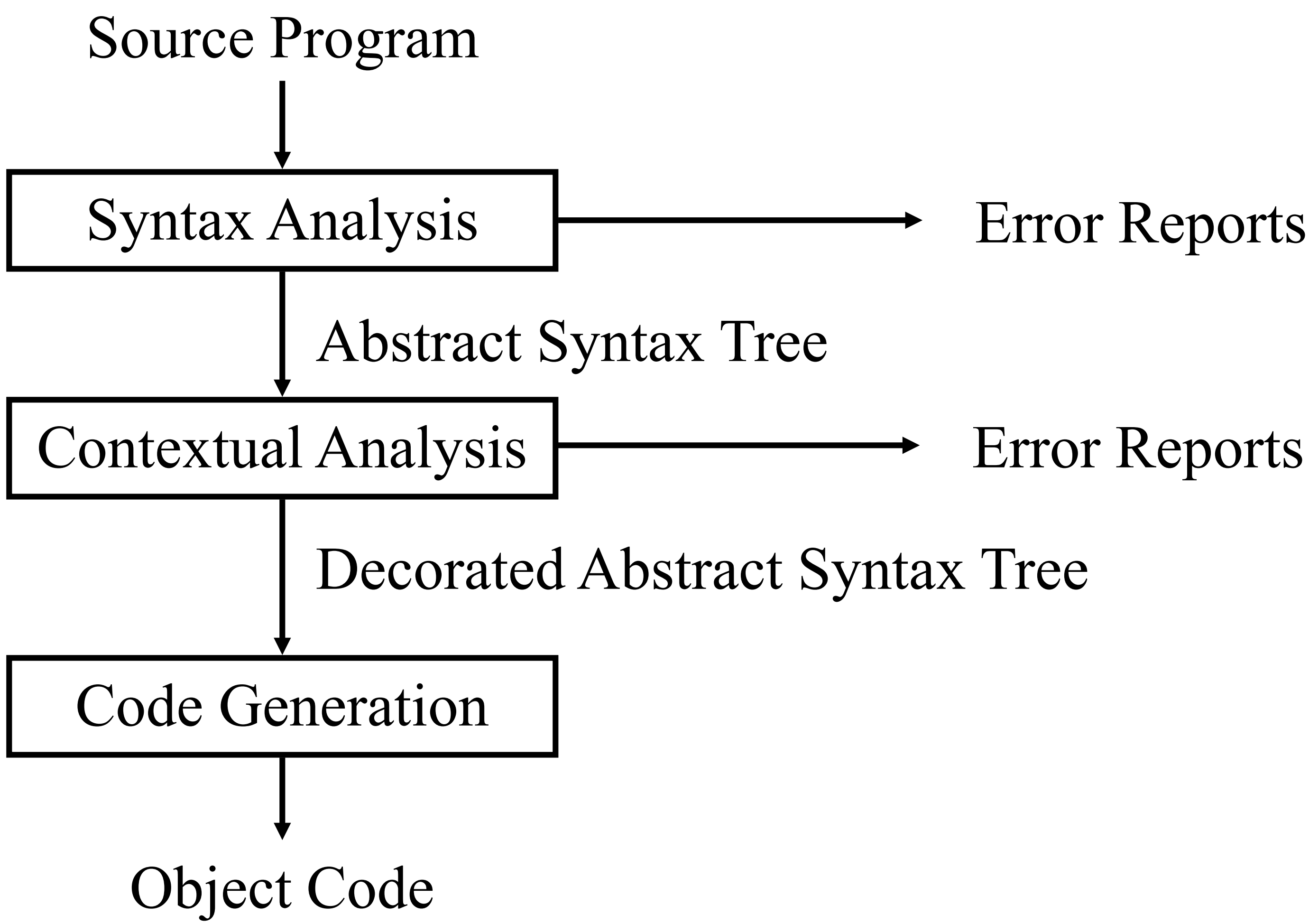
Aalborg University

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

Learning goals

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

The “Phases” of a Compiler



Different Phases of a Compiler

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

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

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

Organization of a Compiler

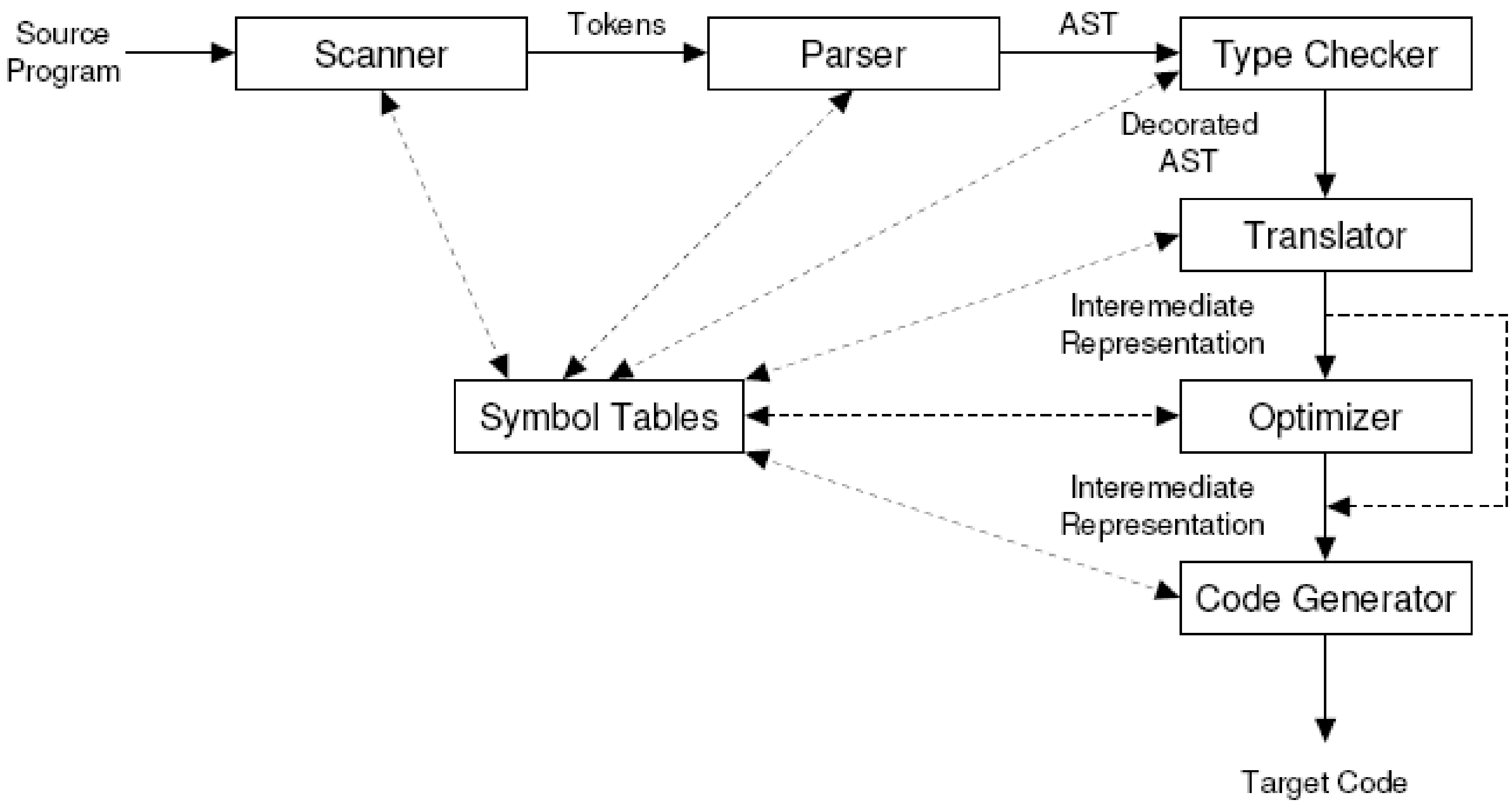


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

Phases of a Simple Compiler

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

An Informal Definition of the ac Language

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

Example Program

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

An Example ac Program

- Example ac program:
  - f b
    i a
    a = 5
    b = a + 3.2
    p b

- Corresponding dc code
  - 5
    sa
    la
    3.2
    +
    sb
    lb
    p

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

Formal Definition of ac

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

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

A sketch SOS for ac

$$
\begin{aligned}
P &\rightarrow Dcl\ Stm \\
Dcl &\rightarrow floatdcl\ id\ Dcl \\
&\mid int\ id\ Dcl \\
&\mid \varepsilon \\
Stm &\rightarrow id\ assign\ Exp \\
&\mid print\ id \\
&\mid stm\ stm \\
&\mid skip \\
Exp &\rightarrow Exp_1 + Exp_2 \\
&\mid Exp_0 - Exp_2 \\
&\mid id \\
&\mid inum \\
&\mid fnum
\end{aligned}
$$

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

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

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

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

$E \vdash \varepsilon : ok$

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

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

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

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

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

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

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

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

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.

Context Free Grammar

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

Example Program

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

Step	Sentential Form	Production Number
1	⟨Prog⟩	
2	$\boxed{\text{⟨Dcls⟩ Stmts \$}}$	1
3	$\boxed{\text{⟨Dcl⟩ Dcls}}$ Stmts $	2
4	$\boxed{\text{floatdcl id}}$ ⟨Dcls⟩ Stmts $	4
5	floatdcl id $\boxed{\text{⟨Dcl⟩ Dcls}}$ Stmts $	2
6	floatdcl id $\boxed{\text{intdcl id}}$ ⟨Dcls⟩ Stmts $	5
7	floatdcl id intdcl id ⟨Stmts⟩ $	3
8	floatdcl id intdcl id $\boxed{\text{⟨Stmt⟩ Stmts}}$ $	6
9	floatdcl id intdcl id $\boxed{\text{id assign ⟨Val⟩ Expr}}$ Stmts $	8
10	floatdcl id intdcl id id assign $\boxed{\text{inum}}$ ⟨Expr⟩ Stmts $	14
11	floatdcl id intdcl id id assign inum ⟨Stmts⟩ $	12
12	floatdcl id intdcl id id assign inum $\boxed{\text{⟨Stmt⟩ Stmts}}$ $	6
13	floatdcl id intdcl id id assign inum $\boxed{\text{id assign ⟨Val⟩ Expr}}$ Stmts $	8
14	floatdcl id intdcl id id assign inum id assign $\boxed{\text{id}}$ ⟨Expr⟩ Stmts $	13
15	floatdcl id intdcl id id assign inum id assign id $\boxed{\text{plus ⟨Val⟩ Expr}}$ Stmts $	10
16	floatdcl id intdcl id id assign inum id assign id plus $\boxed{\text{fnum}}$ ⟨Expr⟩ Stmts $	15
17	floatdcl id intdcl id id assign inum id assign id plus fnum ⟨Stmts⟩ $	12
18	floatdcl id intdcl id id assign inum id assign id plus fnum $\boxed{\text{⟨Stmt⟩ Stmts}}$ $	6
19	floatdcl id intdcl id id assign inum id assign id plus fnum $\boxed{\text{print id}}$ ⟨Stmts⟩ $	9
20	floatdcl id intdcl id id assign inum id assign id plus fnum print id $	7
	f b i a a = 5 b = a + 3.2 p b $	

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

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

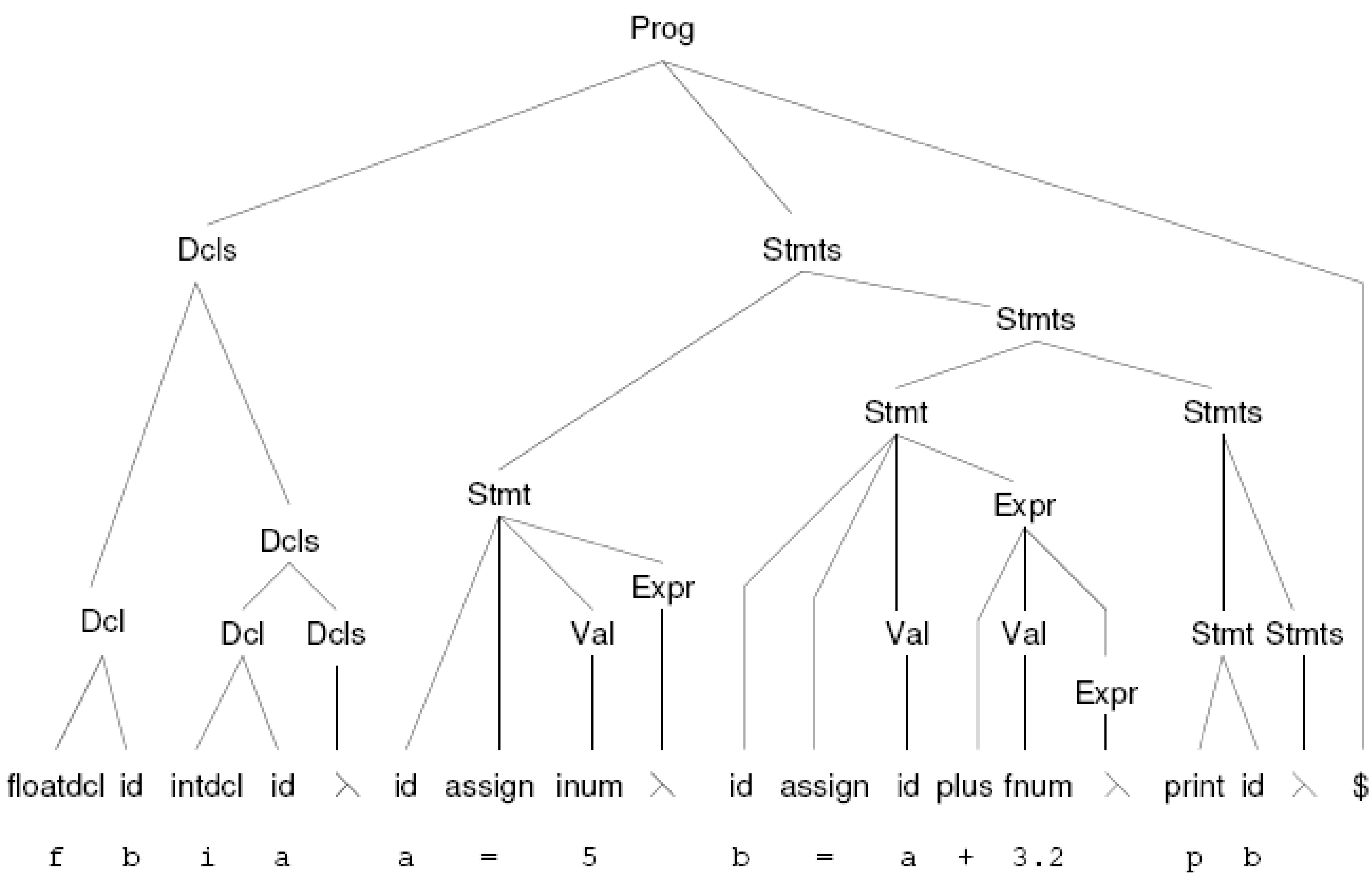


Figure 2.4: An ac program and its parse tree.

Definition of ac language

Regular expression specifies **Token**

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

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

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

Token Specification for ac

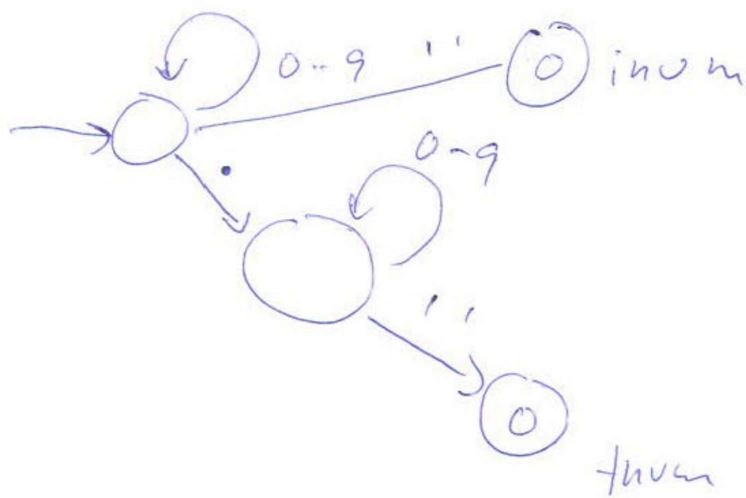
Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	$[a-e] \mid [g-h] \mid [j-o] \mid [q-z]$
assign	"="
plus	"+"
minus	"-"
inum	$[0-9]^+$
fnum	$[0-9]^+ . [0-9]^+$
blank	$(" \ ")^+$

Figure 2.3: Formal definition of ac tokens.

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

Tokens and FSA

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



Phases of an ac compiler

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

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

  - Each token has the two components:

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

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

  - Automatic construction of scanners: Chap.3

Scanning: Divide Input into Tokens

An example ac source program:

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

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

scanner

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

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

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

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

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

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

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

            }
        }

    }
    return ans;
}

/**
```

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

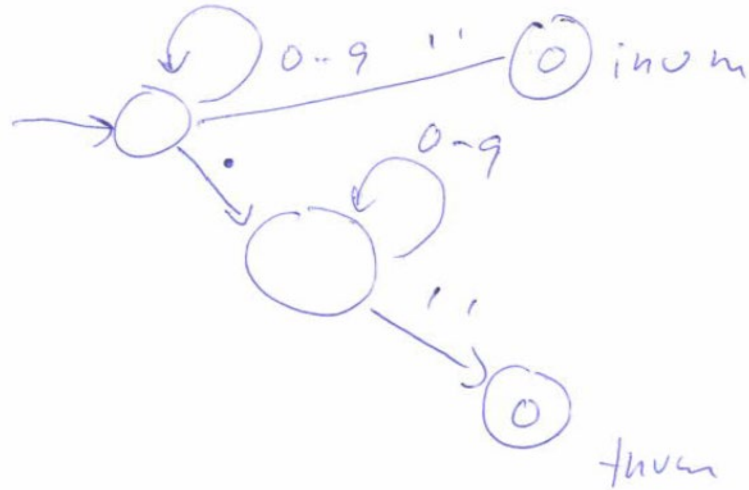


Figure 2.6: Finding inum or fnum tokens for the ac language.

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

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

Pause

Parsing

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

Stmt → id assign Val Expr

```
procedure STMT ( )
    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                                 (6)
        then
            call MATCH (ts, print )
            call MATCH (ts, id )
        else
            call ERROR( )                                     (7)
end
```

Stmt → print id

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

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

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

Figure 2.8: Recursive-descent parsing procedure for Stmts.

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

}

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

}
```

The result of parsing

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

- **AST** serves as a representation of a program for all phases

  after **syntax analysis**.

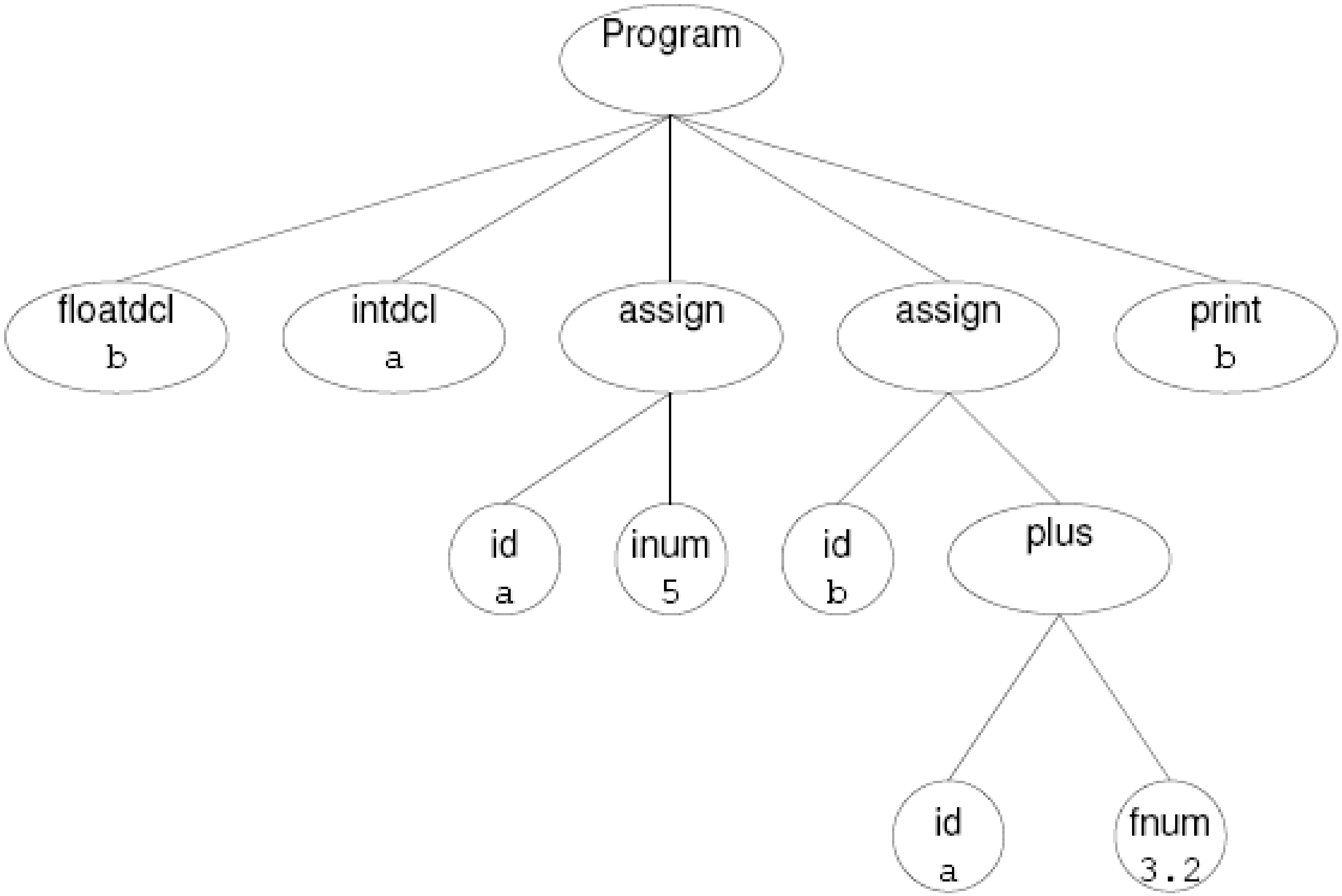


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Abstract Syntax Trees

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

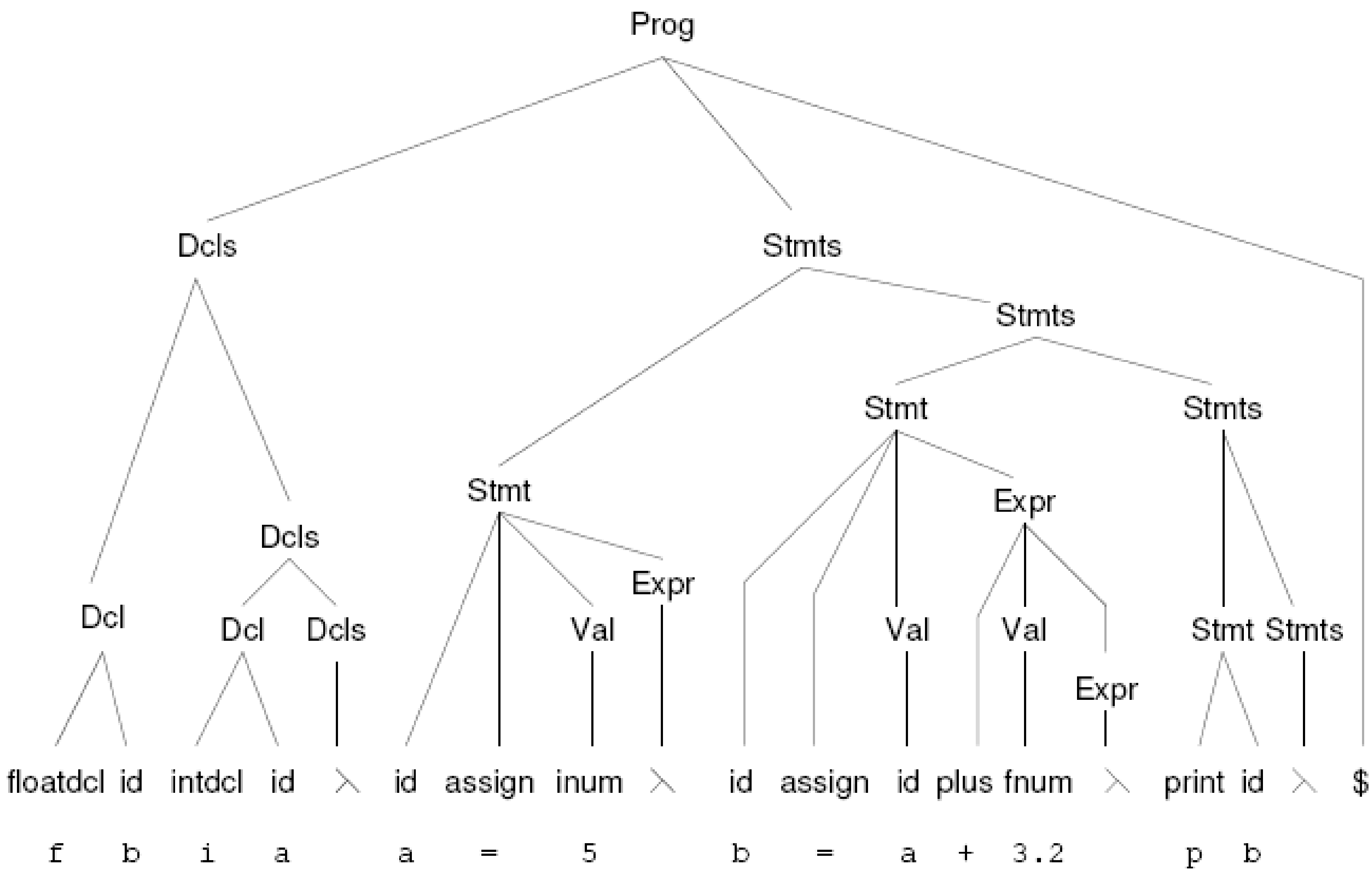


Figure 2.4: An ac program and its parse tree.

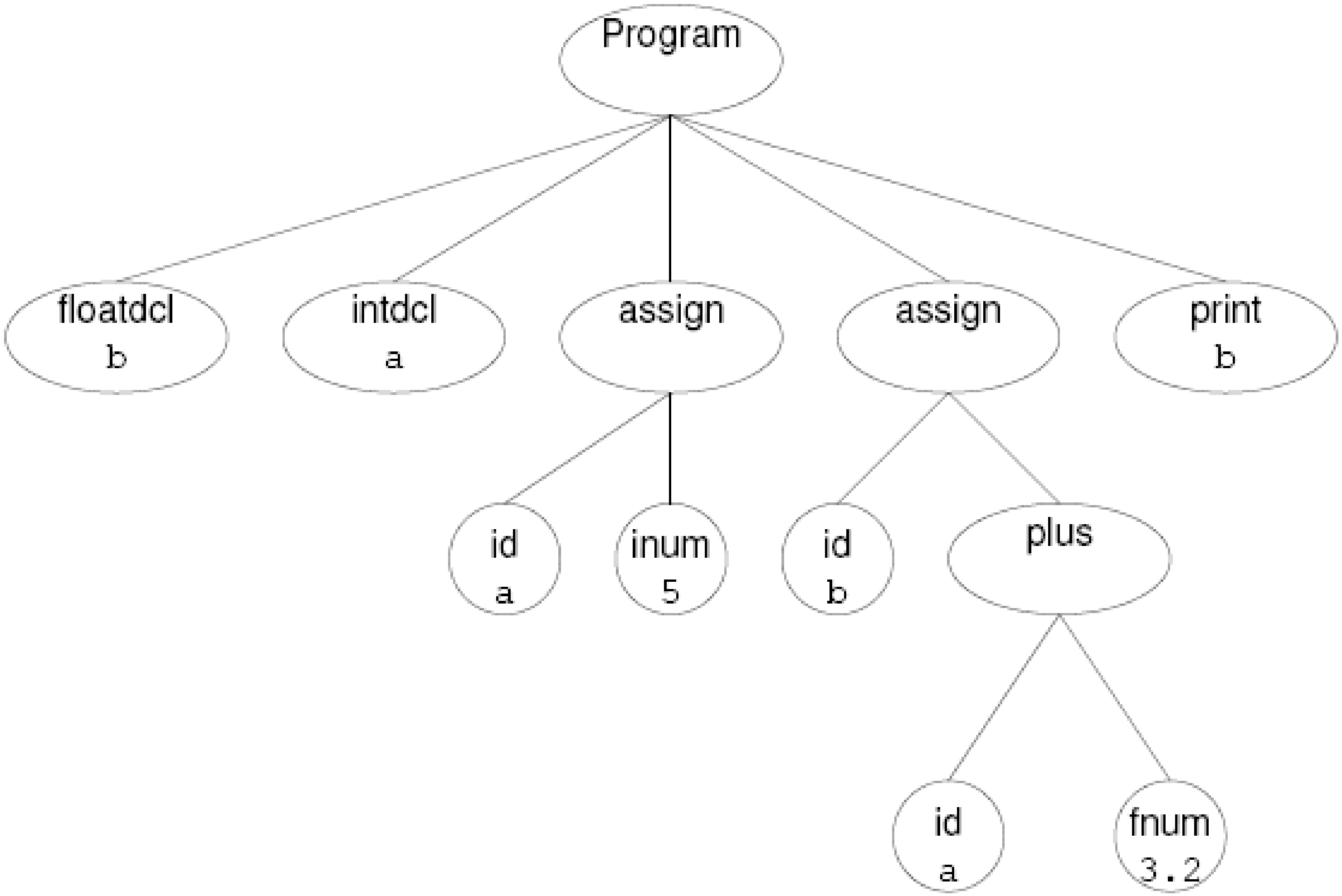


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Contextual Analysis

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

Semantic Analysis

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

Symbol Tables

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

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

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

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

Figure 2.10: Symbol table construction for ac.

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

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

Type Checking

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

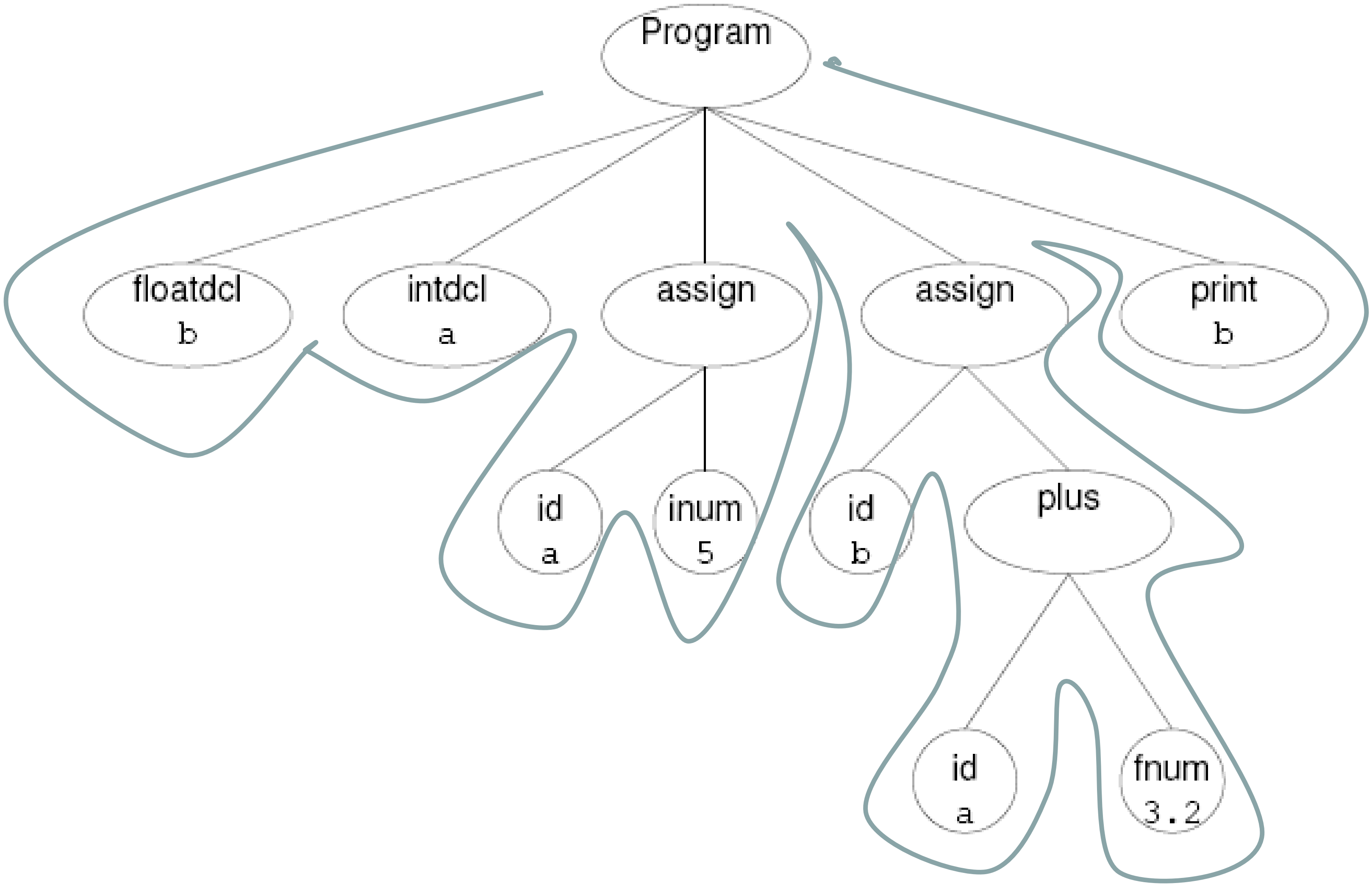


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Phases of an ac compiler (Cont.)

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

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

Type Checking

/⋆ Visitor methods ⋆/

procedure VISIT(*Computing* n)
    $n.type \leftarrow$ CONSISTENT($n.child1, n.child2$)
end
procedure VISIT(*Assigning* n)
    $n.type \leftarrow$ CONVERT($n.child2, n.child1.type$)
end
procedure VISIT(*SymReferencing* n)
    $n.type \leftarrow$ LOOKUPSYMBOL($n.id$)
end
procedure VISIT(*IntConsting* n)
    $n.type \leftarrow$ integer
end
procedure VISIT(*FloatConsting* n)
    $n.type \leftarrow$ float
end

```java
package acASTVisitor;

public class TypeChecker extends Visitor {

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

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

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

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

    }

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

    }
```

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

```java

}*/

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

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

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

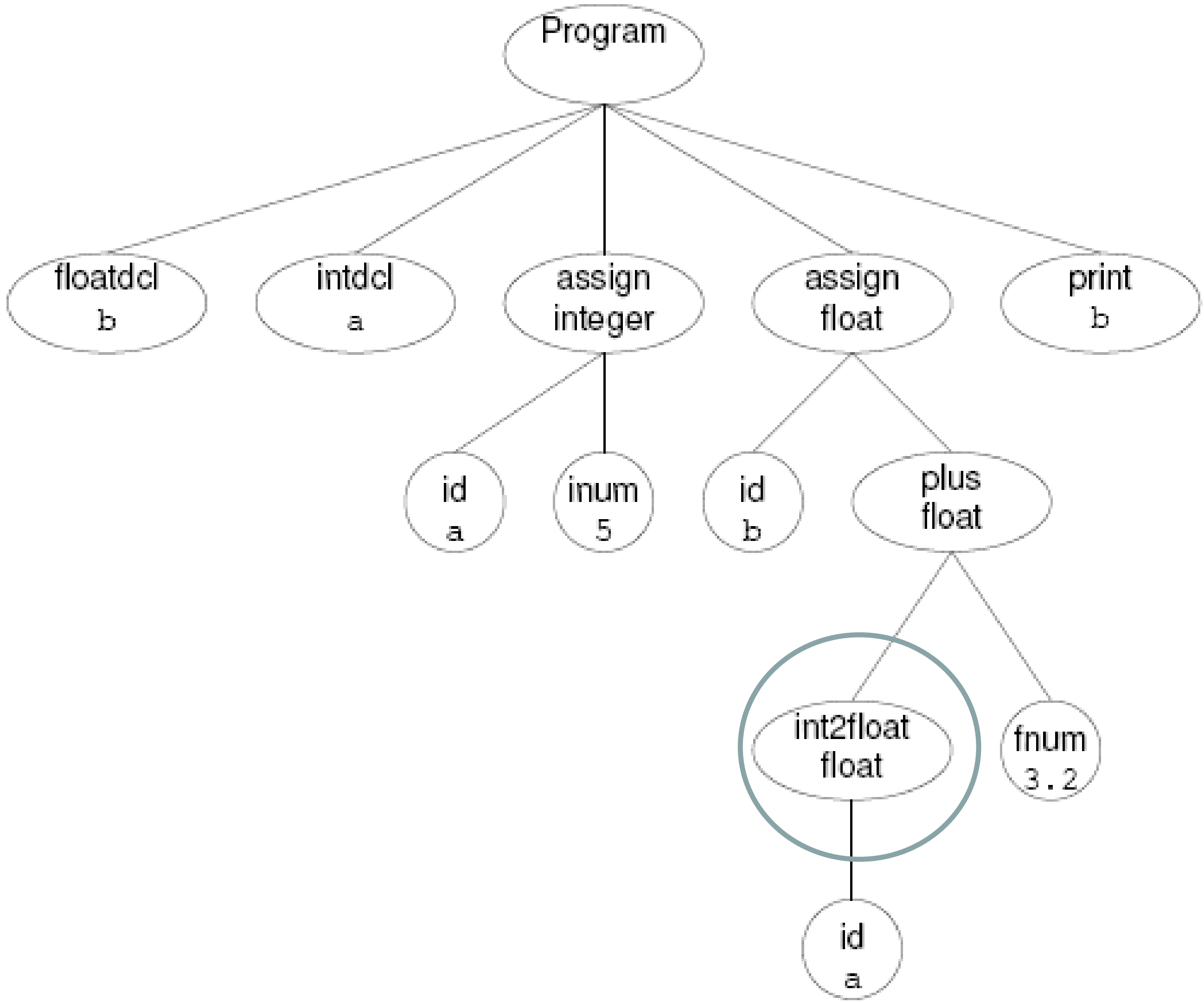


Figure 2.13: AST after semantic analysis.

Code Generation

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

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

Figure 2.14: Code generation for ac

```java
package acASTVisitor;

public class CodeGenerator extends Visitor {

    String code = "";

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

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

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

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

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

    @Override
```

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

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

    @Override
    void visit(SymDeclaring n) {
    }

    @Override
    void visit(FloatDcl n) {
    }

    @Override
    void visit(IntDcl n) {
    }

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


}
```

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

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

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

Some advice

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

Choosing the impl. language

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

input ⟶ Translator ⟶ output

source program — is expressed in the *source language*

Translator — is expressed in the *implementation language*

object program — is expressed in the *target language*

A: All of them!

What can we do now in our projects?

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