Chapter 2

A Simple Compiler

In this chapter, we will describe a simple programming language called:

adding calculator (ac).

Then, describe a simple compiler for ac.

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.

Definition of ac language (Cont.)

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

Figure 2.3: Formal definition of ac tokens.

Sample AC Program

```
f b    // declare a floating variable
i a    // declare an integer variable
a = 5  // a single character identifier
b = a + 3.2
p b    // print the variable
```

Adding calculator (ac) grammar

A context-free grammar (CFG) specifies the syntax of a language. It is:

A set of **productions**

(or **rewriting rules**).

The grammar for ac:

```
 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.

Definition of ac language

- Two kinds of grammar symbols:
  terminal (token) and **non-terminal**.
  1) A terminal cannot be rewritten.
  2) A non-terminal can be rewritten by a production rule.

- A special non-terminal is **start symbol**, which is usually the symbol on the **left-hand side (LHS)** of the grammar's first rule.

- From the start symbol, we proceed by replacing (rewriting) a symbol with the **right-hand side (RHS)** of some production of that symbol.

Definition of ac language (Cont.)

- The symbol **λ** denotes **empty** or **null string**, which indicates that there are no symbols on a production's RHS.
- The special symbol **$** represents the end of the input stream.
- To show how the grammar in Fig. 2.1 defines **ac** programs, the derivation of one ac program is given in Fig. 2.2, beginning with:

  The start symbol **Prog**.

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

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

Definition of ac language (Cont.)

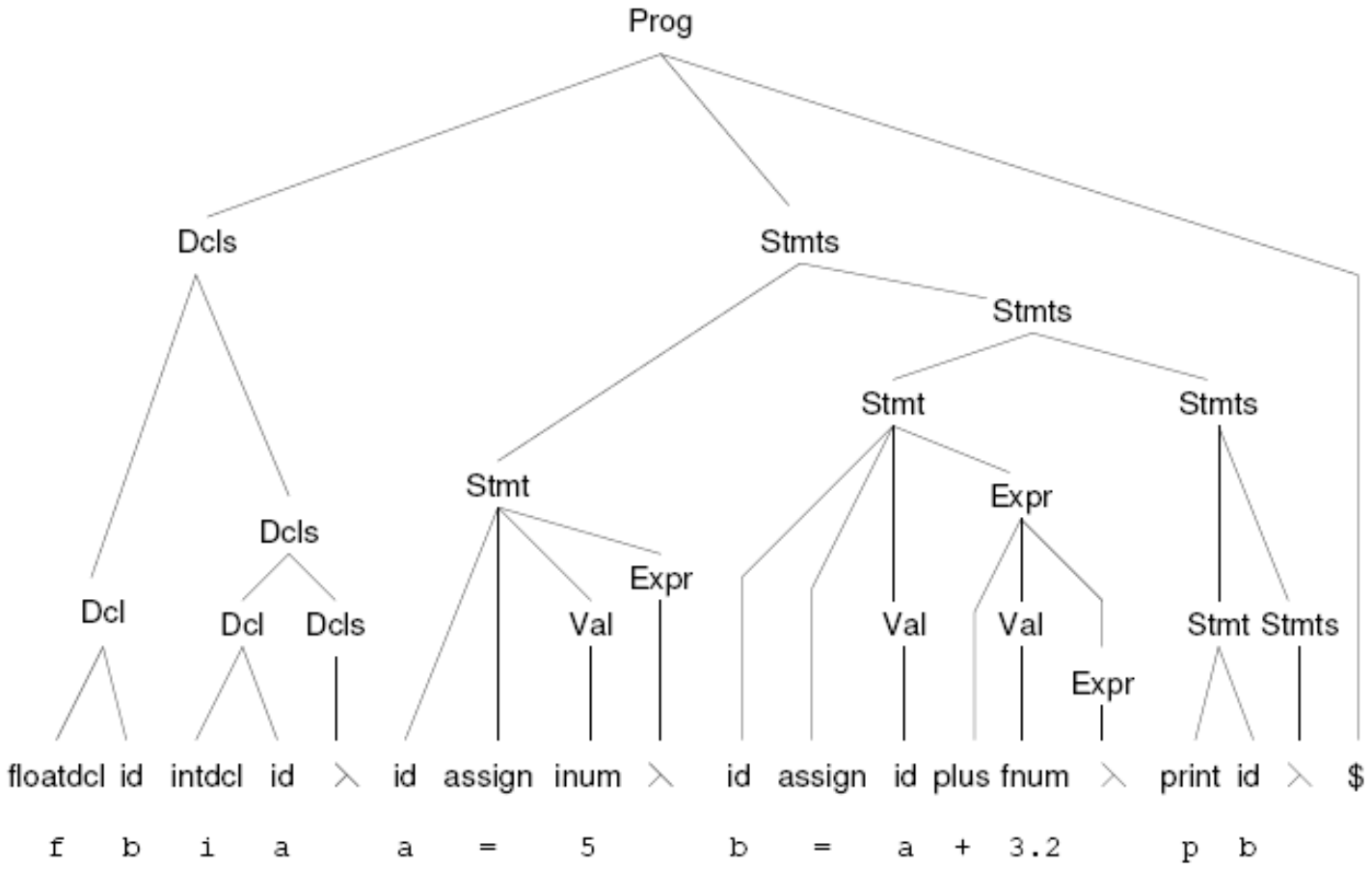


Figure 2.4: An ac program and its parse tree.

Phases of an ac compiler

- Scanning
  - 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")

```
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.

```
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.

Phases of an ac compiler (Cont.)

- Parsing
  - **Parser** processes tokens produced by the scanner, determines the syntactic validity of the token stream, and creates an **abstract syntax tree (AST)** for subsequent phases.
  - **Recursive descent** (LL type parser) is one simple parsing technique used in practical compilers.

    The parsing procedure for “Stmt” is shown in Fig. 2.7.

Definition of ac language (Cont.)

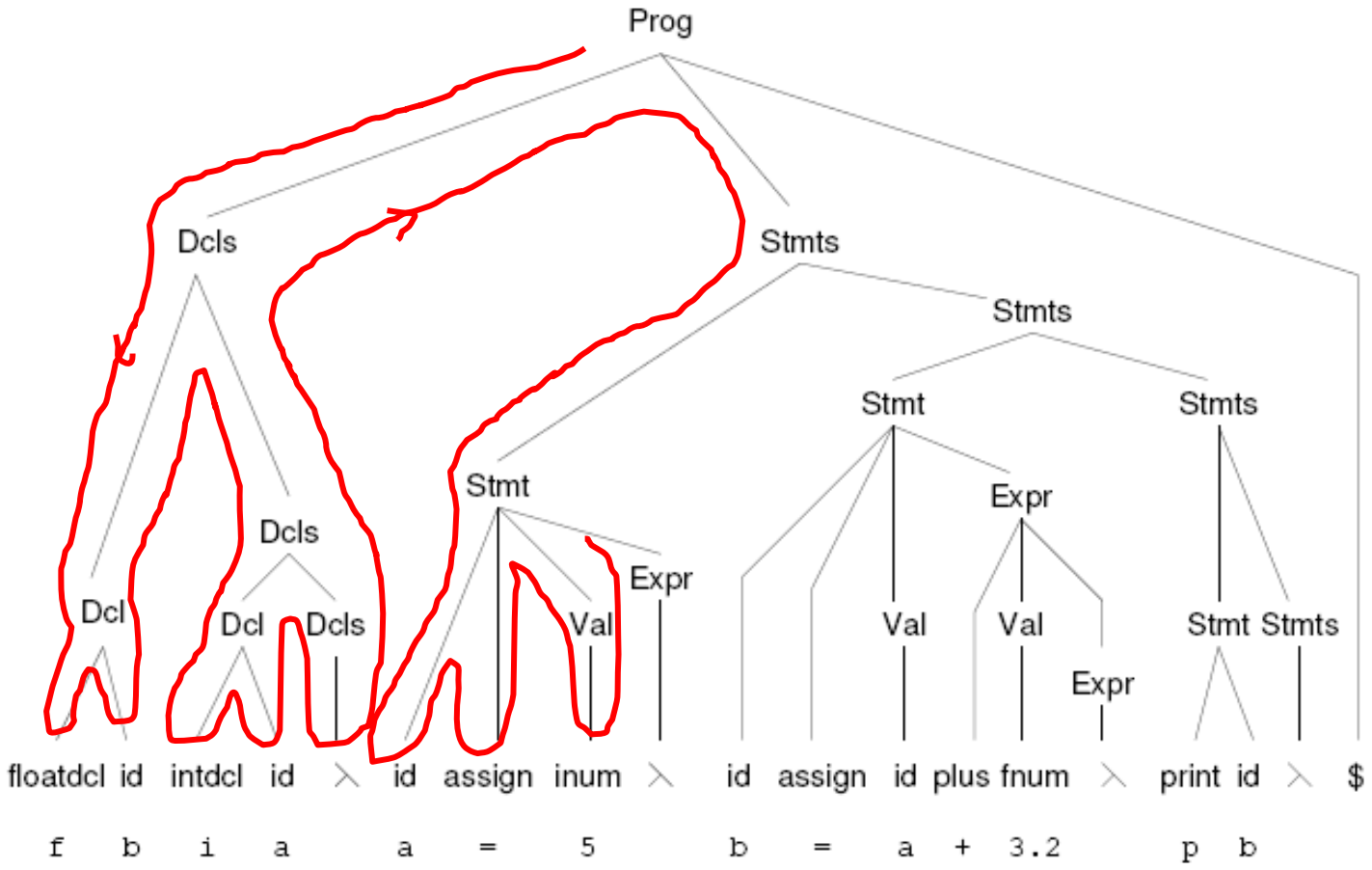


Figure 2.4: An ac program and its parse tree.

Phases of an ac compiler (Cont.)

- Each parsing procedure examines the next input token to predict which production to apply. Ex:

Stmt offers two productions:

Stmt → id assign Val Expr

Stmt → print id

Phases of an ac compiler (Cont.)

- If **id** is the next input token, the parse proceeds with the production:

  Stmt → id assign Val Expr

  and the **predict set** for the production is {id}.

- If **print** is the next, the parse proceeds with:

  Stmt → print id

  the **predict set** for the production is {print}.

- If the next input token is neither **id** nor **print**, neither rule can be applied. (aka, syntax error)

Notes

- OK, we can peek (lookahead) to choose a production to expand (rewrite) if two or more productions are available
- If a lookahead cannot determine a single production to rewrite, what happens? (See future chapters)
  1. Rewrite grammar
  2. Peek more lookaheads

Phases of an ac compiler (Cont.)

Computing the predict sets used in Stmt is relatively easy,

because each production for Stmt begins with a **distinct terminal symbol:**

id or print

In fig 2.7, markers 1 and 6 pick which of the two productions to apply by examining the next input token.

```
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
```

```
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
```

```
 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
```

For each production, write the LHS code

```
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
        end
```

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

Phases of an ac compiler (Cont.)

- Consider the productions for Stmts:
  - Stmts → Stmt Stmts
  - Stmts → λ

- The predict sets for Stmts can be computed by inspecting the following:
  - Stmts → Stmt Stmts

    begins with the non-terminal Stmt.
    - Sol: Find those symbols that predict **any** rule for Stmt.
  - Stmts → λ derives no symbols.
    - Sol: Look for what symbol(s) could occur **following** such a production (that is $).

For production rule that begins with a nonterminal you still need to tell how to keep going based on peeking lookahead

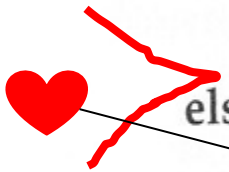
```
procedure STMTS( )
    if ts.PEEK( ) = id or ts.PEEK( ) = print      (8)
    then
        call STMT( )                              (9)
        call STMTS( )                             (10)
    else
        if ts.PEEK( ) = $                         (11)
        then
            /*  do nothing for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

```
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 2.8: Recursive-descent parsing procedure for Stmts.

Let's do semantics !!

```
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
```



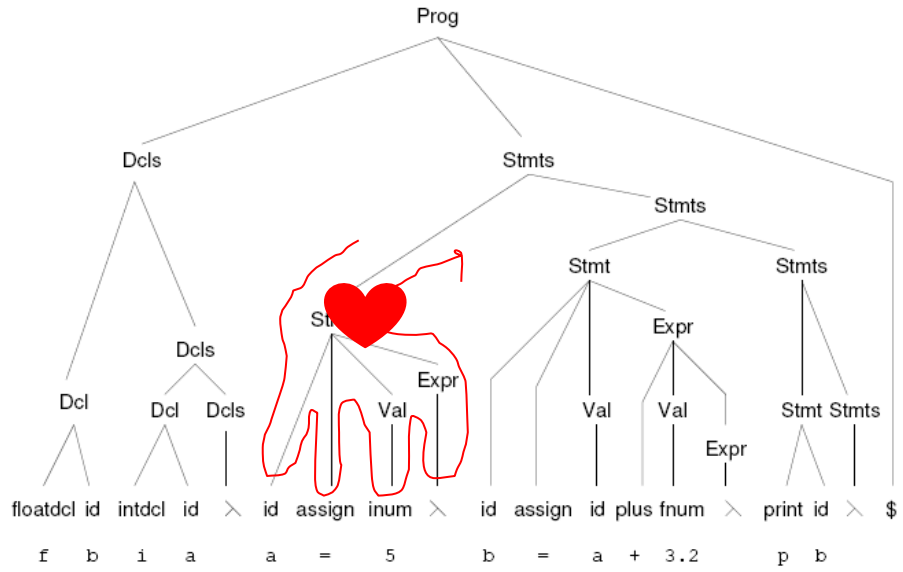


Figure 2.4: An ac program and its parse tree.

OK, when parser does parsing, there are some points that reach a point where you can do semantics. We should do a=5 here

Semantics Actions

- You can execute semantics immediately in an interpreter
- In a Compiler,
  - we don't execute semantics immediately
  - We do not produce assembly code yet.

Phases of an ac compiler (Cont.)

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

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

  after **syntax analysis**.

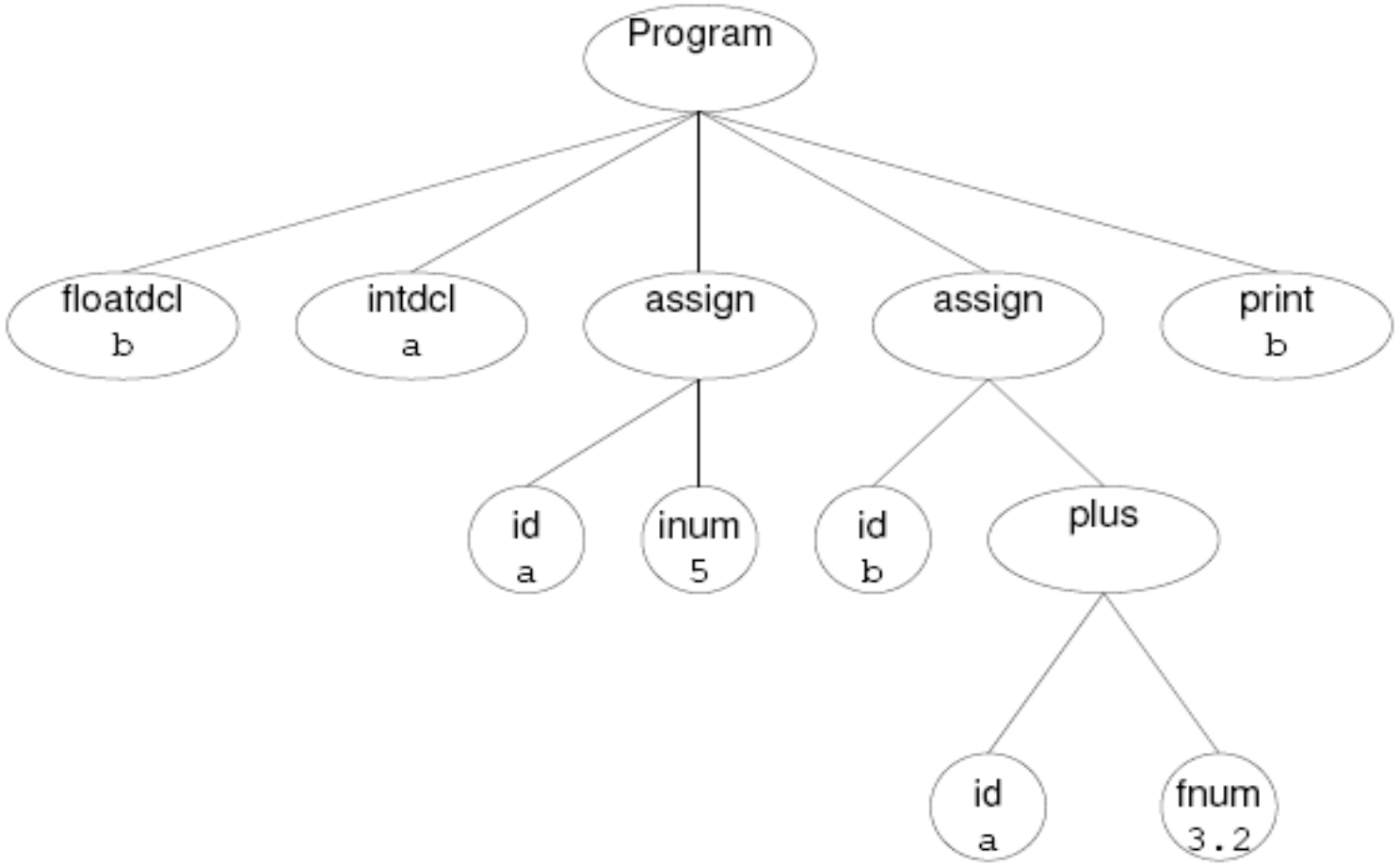


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Phases of an ac compiler (Cont.)

The symbol-table constructor traverses
the AST to record
all *identifiers* and
their *types*
in a **symbol table**.

The constructor is shown in fig 2.10.

Phases of an ac compiler (Cont.)

```
/*    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.

Phases of an ac compiler (Cont.)

An **ac** symbol table has 23 identifier entries with their types:

integer, float, or unused (null).

Fig 2.11 shows the symbol table constructed.

Phases of an ac compiler (Cont.)

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.

Phases of an ac compiler (Cont.)

- Type checking
  - The ac language offers two types:

    integer and *float*,

    and 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.

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.

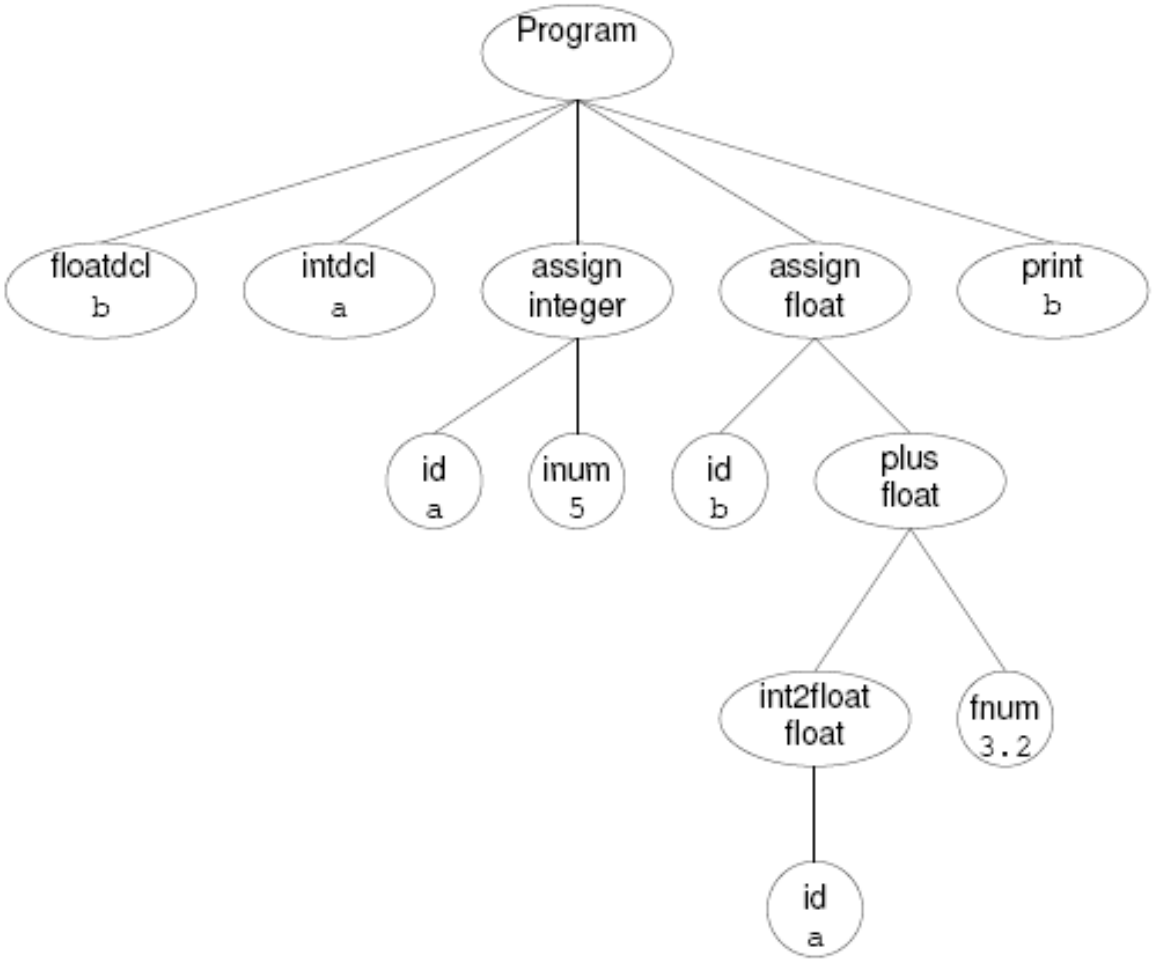


Figure 2.13: AST after semantic analysis.

Phases of an ac compiler (Cont.)

- Code generation
  - Code generation proceeds by traversing (visiting) the AST nodes, starting at its root and working toward its leaves.
  - A visitor generates code based on the node's type (fig 2.14).
  - Fig 2.15 shows the target code generated from the AST in fig 2.9.

```
procedure VISIT( Assigning n )
    call CODEGEN( n.child2 )
    call EMIT( "s" )
    call EMIT( n.child1.id )
    call EMIT( "0 k" )
end
procedure VISIT( Computing n )
    call CODEGEN( n.child1 )
    call CODEGEN( n.child2 )
    call EMIT( n.operation )
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" )
end
procedure VISIT( Converting n )
    call CODEGEN( n.child )
    call EMIT( "5 k" )
end
procedure VISIT( Consting n )
    call EMIT( n.val )
end
```

(14) (15) (16) (17)

Figure 2.14: Code generation for ac

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

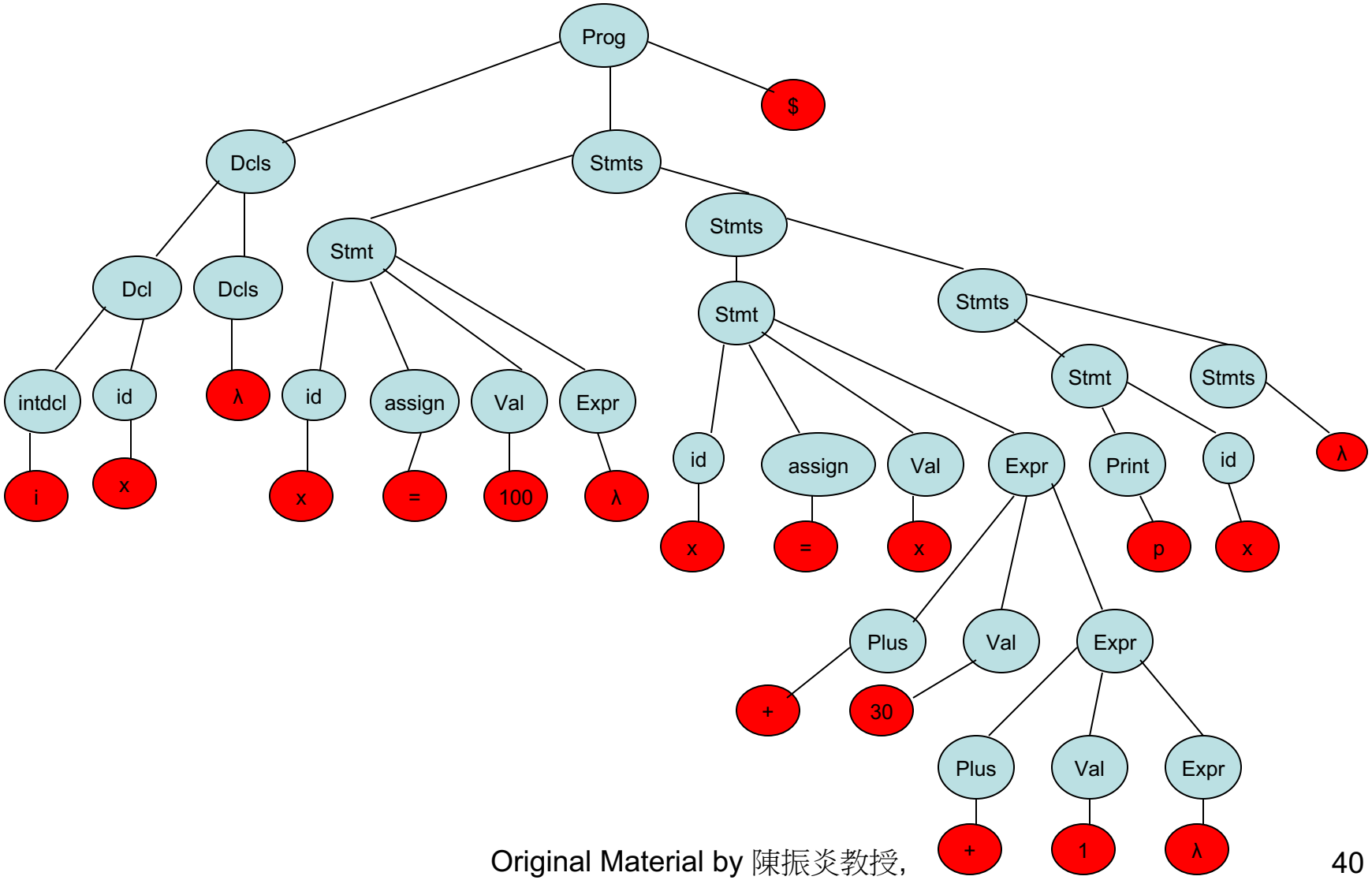
Figure 2.15: Code generated for the AST shown in Figure 2.9.

Homework

1. Based on the grammar in Figure 2.1, construct the parse tree for the following input:

   (a) i x x=100 x=x+30+1 p x

HW Solution 1.(a)



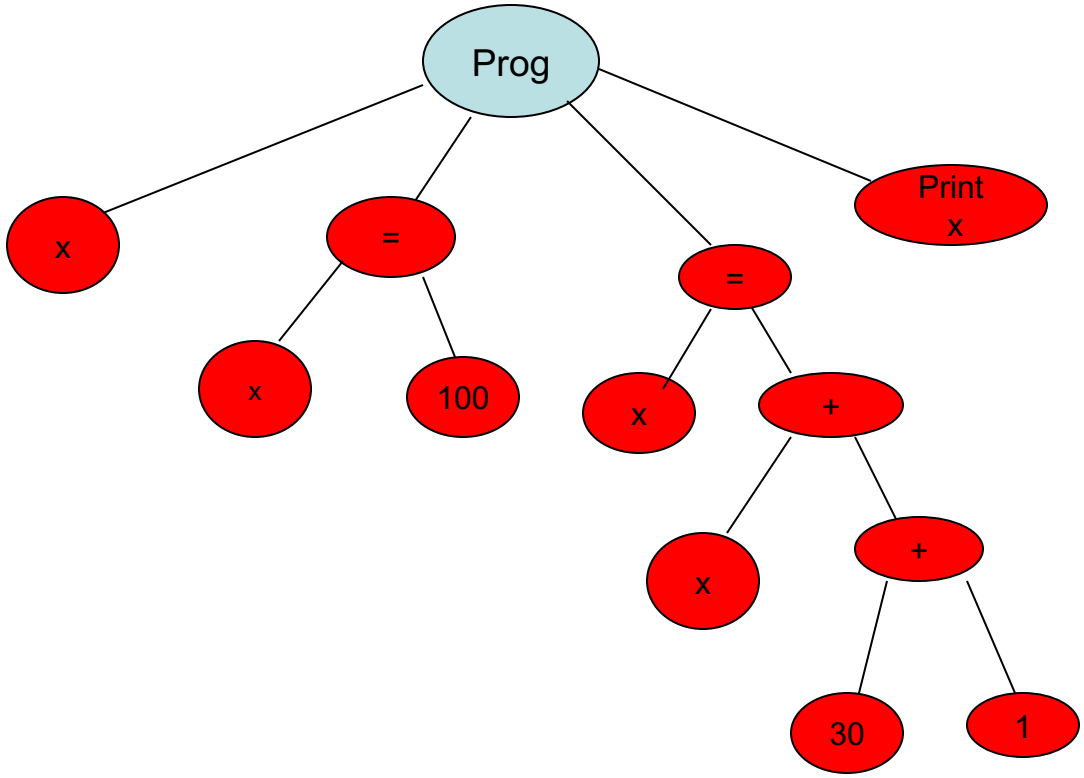
Homework (Cont.)

- 2. For the input shown in Exercise 1:
  (a) Construct the input's AST

  (b) According to the definition of **ac**, is the input semantically correct? If not, what changes to the **ac** language would render the input semantically correct?

  (c) With **ac** changes in place that make the input semantically correct, show the code that would be generated on behalf of the input.

HW Solution 2.(a)



HW Solution 2.(b)

(b) No.

we can do semantic analysis below.

In the AST, the three nodes 30, +, and 1 can be changed to one node 31.

This analysis simplifies the AST.

HW Solution 2.(c)

(c) 100	x=100	push 100 to stack
sx		pop top of stack, store it to reg. x
0 k		set precision 0
lx	x=x+31	load x (push x to stack)
31		push 31 to stack
+		pop 31, pop x, add them, push the result
sx		
0 k		
lx	p x	load x (push x to stack)
p		print top of stack
si		pop top of stack, store it to reg. i

Homework (Cont.)

- 3. Extend the ac scanner (Figure 2.5) in the following ways:

  (a) A floatdcl can be represented as either f or float, allowing a more Java-like syntax for declarations.

  (b) An intdcl can be represented as either i or int

  (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)

HW Solution 3

(a) (b)

Terminal	RegularExpression
floatdcl	“f” | (“f” “l” “o” “a” “t”)
intdcl	“i” | (“i” “n” “t”)

(c) inum [0-9]$^{+}$ e -? [0-9] $^{+}$

fnum [0-9] . [0-9]$^{+}$ e -?[0-9] $^{+}$