Compiler Technology of Programming Languages

Chapter 2

A Simple Compiler

Prof. Farn Wang

Design of a Simple Compiler

- Build a simple compiler for a very small language: **AC (Adding Calculator)**.
  - Computation and printing of numerical values for integer and FP variables
- Although simple, AC serves the purpose of studying the *phases* and *data structures* of a compiler.
- This chapter covers basic techniques that will be discussed in more details in Chapter 3 through Chapter 9

Informal **AC** definition

- Types
  - Support only two types: integer and float
  - Integer: a sequence of decimal numerals.
  - Float: allows 5 fractional digits after the decimal point.
- Keywords
  - Keywords vs. Reserved words
  - All languages have some keywords such as **if, while**, …
  - Three **reserved** words in AC: **f** (declare for float), **i** (declare for integer), **p** (print the value of a variable).
- Variables
  - AC offers 23 variable names, from **a** to **z,** excluding the three reserved words: **i, f**, and **p**.
  - Variables must be declared before using them

Informal **AC** definition (cont.)

Type Conversion

- Implicit type conversion is also called *type coercion.* ***e.g.***

  int I; float f; double d;

  f = I; f = I*1.25+3;   d = d+f;

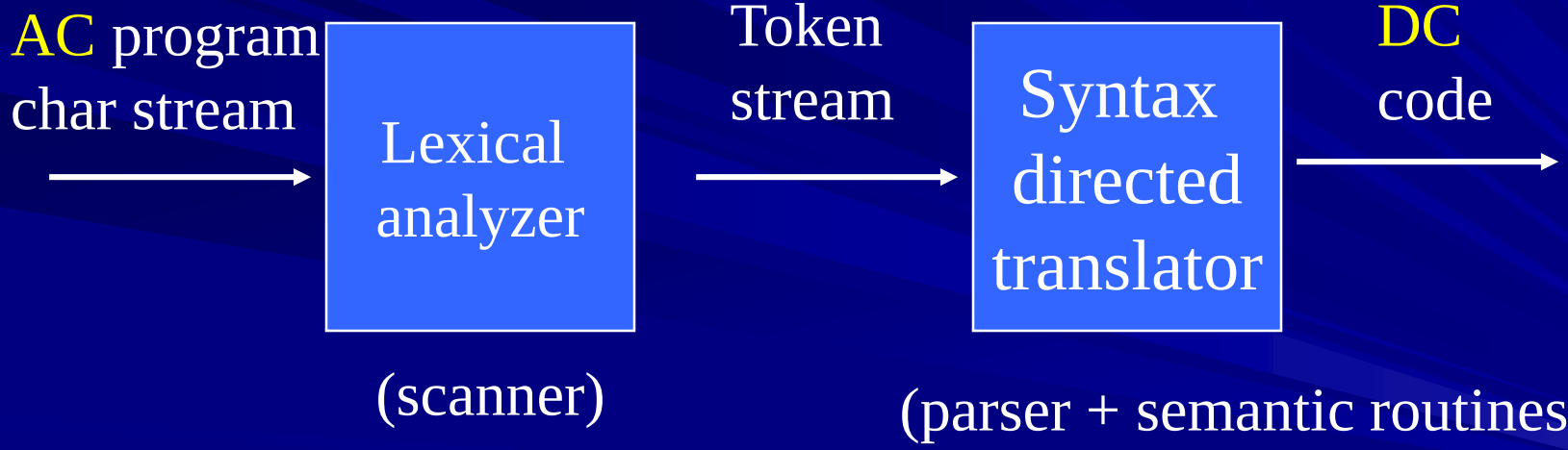
  Quiz:

  unsigned ux;

  Is (ux > -1) TRUE or FALSE?

- Explicit type conversion is often called *type casting*
- ***AC*** allows type coercion from integer to float, but not the other direction.

Structure of the AC Compiler



AC-to-DC Compiler or AC-to-DC Converter

Formal **AC** definition

- Must define the *syntax* and the *semantics*
- Use a CFG (Context Free Grammar) to describe its syntax
- Use RE (Regular Expression) to define its token specification

Formal definition
- enables automatic compiler construction
- minimizes ambiguity

An AC Scanner

- There are 10 tokens in AC.
- our scanner will be defined as a function that

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.

Which tokens need attributes?

Id, inum, fnum

Syntax Definition

- CFG (Context Free Grammar) is a set of rewriting rules. It is also called *Backus-Naur Form* (BNF) grammar.
- The rewriting rule is called a *production*.
- **Example:**

  A → B C D … Z

  where
  - **A** is the left-hand side (LHS) of the production.
  - **B C D … Z** is the right-hand side (RHS).
  - This means any occurrence of A can be replaced by the RHS.

Production Example

<program> → ***begin*** <stmt_list> ***end***

- LHS is the name of the syntactic construct; the RHS shows a possible form of the syntactic construct.
- LHS are always *nonterminals*, RHS can have *terminals* and *nonterminals*.
- "***begin***", "***end***" lexical elements are called *tokens*, or *terminal* symbols
- For example, <program> is a *nonterminal*. <stmt_list> is also a *nonterminal*.

CFG

- A set of tokens
- A set of non-terminals
- A set of productions
- A start symbol (one of the non-terminals)

- *A string is derived by repeatedly applying productions to a non-terminal symbol*
- *The **language** defined by a CFG is the token strings that can be derived from the **start** symbol*

Recursion and Empty String

- Recursion enables repetitions

  Example:

  <stmt_list> → <stmt> <stmt_tail>

  <stmt_tail> → <stmt> <stmt_tail>

- The empty or null string is represented by λ or ε

  Example:

  <stmt_tail> → λ

  The empty string is very useful in representing optional items.

Syntax of 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
```

$ means end-of-file

Figure 2.1: Context-free grammar for ac.

Syntax of 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
```

f

i

id

=

p

+

-

I number

f number

blank

All tokens in AC

Figure 2.1: Context-free grammar for ac.

```
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
```

integer

float

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

Quick Overview

- **A Simple PL** **AC**
- **Some formalisms are used** **CFG + RE**
- **Why do we need formalism**

  Precise definition + automation

- **What is the job of a scanner?**

  Grouping chars into tokens

- **What are the four components of a CFG?**

  Tokens, Non-terminals, Productions, Start symbol

How to derive an AC program

- Remember:

*The **language** defined by a CFG is the token strings that can be derived from the start symbol*.

- Let us check if the following code fragment is a valid AC program?

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

First, the scanner converts this program to

floatdcl id intdcl id id assign inum id assign id plus fnum print id $

Step	Sentential Form	uction nber
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	8
10	floatdcl id intdcl id id assign inum ⟨Expr	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
	rint id $	7

1. prog → dcls stmts $
2. dcls → dcl dcls
3. | λ
4. dcl → floatdcl id
5. | intdcl id
6. stmts → stmt stmts
7. | λ
8. stmt → id assign val expr
9. stmt → print id
10. …

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

ar in

Figure 2.1

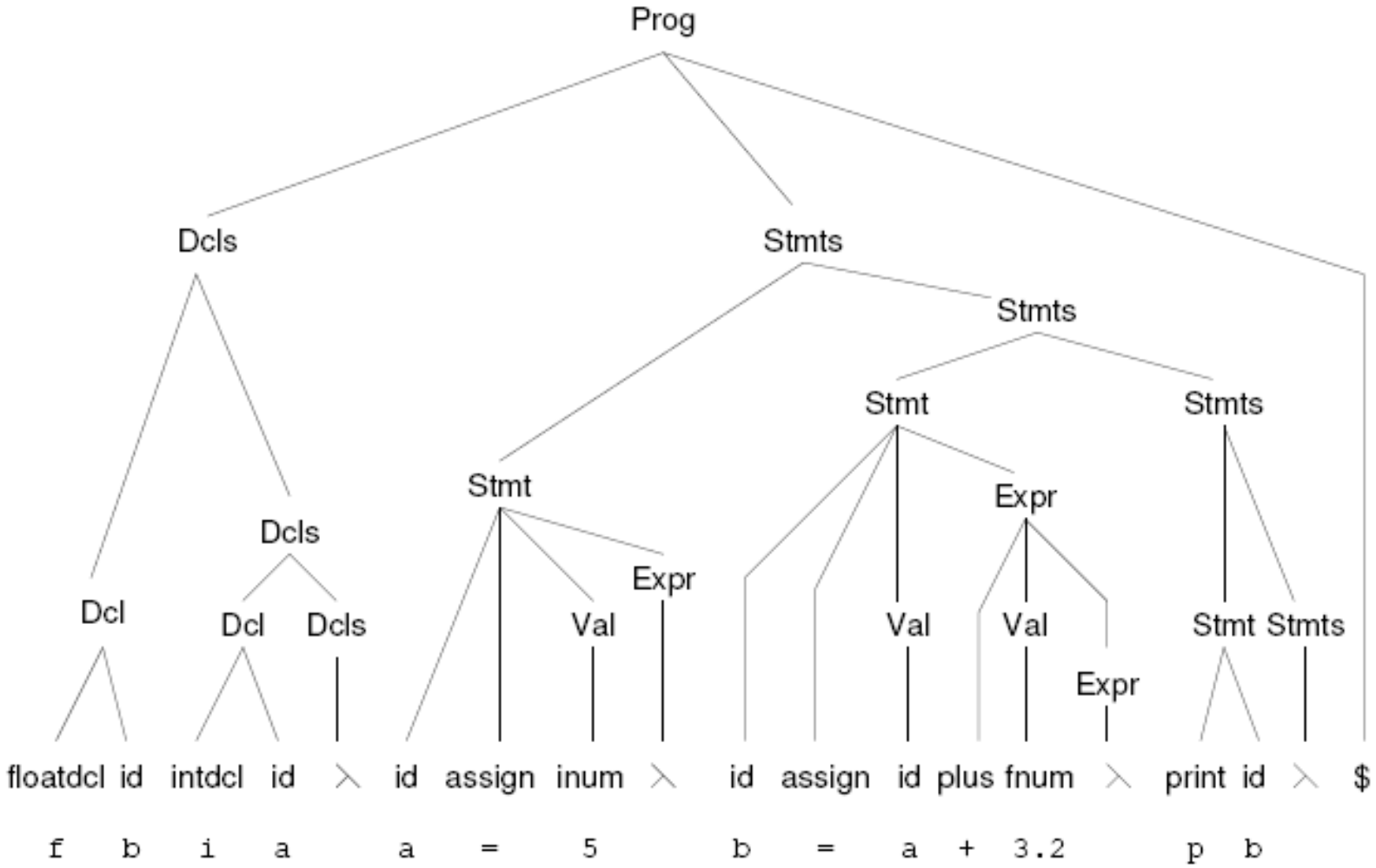


Figure 2.4: An ac program and its parse tree.

Quiz: Which of the following are in the language defined by the AC ***syntax***

a) f b i a a = 0 b = a + 3.2 p b

b) p 15

c) An empty file

d) f b a = 1.0 i c c = 0 p a

e) f b i a a = 0 b = - a - 3.2 p b

Quiz: Which of the following are in the language defined by the AC **syntax**

a) f b i a a = 0 b = a + 3.2 p b

b) p 15

c) An empty file

d) f b a = 1.0 i c c = 0 p a

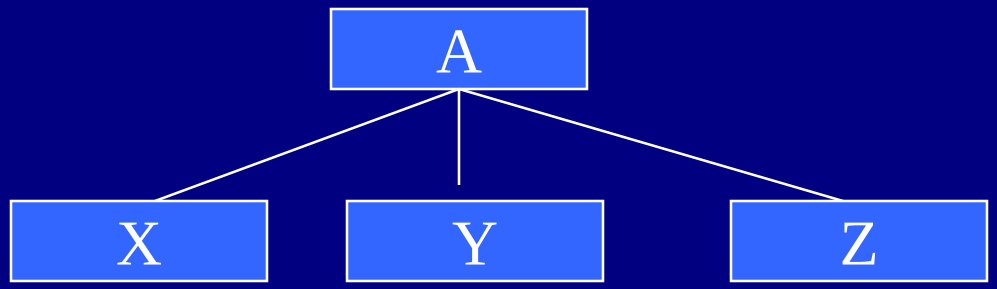
e) f b i a a = 0 b = - a - 3.2 p b

8. stmt → id assign val expr
9. stmt → print id
10. expr → plus val expr
11. | minus val expr
12. | λ
13. val → id
14. | inum
15. | fnum

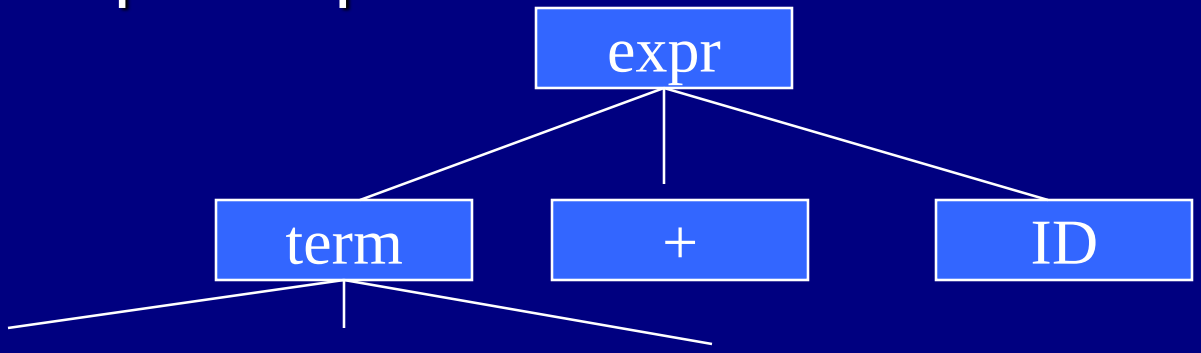
Parse Tree

- A Parse Tree shows how the start symbol derives a string.

  Example: A→ XYZ



Example: expr → term + ID



Example

A subset of C grammar:

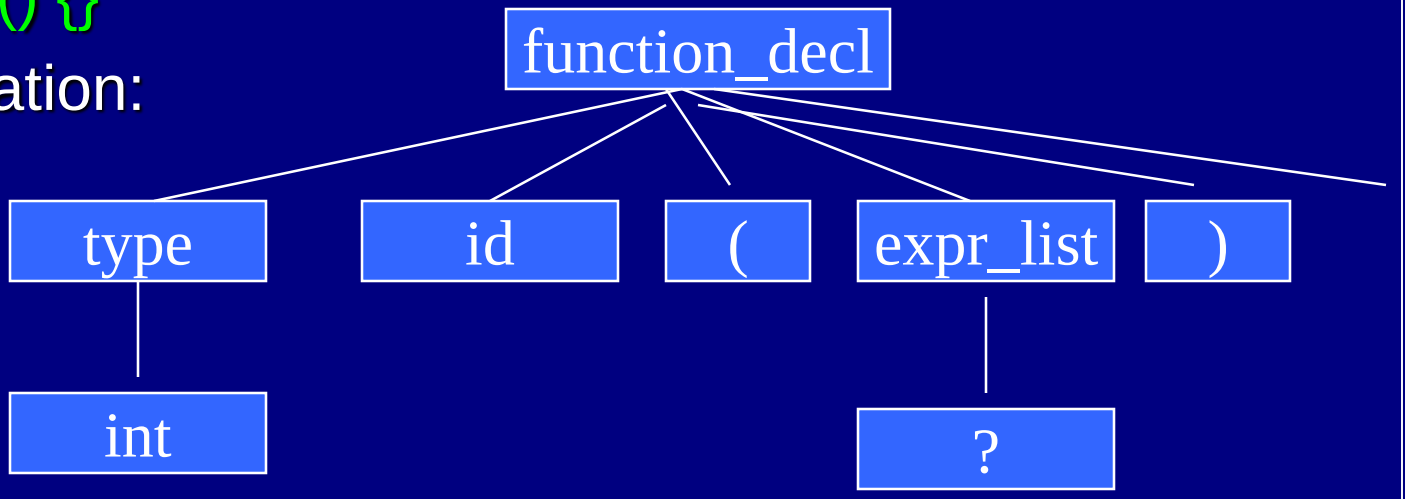
<function_decl> → <type> id (<expr_list>) { <block> }

<block> → <decl_list> <statement_list> | λ

A C program – a token string

int func() {}

A derivation:



Parsing: to come up with the parse tree and validate the token string is a correct C program.

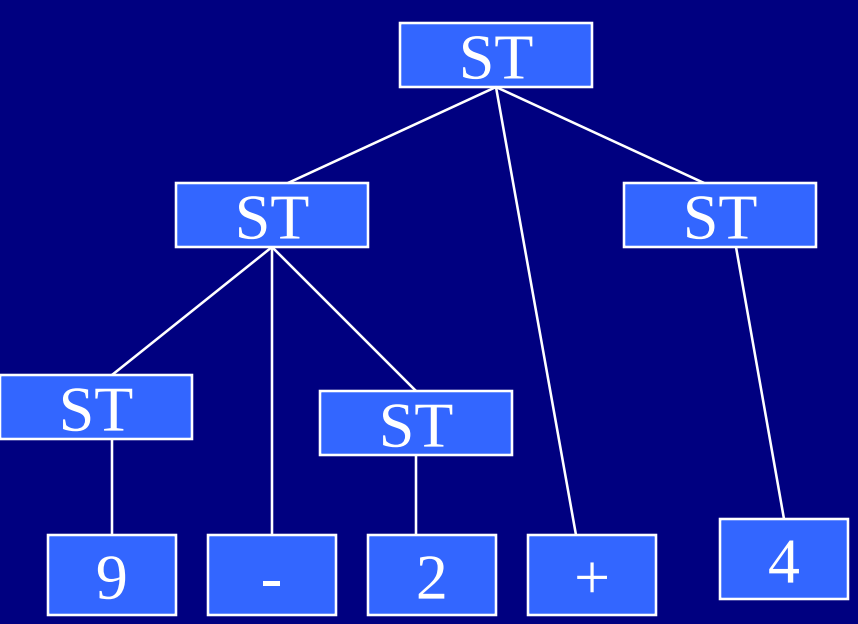
Parse Tree

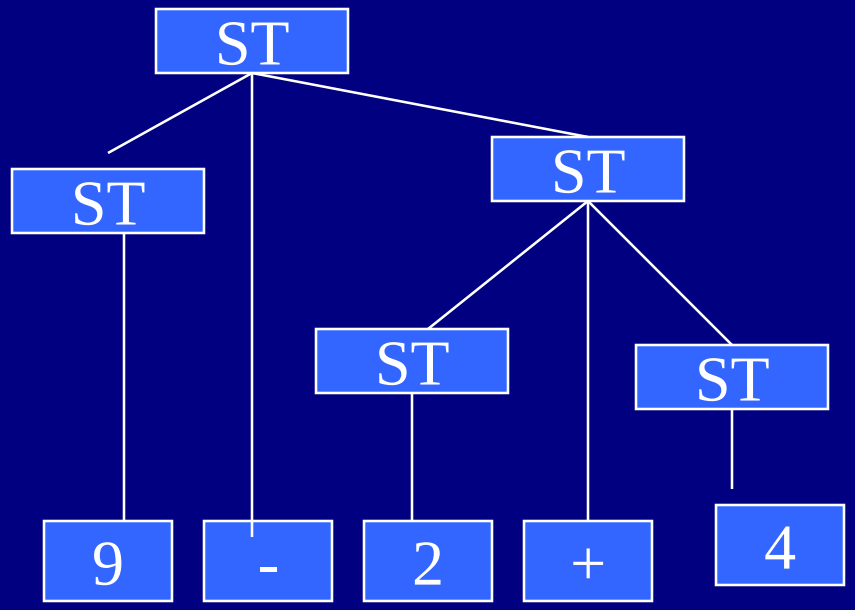
- A Parse Tree has the following properties
  - The root is labeled by the start symbol
  - Each leaf is labeled by a token or by λ
  - Each internal node is labeled by a non-terminal
  - If A is the non-terminal labeling, and X,Y,Z are labels of the children of A from left to right, then A→ XYZ is a production
- *Parsing* is the process of finding a parse tree for a given string of token.
- If a grammar can have more than one parse tree for a given string, this grammar is *ambiguous*.

Ambiguous Grammar

- Example CFG
  - String → String + String
  - String → String – String
  - String → digit

The string 9-2+4 can have two parse trees





Association and Precedence

- Left Association and Right Association
  - An operator associates to the left if an operand has operators on both side, and the operand is taken by the operator to its left.
  - Left associative operators:
    - +, -, *, /
    - e.g. A + B + C, A-B-C, A*B*C
  - Right associative operators:
    - Exponential and the assign operator
    - e.g. A = B =C, (A?B:C?D:E)

Exercise

1. <expr> → <expr> { <add_op> <primary>}

+ and – are left associative operators, see how a string is derivated.

→ A {+ B} {+ C……

In the above example, B will be associated with the left operator

However, if the operator is right associative, how would the CFG be different? Consider the assignment operator, for example.

Exercise (cont.)

<expr> → {<primary> <assign_op> } <expr>

→ …. {A= } { B = } C

In the above example, B will be associated with the right operator.

Another way to define the production is

<expr> → <primary> <assign_op> <expr>

Use right recursion instead.

Exercise (cont.)

CFG1. <expr> → <expr> OP <primary>

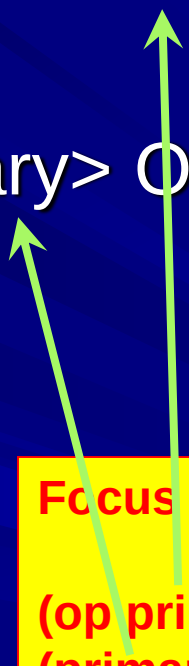
CFG2. <expr> → <primary> OP <expr>

In CFG1, OP is ?

Left Associative

In CFG2, OP is

Right Associative



Focus on what is repeating!!

(op primary) → left associative
(primary op) → right associative

Precedence of Operators

* (multiply) and / (divide) have higher precedence than + and -. How is precedence of operators handled?

- We can create two non-terminals for the two levels of precedence.
- Example CFG
  - Expr → expr + term | expr – term | term
  - Term → term * factor | term / factor | factor
  - Factor → digit | (expr)

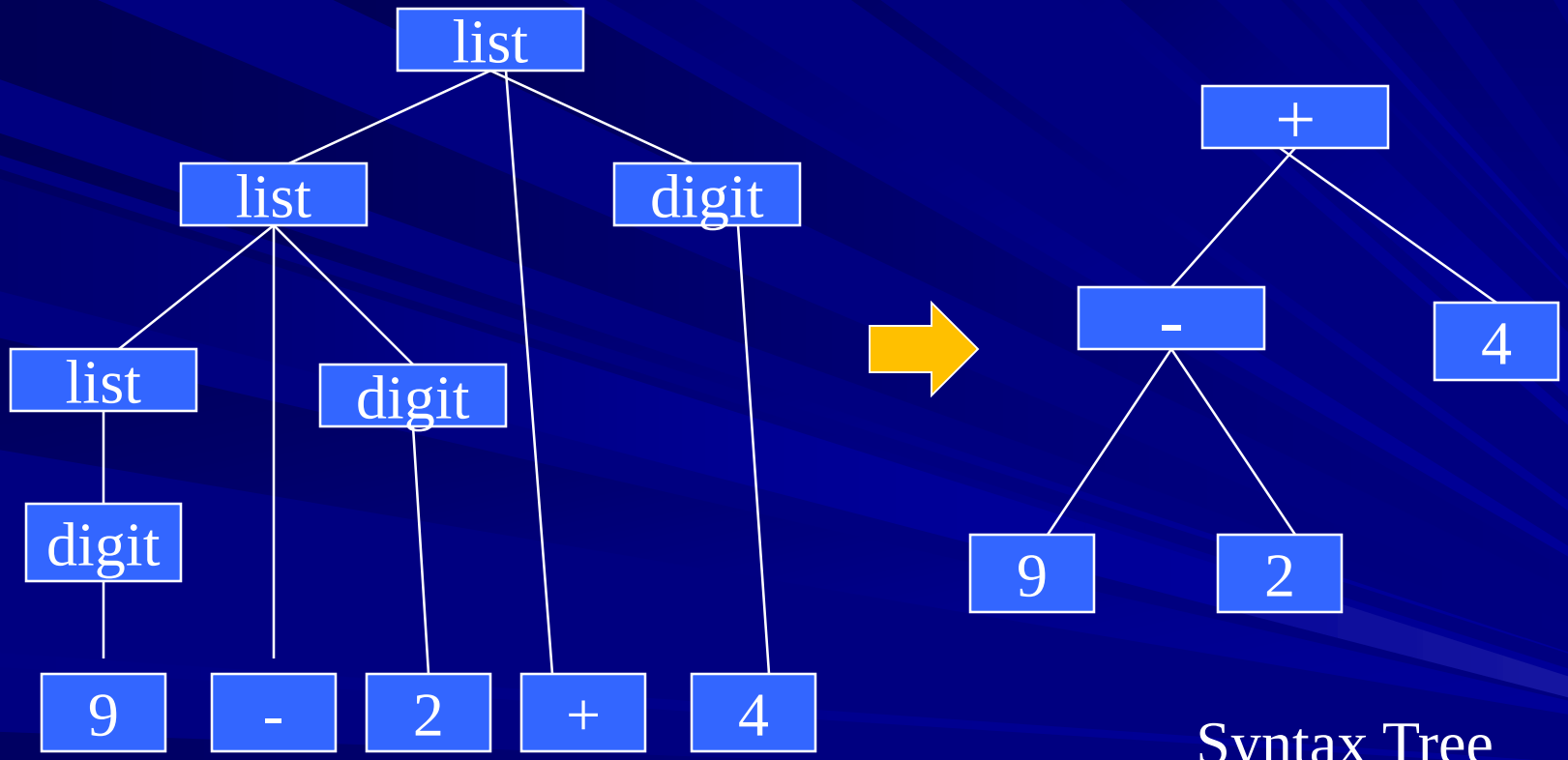
But what if there are many levels of precedences?

Operator Precedence parsing !!
table of operator priority + a stack …

Precedence of Operators in C

Operators	Associativity
() [] - > .	Left to Right
! ~ ++ -- + - * & (type) sizeof *(unary operators)*	Right to Left
* / %	Left to Right
+ -	Left to Right
<< >>	Left to Right
< <= > >=	Left to Right
== !=	Left to Right
&	Left to Right
^	Left to Right
|	Left to Right
&&	Left to Right
||	Left to Right
?:	Right to Left
= += -= *= /= %= &= ^= |= <<= >>=	Right to Left
,	Left to Right

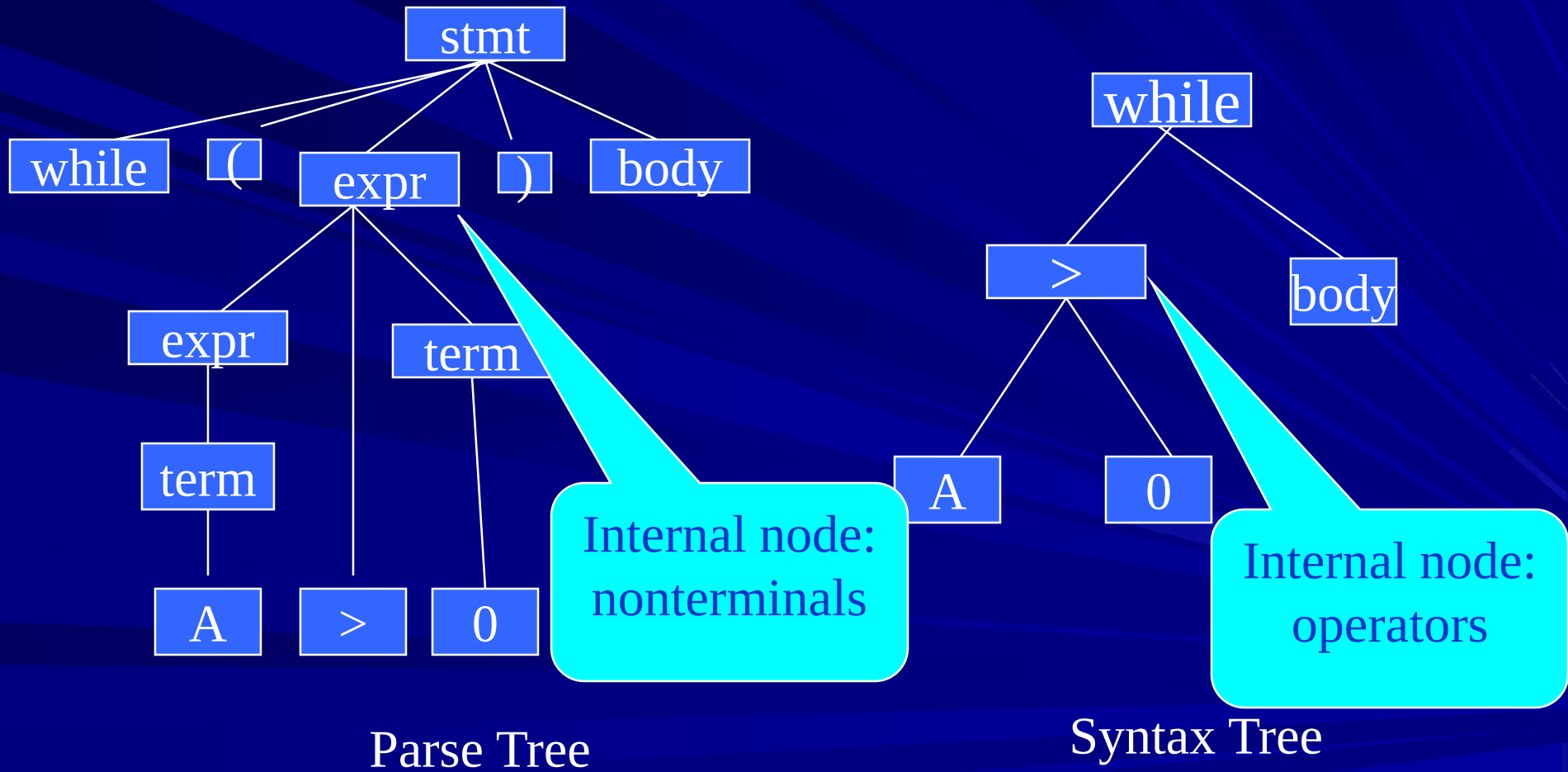
Parse Tree and Syntax Tree



Parse Tree

Syntax Tree
(also called AST)

Parse Tree and Syntax Tree



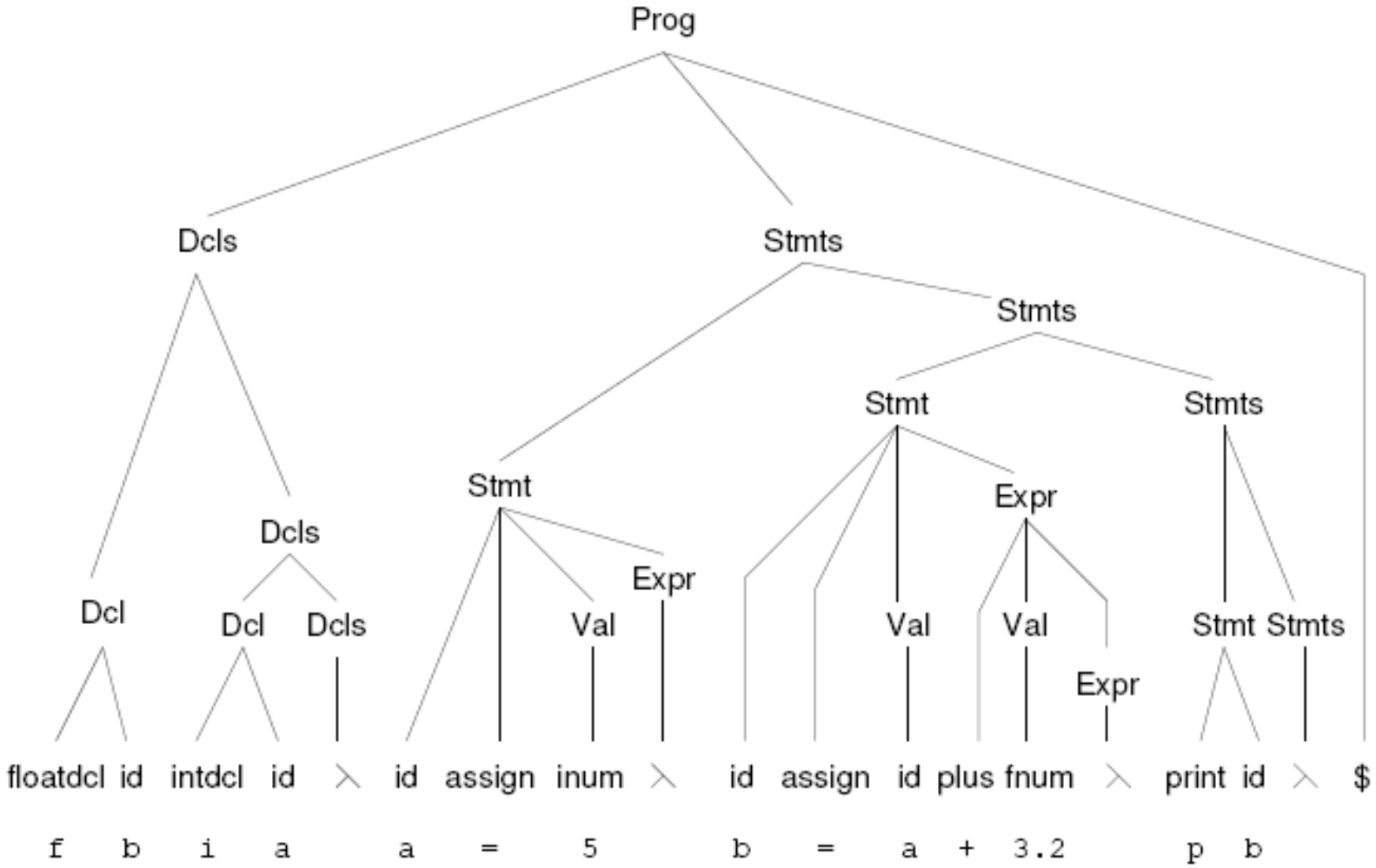


Figure 2.4: An ac program and its parse tree.

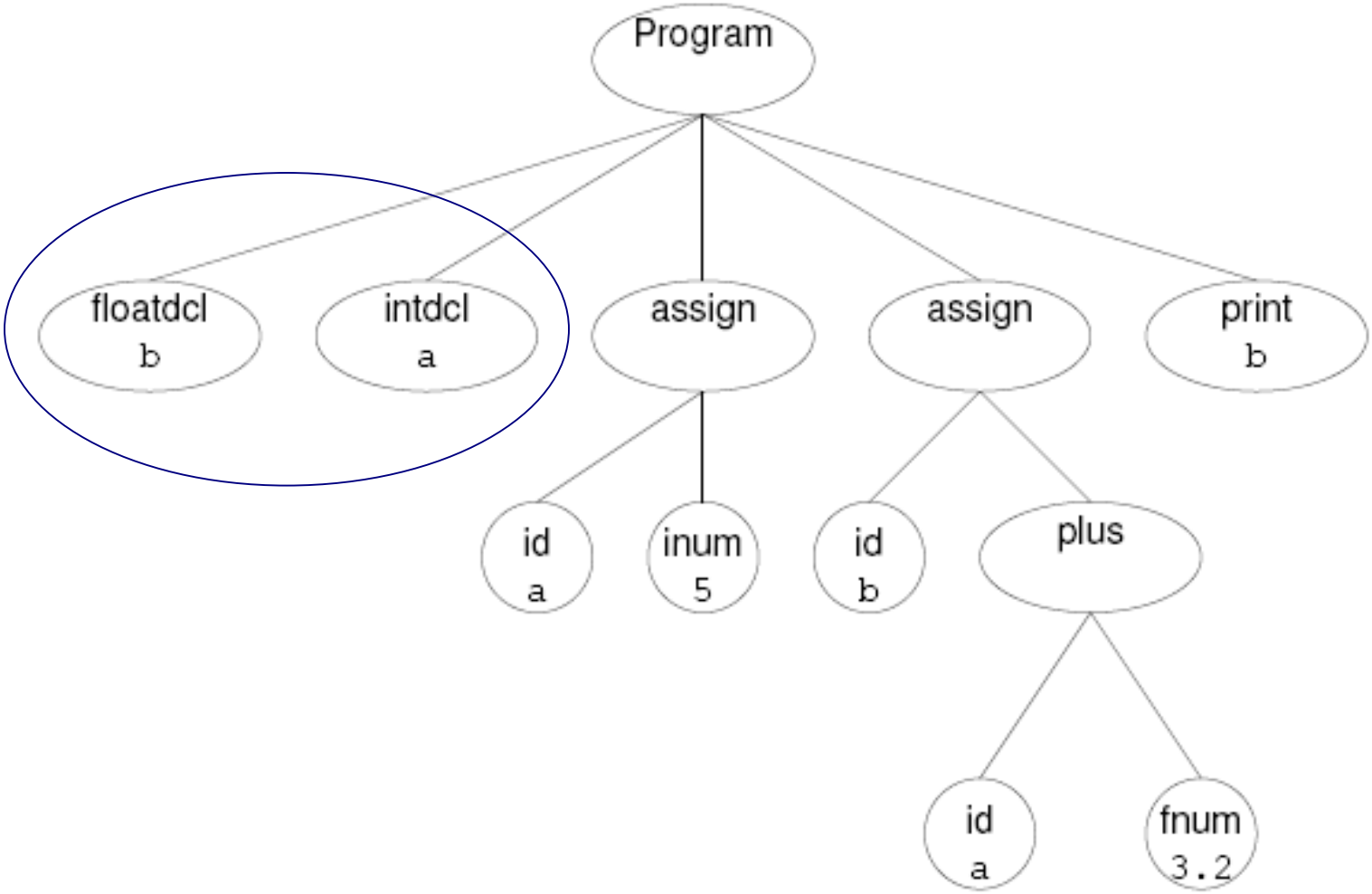


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

Parsing

- Parsing is to determine if a token string can be generated by a CFG
- Two common parsing methods: Top-down and bottom-up to come up with the parse tree.
- Top-down starts at the root and proceeds towards leaves
- Bottom-up starts at the leaves and proceeds towards the root

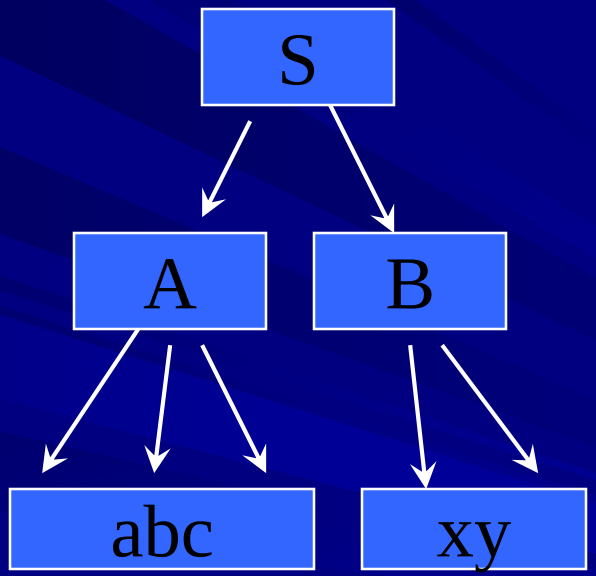
Parsing Example

String: abcxy

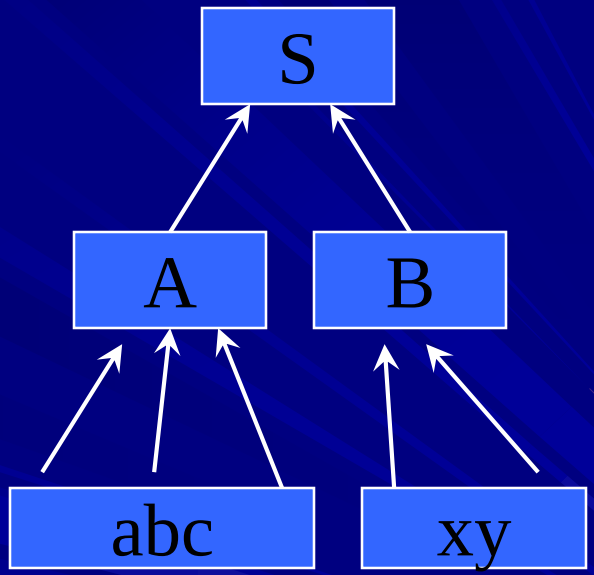
Productions:

S → AB
A → abc | w
B → de | xy

Top-down



Bottom-up



Top-down vs. Bottom-up

Top-down	Bottom-up
Easy to understand	Can handle a larger class of grammars
Efficient parsers can be built by hand	Efficient parsers can be built by tools
Some restrictions on grammars, may need to modify productions	Less restrictions on grammars
Also known as predictive parsing	More commonly used in production compilers

Top Down Parsing

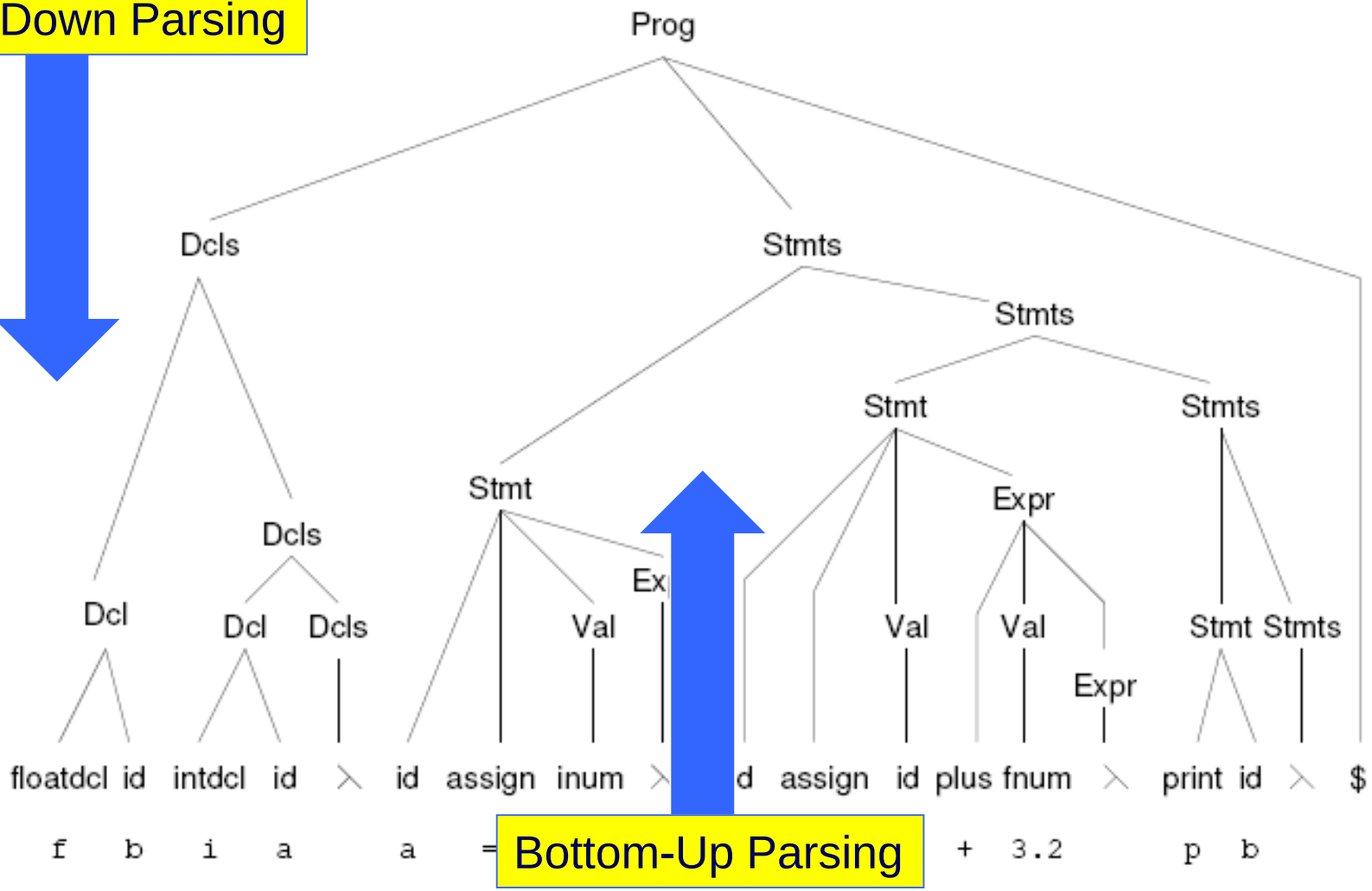


Figure 2.4: An ac program and its parse tree.

Recursive Descent Parsing

- One of the simplest parsing method
- Basic idea
  - Each non-terminal has a parsing procedure
  - For symbols on RHS,
    - to match a non-terminal A, call the parsing procedure A,
    - to match a terminal t, call match(t).
- example

```
Void system_goal (void)
 /* <system_goal> → <program> SCANEOF */
{
    program();
    match (SCANEOF);
}
```

Abstract Example

A → BCD

```
Function A
{
    call B;
    call C;
    call D;
}
```

A → BxCyD

```
Function A
{
    call B;
    match(x);
    call C;
    match(y);
    call D;
}
```

B,C,D are non-terminals
x,y are tokens

Abstract Example

A → BCD | EF

```
Function A
{
    if (lookahead == First(BCD))
        {call B; call C; call D;}
    elseif (lookahead == First (EF))
        {call E; call F;}
}
```

First() is a set of terminal symbols (tokens) that can begin a string derivable from a string of grammar symbols

Parsing Statement

Stmt → ID = Val Expr | Print ID

```
Stmt() {
   token t=next_token();  /* does not consume the token */
   if (t == ID) { match(ID);
                  match(ASSIGN);
                  Val();
                  Expr();}
   else if (t==Print) { match(Print); match(ID);} }
```

the next_token is used to determine what production to apply

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

Constructing a Predictive Parser

- Create a procedure for each non-terminal
- It decides which production to use by looking at the lookahead symbol.
  - The lookahead token can uniquely determine which production to apply.
- The procedure mimicking the right hand side: non-terminal will be a call, and a token match with the lookahead will cause the next token to be read.
- Action routines can be copied into the parser.

```
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.

Stmts → Stmt Stmts
| λ

Predictive Parsing

- In predictive parsing, the lookahead symbol can uniquely select the procedure for each non-terminal.
- The FIRST(α) is defined to be the set of tokens that appear as the first symbols that can be generated from α.

  example:

  FIRST(*Stmt*) = {ID, **print**}
- λ-production is used as default when no other productions can be used.

Predictive Parsing (cont.)

- Not all CFG have the property of predictive parsing -- productions sharing the same LHS can be distinguished by the FIRST() set.
- CFG with the above property is called LL(1) grammar.
- LL(1) stands for “Left-to-Right, Left-most derivation”. (1) means only one lookahead token is required in predictive parsing.
- The *AC* CFG is LL(1).

Left Recursion Removal

- Left recursion creates troubles for recursive descent parsing
- Example of left recursion

  expr → expr + term

  A → Aα | β { β, βα, βαα, βααα, …..}

To eliminate left recursion, we can rewrite the productions.

A → β R

R → α R | λ { β, βα, βαα, βααα, …..}

Exercise

CFG

1. Expr → Expr + term
2. Expr → Expr – term
3. Expr → term

What is α ?

α is + term and – term

What is β ?

β is term

After rewriting:

Expr → term Rest

Rest → + term Rest

Rest → - term Rest

Rest → λ

Another Exercise

ID_List → ID_List , ID | ID

What is α ?

After rewriting

α is , ID

What is β ?

ID_List → ID Rest

Rest → , ID Rest | λ

β is ID

Quick Review

- Parsing
  - What is parsing
  - Two common methods of parsing
  - Recursive descent parsing
  - Predictive parsing
  - Challenges to predicative parsing
  - How to eliminate left recursions

A process of finding a parse tree

Top-down & Bottom up

1st lookahead can uniquely select a production (parsing procedure).

Left recursion & Left factoring

$A \rightarrow A\alpha \mid \beta$ $\longrightarrow$ $A \rightarrow \beta R$, $R \rightarrow \alpha R \mid \lambda$

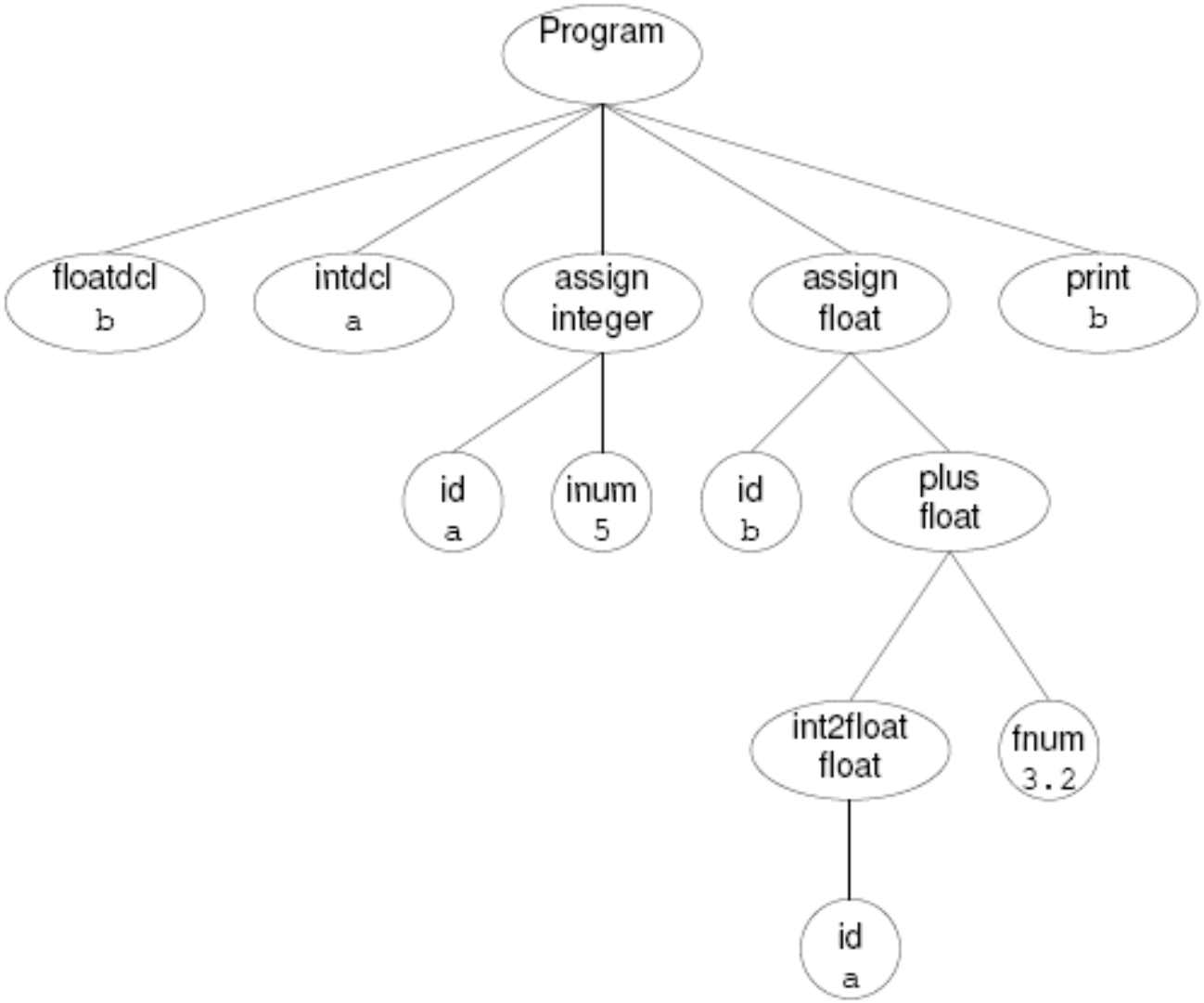


Figure 2.13: AST after semantic analysis.

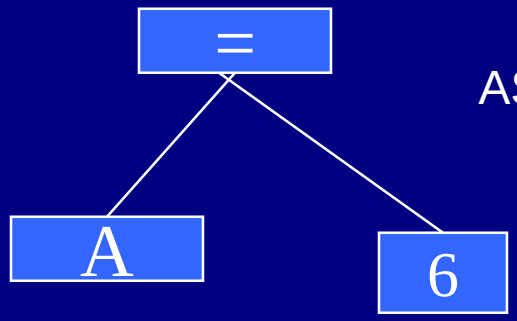
Building AST

- AST is built during parsing
- Example:

```
ParseStatement()
{
    switch (source.peek().type) {
    case: Token.identifier:
        Token identifier = source.nextToken();
        Token assign = source.peek();
        if (source.peek().type() == Token.AssignmentOp) {
            source.nextToken();
            AST v=parseValue();
            return AST.makeTree(assign, identifier, v,
                                        parseExpressionTail(v));
```

Building AST

- Example A = 6

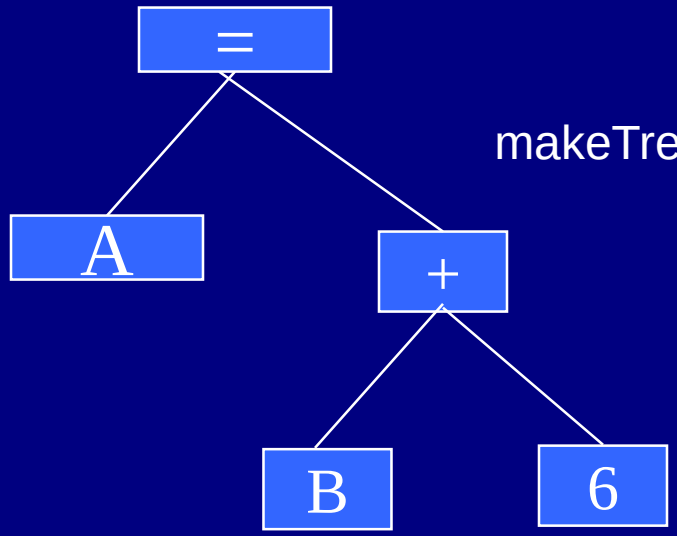


AST.makeTree(assign, identifier, v,
parseExpressionTail(v));

↓

λ

- Example A = B + 6



makeTree(=, A, B, ParseExpTail)

↓

(+, 6)

Semantic Analysis

- Post-parsing processing to enforce aspects of a language's definition that are not easily checked by syntax analysis.
  - Symbol table
  - Type checking

    Types are examined for consistency

    Type dependent behavior becomes explicit

```
/*    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		

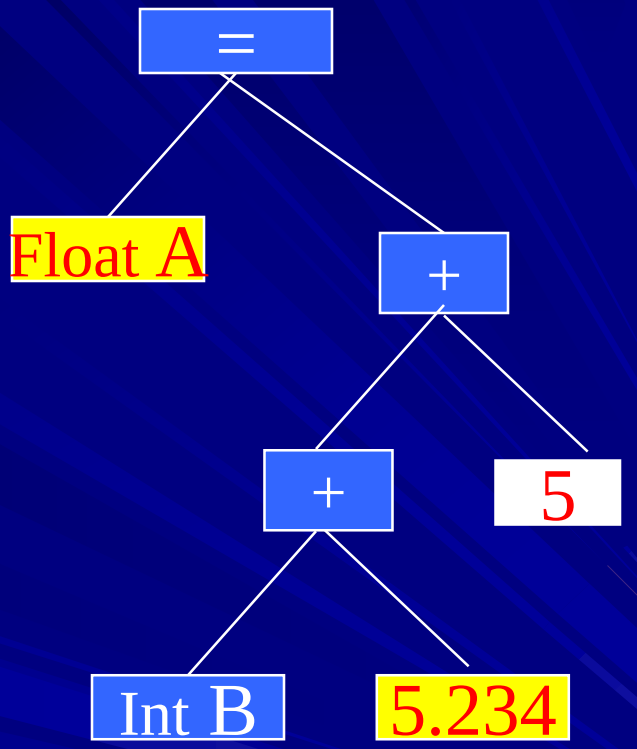
Figure 2.11: Symbol table for the ac program from Figure 2.4.

Type Checking in AC

- All identifiers must be declared. After symbol table construction, executable statements can be checked for type consistency.
- Automatic widening (e.g. from **integer** to **float**) is allowed in AC.
- Narrowing is considered illegal in AC.
- This process walks the AST bottom-up, from its leaves toward its root. At each node, the appropriate visitor method is applied.

```
/*    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
```

Figure 2.12: Type analysis for ac.



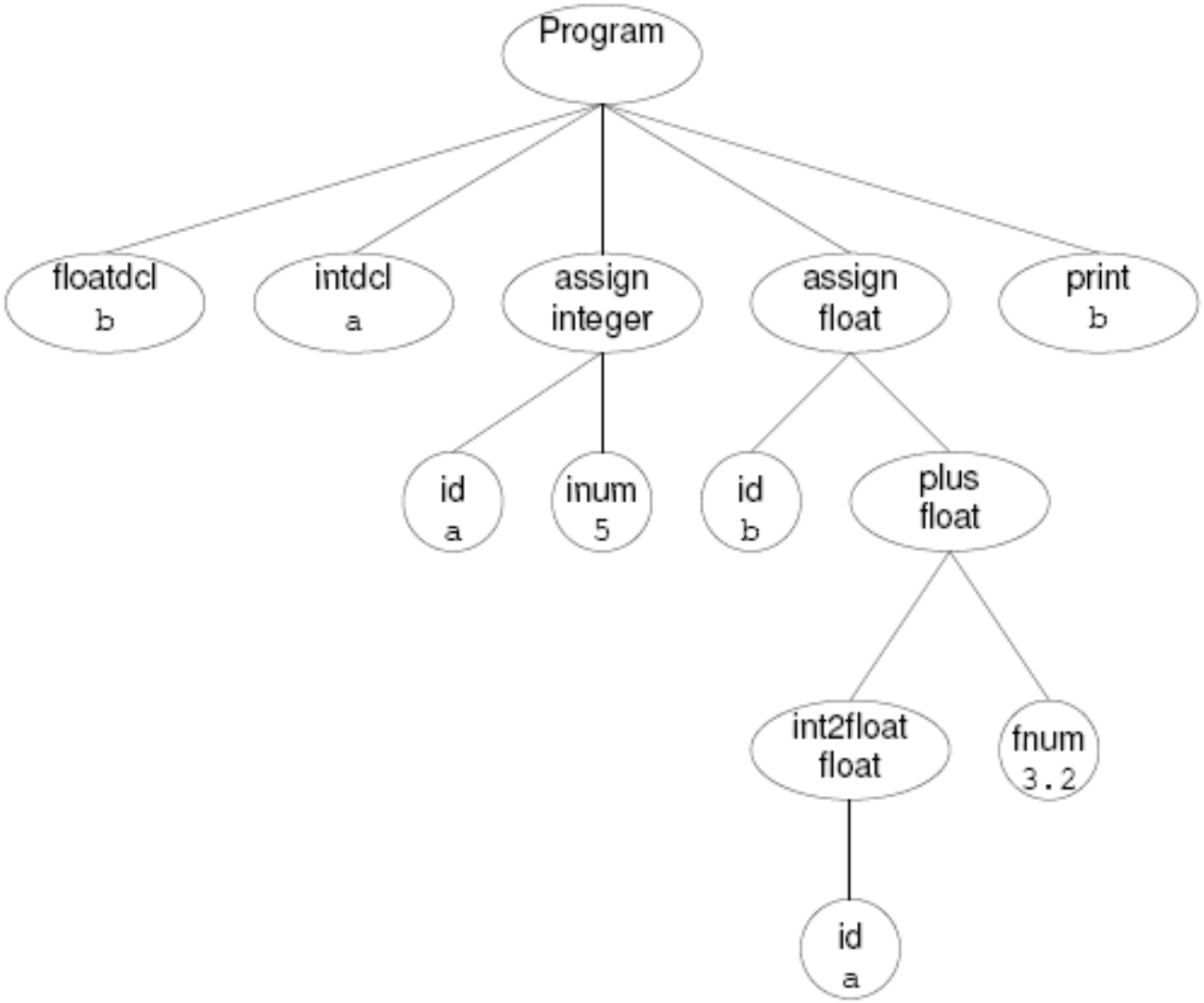


Figure 2.13: AST after semantic analysis.

Translating AC

- Target language: DC
- Note that we do not worry about registers at this time.
  - temporaries: To evaluate expressions, we need to hold temporary values.
    - e.g. A+B+C

      ADD A,B,TEMP&1

      ADD TEMP&1,C,TEMP&2
  - In real compilers, registers are typically used as temporaries.
  - For our target code DC, we use the stack for temporaries.

About Assignment #1

- Loading/Storing variables
  - DC allows single character named registers for storing variables and for later retrieval.

    e.g.

    sc

    means popping the top of the stack and stores it in register c

    lc

    means pushing the value of register c onto the stack

  To simplify our first assignment, we limit the number of variables declared/used to be <= 23.

Some Questions

What are differences among Top-down, recursive descent, and predictive parsing?

Top-down parsing: build the parse tree top-down, left-to-right.

Recursive descent: one way to implement top-down parsing, may need to *backtrack*

Predictive parsing: recursive descent parsing when the CFG is LL(k), no *backtracking*.

Example

What parsing method is used by the AC compiler?

a) Top-down parsing
b) Recursive descent parsing
c) Predictive parsing
d) Bottom-up parsing

a,b,c

About AST building

CFG with left recursion

Expr → Expr + Term

maketree("+", Expr(), Term());

AC CFG with right recursion

Stmt → id assign Val Expr

Expr → + Val Expr

| - Val Expr

maketree(op,*previous-subtree*,Val);

Code Generation for AC

- Code generation proceeds by traversing the AST, starting at its root and walking toward its leaves.
  - Computing node: + and –

    code generator is called recursively to generate code for the left and right subtrees.
  - Assigning node

    Evaluate the expression first, then the result is stored to a DC register. Also reset the precision to integer initially.
  - SymReferencing node, Printing node, Converting node

Code Generation for AC

```
Procedure visit(Computing n)
    call CodeGen(n.child1);
    call CodeGen(n.child2);
    call Emit(n.operation);
End
Procedure (SymReferencing n)
    call Emit("l");  /* load a variable */
    call Emit(n.id);   /* In our assignment, this n.id
should be the register the n.id has mapped to */
End
```

Syntax Directed Translation

- Syntax-directed definition
  - CFG + semantic rules
  - Translation of a construct is specified by
    - **attributes** associated with the grammar symbol
    - **semantic rules** associated with each production
  - Semantic rules can be actual code known as semantic routines (or action routines)
  - Attributes can be type, name, memory location, structure, .. etc
  - Most of the translation is done by semantic routines

e.g. Exp1→Exp2+Term (Exp2.type == numeric && Term.type==numeric).

attributes

Semantic rules

Single Pass vs. Multi-Pass Compilation

- For our term project, semantic check and code generation can be accomplished together. The compilation process only goes through the code once. This is called single pass compilation.
- Production compilers usually adopt multiple pass compilation. They usually generate AST (Abstract Syntax Tree) as IR, and conduct semantic check, code generation, and optimization in multiple passes. Each phase is implemented as a separate pass.

Semantic Routines

- Action Symbols
  - The bulk of a translation is done by semantic routines
  - Action symbols can be added to a grammar to specify when semantic processing should take place
    - Placed anywhere in the RHS of a production
  - translated into procedure call in the parsing procedures
    - #add corresponds to a semantic routine named add()
  - No impact on the languages recognized by a parser driven by a CFG

Semantic Records

- In the code example, the semantic record for <expr> needs to specify information like type, what location is the expression saved (e.g. a temporary name), a constant value, …
- Semantic record for a non-… created by a semantic routine after using the information of the symbols on RHS.
- Example

  <expr> → <expr1> + <primary> #gen_infix

  The gen-infix routine uses the information in the two semantic records for <expr1> and <primary> to generate a new record for <expr>.

Such as the constant folding optimization in your assignment

Semantic Record (cont.)

- When the semantic routine gen_infix is called, it must be given the semantic records of the two symbols <expre1> and <primary>.
- Semantic records of non-terminals can be returned by the parsing routine (either as a result parameter or as a function return value)
- Variant record (and nested variant record)

Information stored for various symbols are often different. So variant records (i.e. **union** in C) is often used to define one structure for all semantic records (especially true for using a parser generator)

Semantic Record for Expr

```
typedef struct operator {
    enum op {PLUS, MINUS} operator;
} op_rec;
enum expr {IDexpr, LITexpr, TEMPexpr}
typedef struct expression {
    enum expr kind;
    union {
        string name;
        int value; }
    } expr_rec;
```

Combined Variant Record

```
enum sem_type {op_type, expr_type}
typedef struct semantic_record {
    enum sem_type tag;
    union {
        op_rec          *op;
        expr_rec        *expr;     } u;
  } semantic_rec;
```

Combined Variant Record

Example:

```
union {
        char *lexeme;
        int intval;
        float   floatval;
        decl   *dcl;
        id_list*idl;
        par_list        *pal;
        ArrayInfo     *arrinfo;
        …}
```

Summary

- A syntax-directed translator for simple language AC
- Syntax definition – using CFG
- Predictive parsing – recursive descent parsing with unique FIRST() set
- Left recursion elimination
- Single pass translation: action symbols and semantic records
- Multi-pass translation: AST generation and tree traversals for type checking and code generation.