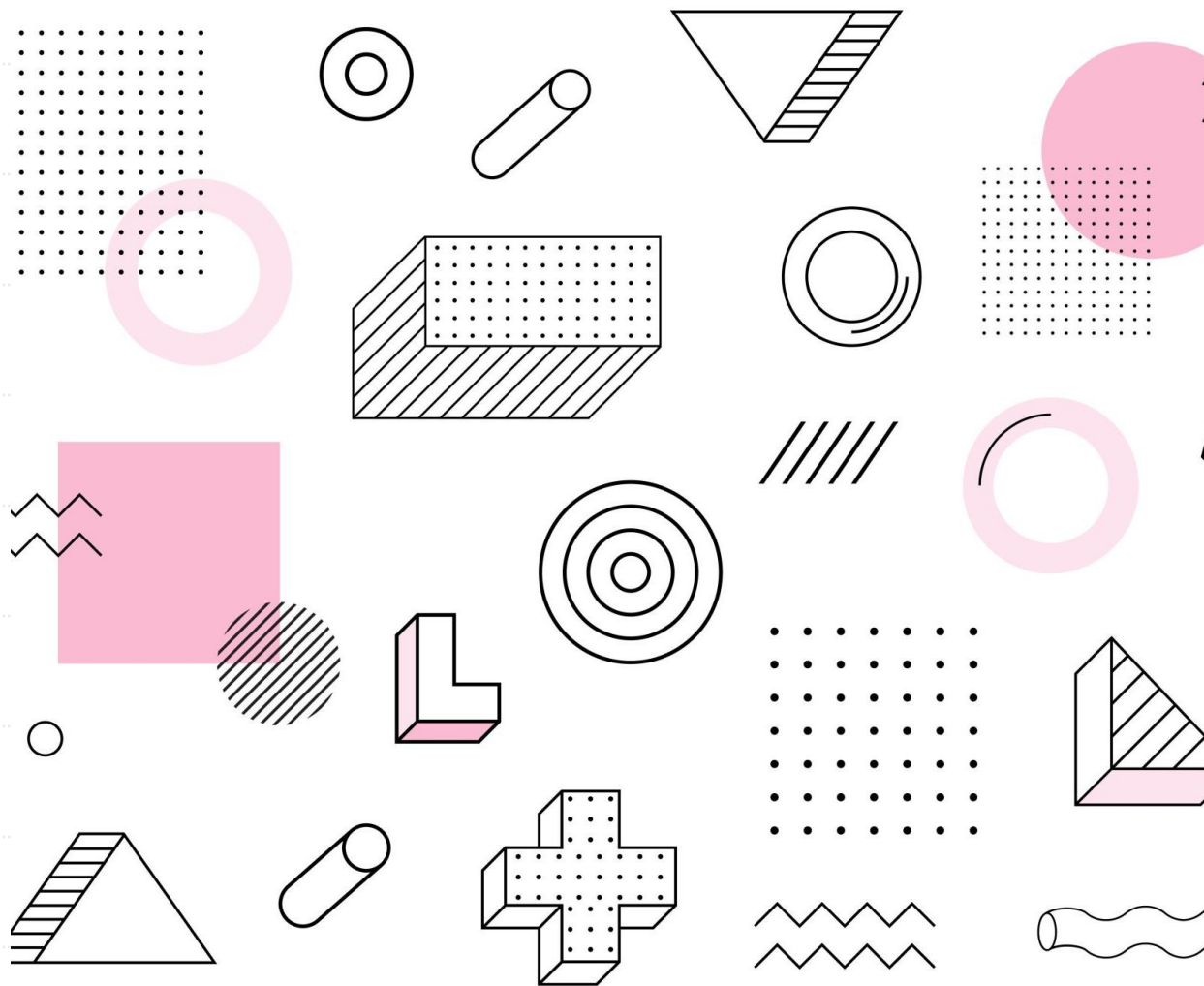


Chapter 2: A Simple Compiler

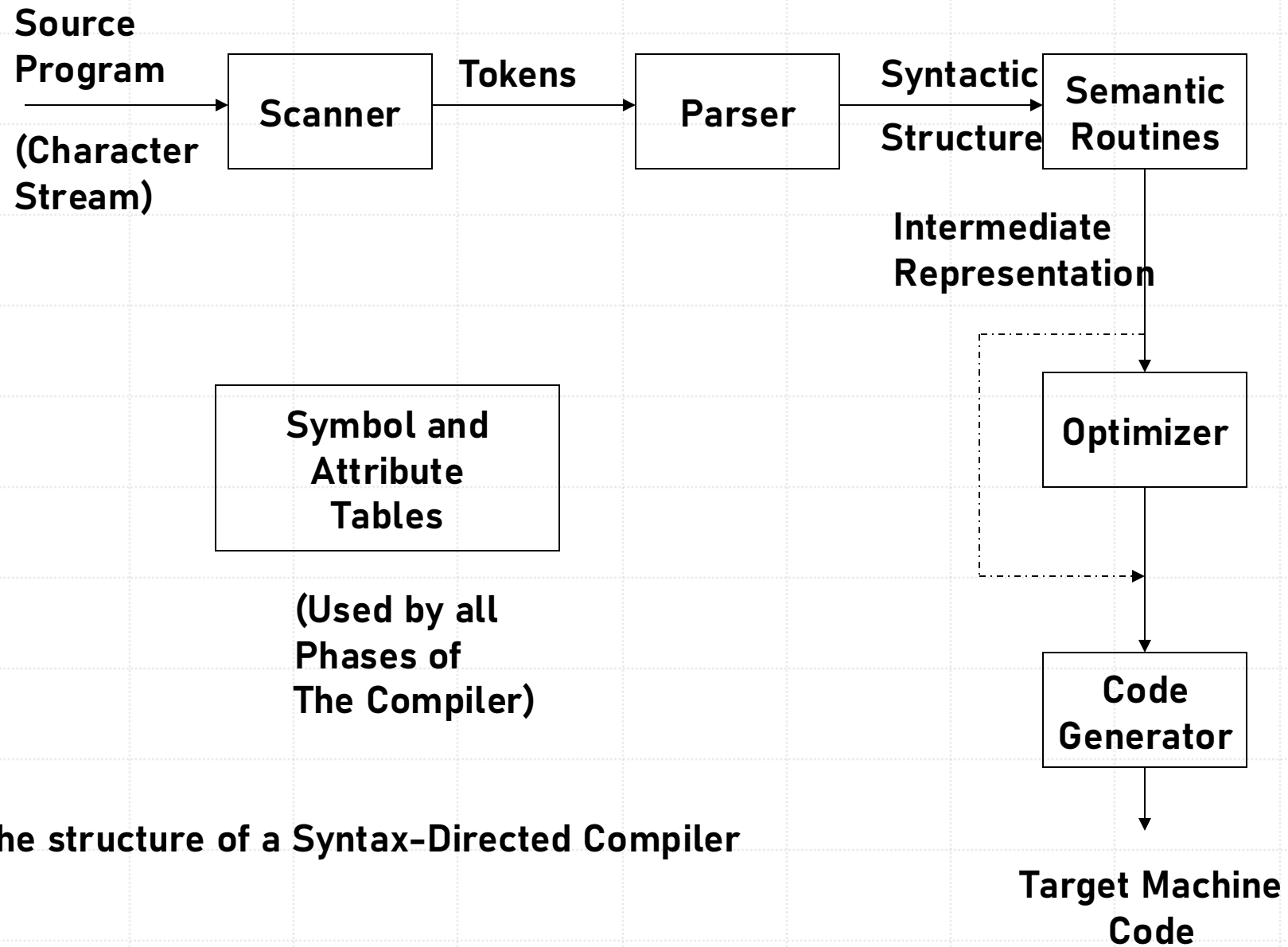
陳奇業 成功大學資訊工程系





Outlines

- 2.1 An Informal Definition of the ac Language
- 2.2 Formal Definition of ac
- 2.3 Phases of a Simple Compiler
- 2.4 Scanning
- 2.5 Parsing
- 2.6 Abstract Syntax Trees
- 2.7 Semantic Analysis
- 2.8 Code Generation



The structure of a Syntax-Directed Compiler

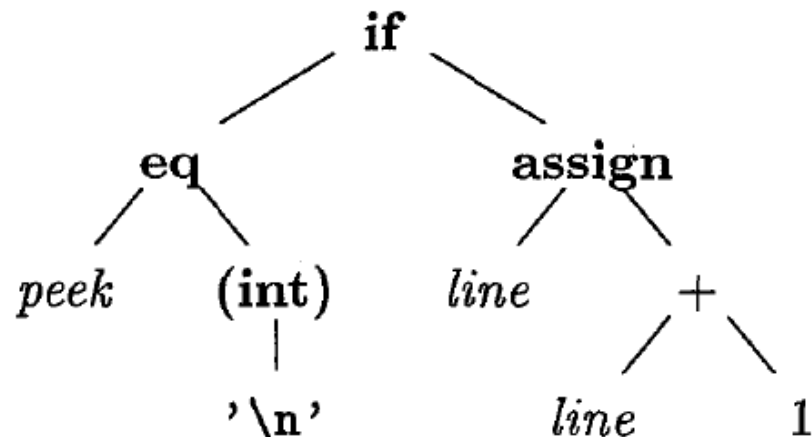
```
if( peek == '\n' ) line = line + 1;
```

Lexical Analyzer

```
<if> <(> <id, "peek"> <eq> <const, '\n'> <)>  
<id, "line"> <assign> <id, "line"> <+> <num, 1> <;>
```

Syntax-Directed Translator

or



```
1: t1 = (int) '\n'  
2: ifFalse peek == t1 goto 4  
3: line = line + 1  
4:
```



An Informal Definition of the ac Language

- **Types:** There are only two data types: **integer** and **float**. An integer type is a sequence of decimal numerals, as found in most programming languages. A float type allows five fractional digits after the decimal point.
- **Keywords:** There are **three reserved keywords**, each limited for simplicity to a **single letter**: **f** (declares a float variable), **i** (declares an integer variable), and **p** (prints the value of a variable).
- **Variables:** The ac language offers only **23 possible variable names**, drawn from the lowercase Roman alphabet and excluding the three reserved keywords f, i, and p. Variables must be declared prior to using them.



An Informal Definition of the ac Language

- In some cases, such type conversion is handled automatically by the compiler, while other cases require explicit syntax (such as casts) to allow the type conversion.
- In ac, conversion from integer type to float type is accomplished automatically. Conversion in the other direction is not allowed under any circumstances.



An Informal Definition of the ac Language

- For the target of translation, we use the widely available program dc (for desk calculator), which is a stack-based calculator that uses **reverse Polish notation** (RPN 逆波蘭表示法).
- When an ac program is translated into a dc program, the resulting instructions must be acceptable to the dc program and must faithfully represent the operations specified in an ac program.



Formal Definition of ac

- Before translating ac to dc we must first understand the **syntax** and **semantics** of the ac language.
- We use **a context-free grammar (CFG)** to specify our language's syntax and **regular expressions** to specify the basic symbols of the language.

The Syntax of ac

- Ac's syntax is defined by a context-free grammar (CFG)
- CFG is also called BNF (Backus-Naur Form 巴科斯範式) grammar
- CFG consists of a set of production rules,

A → B C D ... Z

LHS

RHS

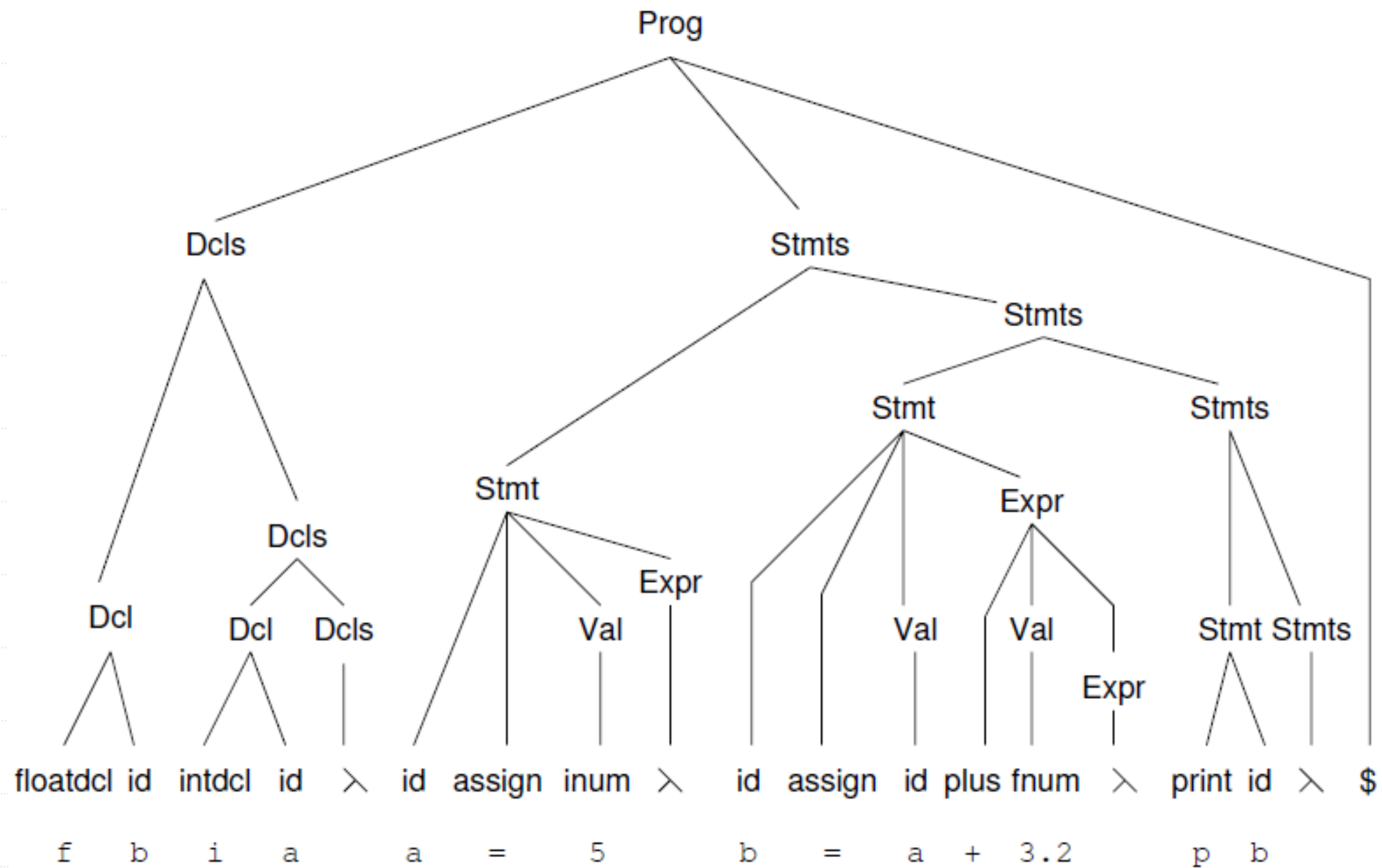
LHS must be a single **nonterminal**

RHS consists 0 or more **terminals** or **nonterminals**

Syntax Specification

```
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
```

Step	Sentential Form	Production Number	
1	$\langle \text{Prog} \rangle$		1 $\text{Prog} \rightarrow \text{Dcls Stmts } \$$
2	$\langle \text{Dcls} \rangle \text{ Stmts } \$$	1	2 $\text{Dcls} \rightarrow \text{Dcl Dcls}$
3	$\langle \text{Dcl} \rangle \text{ Dcls Stmts } \$$	2	3 $\quad \quad \quad \lambda$
4	$\text{floatdcl id } \langle \text{Dcls} \rangle \text{ Stmts } \$$	4	4 $\text{Dcl} \rightarrow \text{floatdcl id}$
5	$\text{floatdcl id } \langle \text{Dcl} \rangle \text{ Dcls Stmts } \$$	2	5 $\quad \quad \quad \text{intdcl id}$
6	$\text{floatdcl id } \text{intdcl id } \langle \text{Dcls} \rangle \text{ Stmts } \$$	5	6 $\text{Stmts} \rightarrow \text{Stmt Stmts}$
7	$\text{floatdcl id } \text{intdcl id } \langle \text{Stmts} \rangle \$$	3	7 $\quad \quad \quad \lambda$
8	$\text{floatdcl id } \text{intdcl id } \langle \text{Stmt} \rangle \text{ Stmts } \$$	6	8 $\text{Stmt} \rightarrow \text{id assign Val Expr}$
9	$\text{floatdcl id } \text{intdcl id } \text{id assign } \langle \text{Val} \rangle \text{ Expr Stmts } \$$	8	9 $\quad \quad \quad \text{print id}$
10	$\text{floatdcl id } \text{intdcl id } \text{id assign inum } \langle \text{Expr} \rangle \text{ Stmts } \$$	14	10 $\text{Expr} \rightarrow \text{plus Val Expr}$
11	$\text{floatdcl id } \text{intdcl id } \text{id assign inum } \langle \text{Stmts} \rangle \$$	12	11 $\quad \quad \quad \text{minus Val Expr}$
12	$\text{floatdcl id } \text{intdcl id } \text{id assign inum } \langle \text{Stmt} \rangle \text{ Stmts } \$$	6	12 $\quad \quad \quad \lambda$
13	$\text{floatdcl id } \text{intdcl id } \text{id assign inum id assign } \langle \text{Val} \rangle \text{ Expr Stmts } \$$	8	13 $\text{Val} \rightarrow \text{id}$
14	$\text{floatdcl id } \text{intdcl id } \text{id assign inum id assign id } \langle \text{Expr} \rangle \text{ Stmts } \$$	13	14 $\quad \quad \quad \text{inum}$
15	$\text{floatdcl id } \text{intdcl id } \text{id assign inum id assign id plus } \langle \text{Val} \rangle \text{ Expr Stmts } \$$	10	15 $\quad \quad \quad \text{fnum}$
16	$\text{floatdcl id } \text{intdcl id } \text{id assign inum id assign id plus fnum } \langle \text{Expr} \rangle \text{ Stmts } \$$	15	
17	$\text{floatdcl id } \text{intdcl id } \text{id assign inum id assign id plus fnum } \langle \text{Stmts} \rangle \$$	12	
18	$\text{floatdcl id } \text{intdcl id } \text{id assign inum id assign id plus fnum } \langle \text{Stmt} \rangle \text{ Stmts } \$$	6	
19	$\text{floatdcl id } \text{intdcl id } \text{id assign inum id assign id plus fnum print id } \langle \text{Stmts} \rangle \$$	9	
20	$\text{floatdcl id } \text{intdcl id } \text{id assign inum id assign id plus fnum print id } \$$	7	





Token Specification

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	(" ") ⁺



An ac Scanner

- The ac Scanner will be a function of no arguments that returns token values
- There are 10 tokens.

```
typedef enum token_types {  
    floatdcl, intdcl, print, id, assign, plus,  
    minus, inum, fnum, blank  
} token;
```

```
Extern token scanner(void);
```



An ac Scanner (Cont'd)

- The scanner returns the longest string that constitutes a token, e.g., in

abcdef

ab, abc, abcdef are all valid tokens.

The scanner will return the
longest one (i.e., abcdef).



Phases of a Simple Compiler

1. The **scanner** reads a source ac program as a text file and produces a stream of tokens.
2. The **parser** processes tokens produced by the scanner, determines the syntactic validity of the token stream, and creates an abstract syntax tree (AST) suitable for the compiler's subsequent activities.
3. The AST created by the parsing task is next traversed to **create a symbol table**. This table associates type and other contextual information with variables used in an ac program.
4. The AST is next traversed to perform **semantic analysis**.
5. Finally, the AST is traversed to generate a translation of the original program.



Scanning

- The scanner's job is to translate a stream of characters into a stream of tokens, where each token represents an instance of some terminal symbol.
- Each token found by the scanner has the following two components:
 1. A token's type explains the token's membership in the terminal alphabet. All instances of a given terminal have the same token type.
 2. A token's semantic value provides additional information about the token.

For terminals such as plus, **no semantic information** is required, because only one token (+) can correspond to that terminal. Other terminals, such as **id** and **num**, require semantic information so that the compiler can **record which identifier or number** has been scanned.




Scanning

- For most programming languages, the scanner's job is not so easy. Some tokens (+) can be prefixes of other tokens (++); other tokens such as comments and string constants have special symbols involved in their recognition.

Scanner for the ac language.

```
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
```



Finding inum
or fnum tokens
for the ac
language

```
function SCANDIGITS( ) returns token
  tok.val  $\leftarrow$  " "
  while S.PEEK( )  $\in \{0, 1, \dots, 9\}$  do
    tok.val  $\leftarrow$  tok.val + S.ADVANCE( )
  if S.PEEK( )  $\neq$  "."
  then tok.type  $\leftarrow$  inum
  else
    tok.type  $\leftarrow$  fnum
    tok.val  $\leftarrow$  tok.val + S.ADVANCE( )
    while S.PEEK( )  $\in \{0, 1, \dots, 9\}$  do
      tok.val  $\leftarrow$  tok.val + S.ADVANCE( )
  return (tok)
end
```

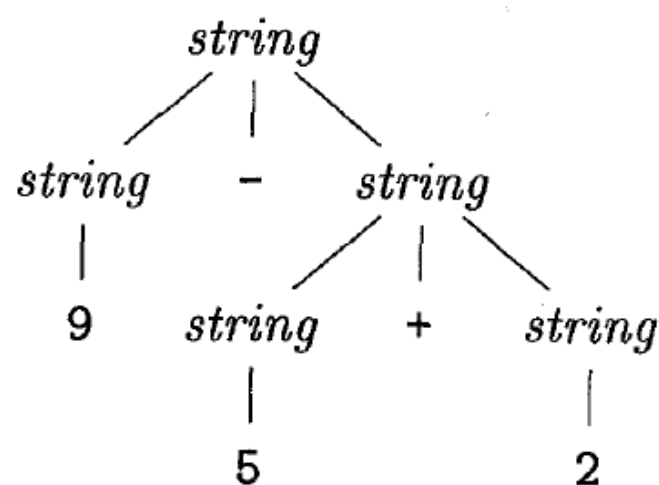
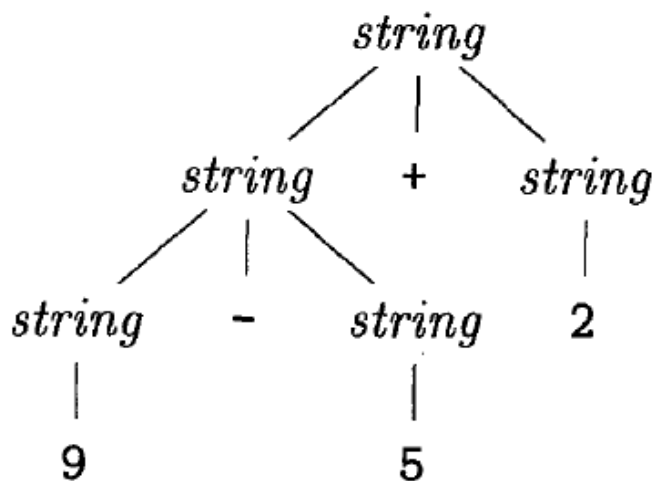


Parsing

- The parser is responsible for determining if the stream of tokens provided by the scanner conforms to the language's grammar specification.
- We build a parser for ac using a well-known parsing technique called **recursive descent**.

Ambiguity (模稜兩可)

- Suppose we used a single nonterminal string and did not distinguish between digits and lists. We could have written the grammar
 $\text{string} \rightarrow \text{string} + \text{string} \mid \text{string} - \text{string} \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$





Associativity of Operators

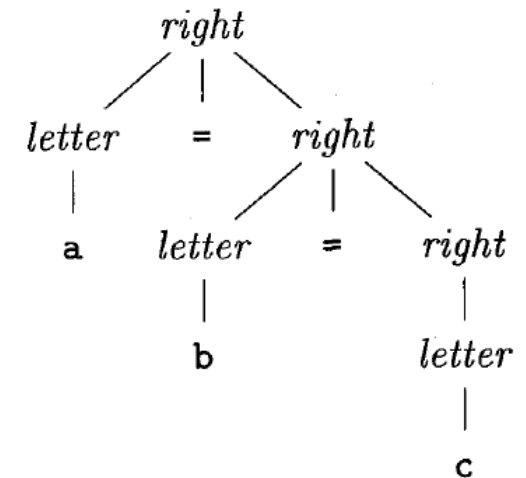
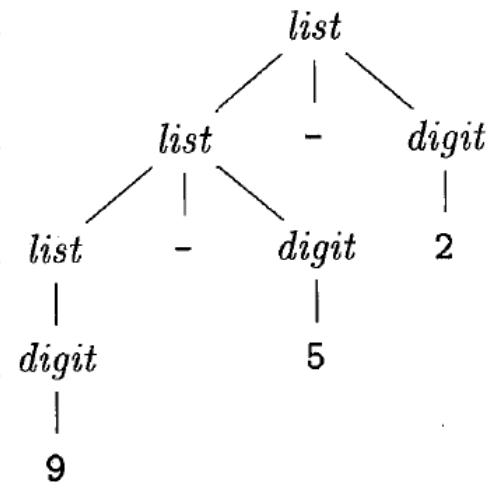
- By convention, $9+5+2$ is equivalent to $(9+5)+2$ and $9-5-2$ is equivalent to $(9-5)-2$. When an operand like 5 has operators to its left and right, conventions are needed for deciding which operator applies to that operand. We say that the operator $+$ associates to the left, because an operand with plus signs on both sides of it belongs to the operator to its left.
- Some common operators such as exponentiation are right-associative. As another example, the assignment operator $=$ in C and its descendants is right associative; that is, the expression $a=b=c$ is treated in the same way as the expression $a=(b=c)$.

Associativity of Operators

- Left recursion => left-associative
- Right recursion => right-associative
- Strings like $a=b=c$ with a right-associative operator are generated by the following grammar:

$\text{right} \rightarrow \text{letter} = \text{right} \mid \text{letter}$

$\text{letter} \rightarrow a \mid b \mid \dots \mid z$





Precedence of Operators

- Consider the expression $9+5*2$. There are two possible interpretations of this expression: $(9+5) * 2$ or $9+ (5*2)$.
- Ex. : A grammar for arithmetic expressions can be constructed from a table showing the associativity and precedence of operators.

$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}$

$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$

$\text{factor} \rightarrow \text{digit} \mid (\text{expr})$

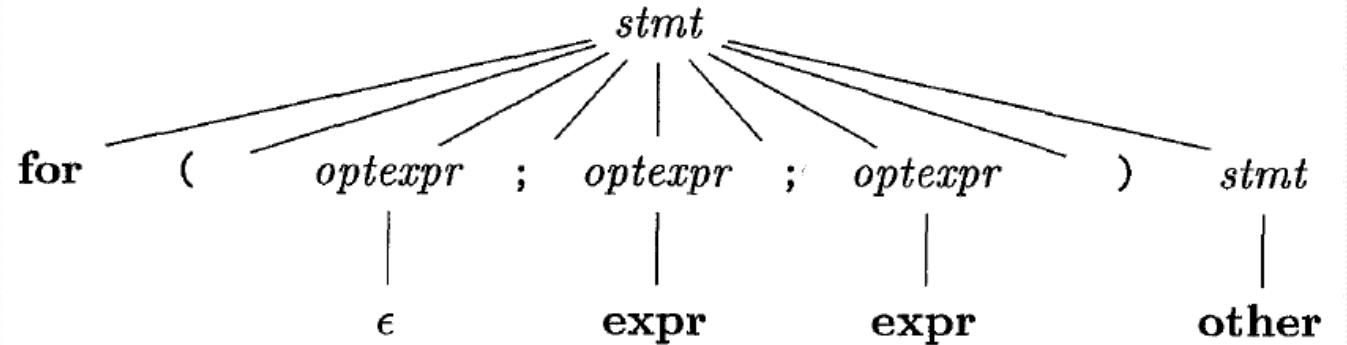


Parse Tree Construction

- Most parsing methods fall into one of two classes, called the **top-down** and **bottom-up** methods.
- In top-down parsers, construction starts at the root and proceeds towards the leaves, while in bottom-up parsers, construction starts at the leaves and proceeds towards the root.

Top-Down Parsing

- $\text{stmt} \rightarrow \text{expr} ;$
| $\text{if} (\text{expr}) \text{stmt}$
| $\text{for} (\text{optexpr} ; \text{optexpr} ; \text{optexpr}) \text{stmt}$
| other
- $\text{optexpr} \rightarrow \epsilon \mid \text{expr}$



Predicting a Parsing Procedure

- Each procedure first examines the next input token to predict which production should be applied. For example, Stmt offers two productions:

Stmt \rightarrow id assign Val Expr

Stmt \rightarrow print id

```
procedure STMT( )  
  if ts.PEEK( ) = id  
  then  
    call MATCH(ts, id)  
    call MATCH(ts, assign)  
    call VAL( )  
    call EXPR( )  
  else  
    if ts.PEEK( ) = print  
    then  
      call MATCH(ts, print)  
      call MATCH(ts, id)  
    else  
      call ERROR( )  
  end
```

Recursive-descent Parsing

- Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input.
- $\text{FIRST}(\text{stmt}) = \{\text{expr}, \text{if}, \text{for}, \text{other}\}$

```
void stmt() {  
    switch ( lookahead ) {  
        case expr:  
            match(expr); match(';'); break;  
        case if:  
            match(if); match('('); match(expr); match(')'); stmt();  
            break;  
        case for:  
            match(for); match('(');  
            optexpr(); match(';'); optexpr(); match(';'); optexpr();  
            match(')'); stmt(); break;  
        case other:  
            match(other); break;  
        default:  
            report("syntax error");  
    }  
}  
  
void optexpr() {  
    if ( lookahead == expr ) match(expr);  
}  
  
void match(terminal t) {  
    if ( lookahead == t ) lookahead = nextTerminal;  
    else report("syntax error");  
}
```



Left Recursion

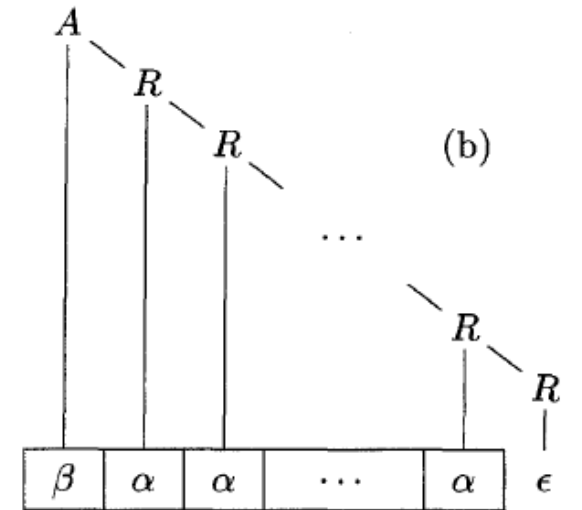
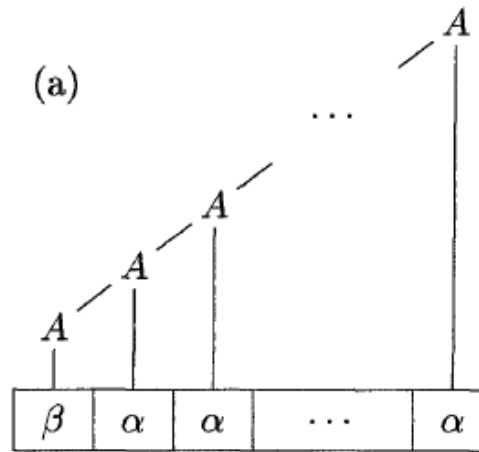
- It is possible for a recursive-descent parser to loop forever. A problem arises with "left-recursive" productions like
 $\text{expr} \rightarrow \text{expr} + \text{term}$
- A left-recursive production can be eliminated by rewriting the offending production. Consider a nonterminal A with two productions
 $A \rightarrow A\alpha | \beta$
- For example, $A = \text{expr}$, $\alpha = + \text{term}$, $\beta = \text{term}$

Left Recursion

- We can convert left recursion to right recursion in the following manner, using a new nonterminal R:

$$A \rightarrow \beta R$$

$$R \rightarrow \alpha R | \epsilon$$





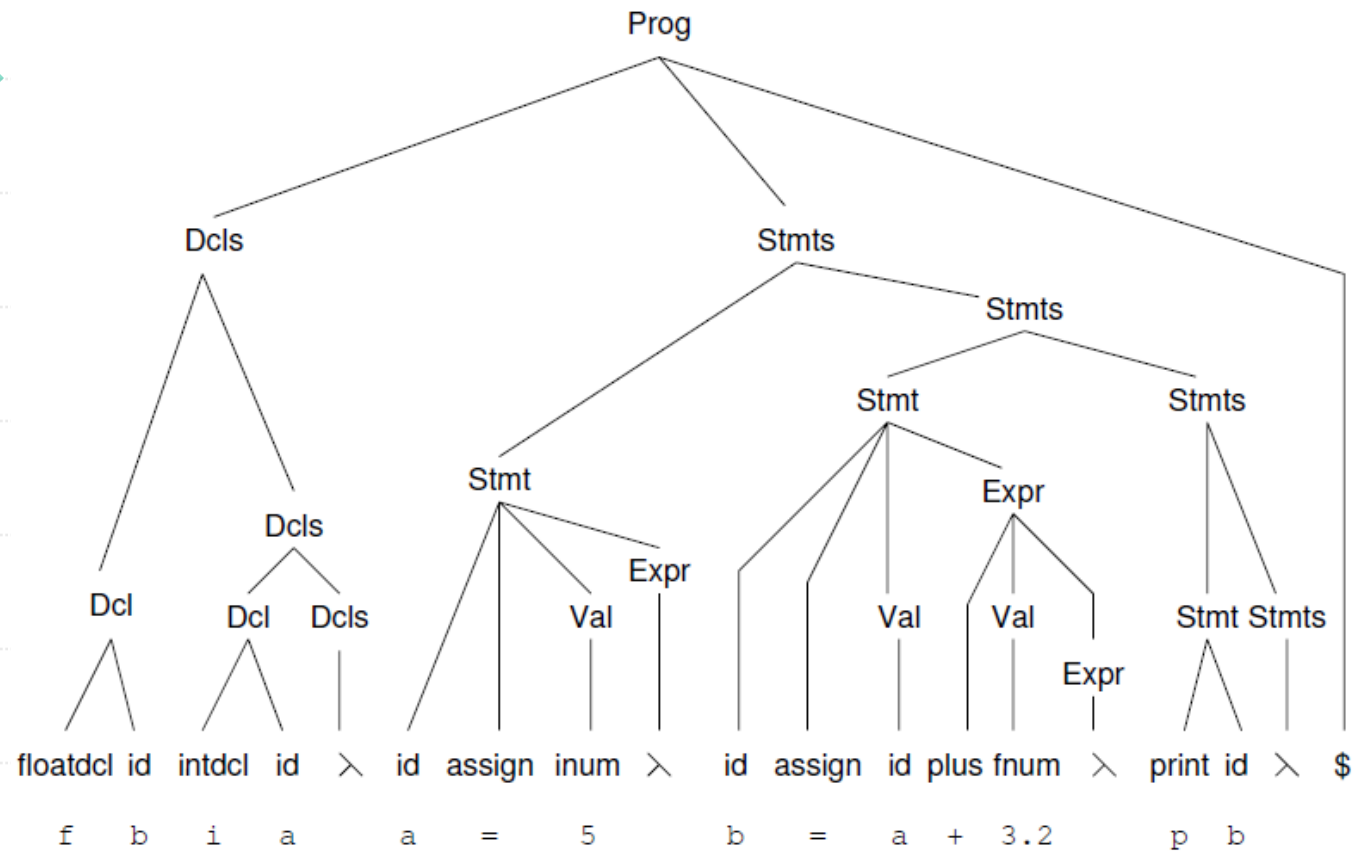
Abstract Syntax Trees

- While the process of compilation begins with scanning and parsing, following are some aspects of compilation that can be difficult or even impossible to perform during syntax analysis:
 - Most programming language specifications include prose that describes aspects of the language that cannot be specified in a CFG. Ex: x.y.z in Java, operator overloading.
 - For relatively simple languages, syntax-directed translation can perform almost all aspects of program translation during syntax analysis. However, from a software engineering perspective, the separation of activities and concerns into phases (such as syntax analysis, semantic analysis, optimization, and code generation) makes the resulting compiler much easier to write and maintain.
- In response to the above concerns, we might consider using the **parse tree** as the structure that survives syntax analysis and is used **for the remaining phases**.

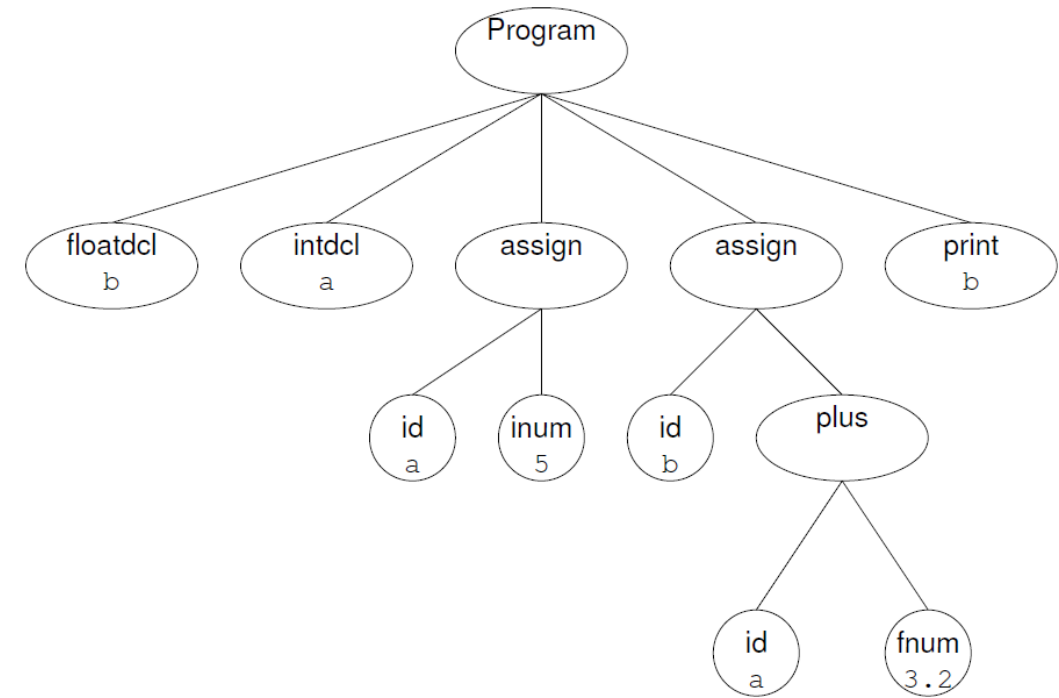


Abstract Syntax Trees

- However, such trees can be rather large and unnecessarily detailed, even for very simple grammars and inputs.
- It is therefore common practice to create an artifact of syntax analysis known as the abstract syntax tree (AST). This structure contains the essential information from a parse tree, but inessential punctuation and delimiters (braces, semicolons, parentheses, etc.) are not included.



Parse Tree



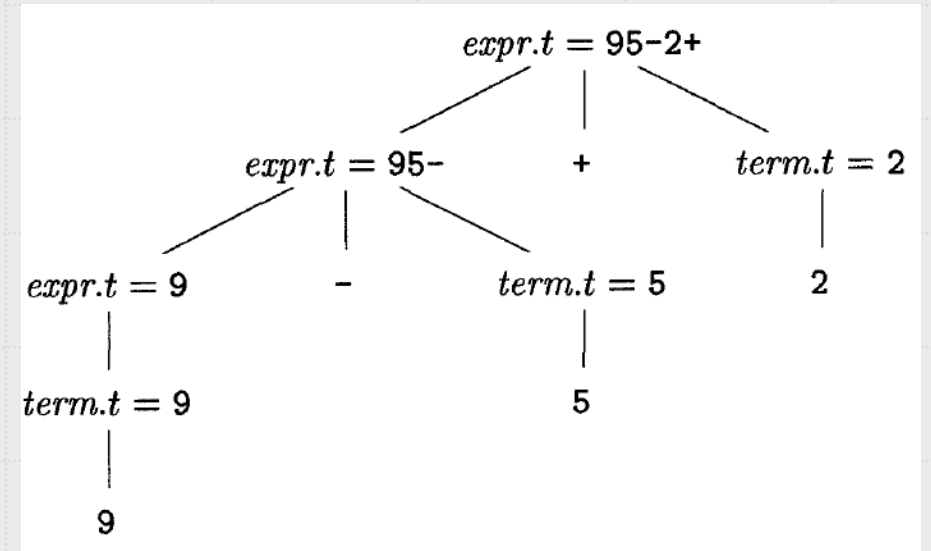
Abstract Syntax Tree



Syntax-Directed Translation

- Two concepts related to syntax-directed translation:
 - **Attributes**. An attribute is any quantity associated with a programming construct. Examples of attributes are data types of expressions, the number of instructions in the generated code, or the location of the first instruction in the generated code for a construct ...
 - **Translation schemes**. A translation scheme is a notation for attaching program fragments to the productions of a grammar. The program fragments are executed when the production is used during syntax analysis. The combined result of all these fragment executions, in the order induced by the syntax analysis, produces the translation of the program to which this analysis/synthesis process is applied.

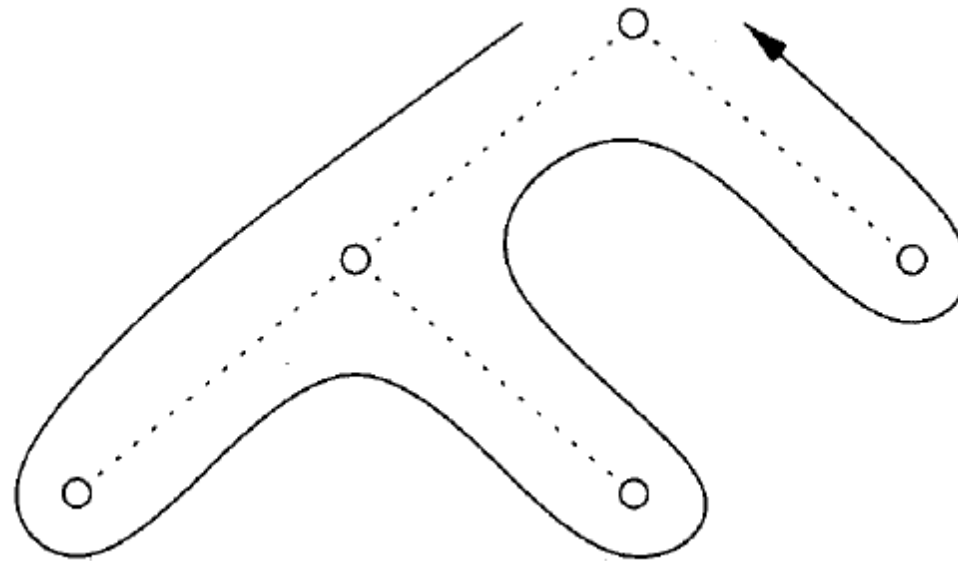
Synthesized Attributes



PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \parallel term.t \parallel '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \parallel term.t \parallel '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Tree Traversals

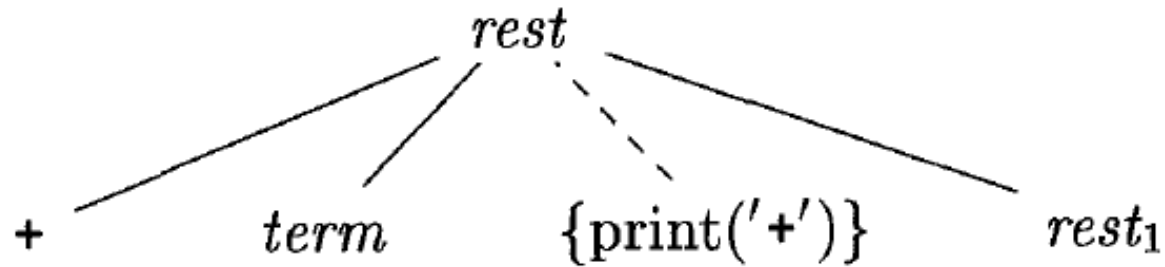
- Tree traversals will be used for describing attribute evaluation and for specifying the execution of code fragments in a translation scheme.



depth-first traversal

Translation Schemes

- A syntax-directed translation scheme is a notation for specifying a translation by attaching program fragments to productions in a grammar.
- The position at which an action is to be executed is shown by enclosing it between curly braces and writing it within the production body, as in $\text{rest} \rightarrow + \text{term} \{ \text{print}(' + ') \} \text{rest}_1$



Translation Schemes

$\text{expr} \rightarrow \text{expr}_1 + \text{term} \{\text{print}(' + ')\}$

$\text{expr} \rightarrow \text{expr}_1 - \text{term} \{\text{print}(' - ')\}$

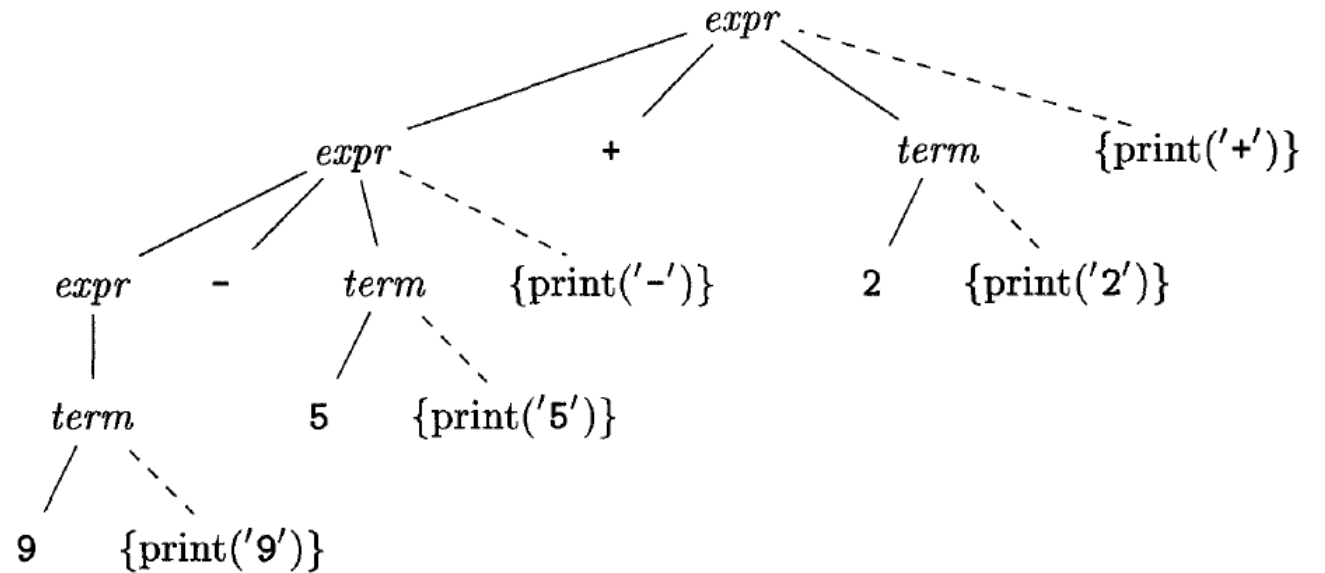
$\text{expr} \rightarrow \text{term}$

$\text{term} \rightarrow 0 \{\text{print}(' 0 ')\}$

$\text{term} \rightarrow 1 \{\text{print}(' 1 ')\}$

...

$\text{term} \rightarrow 9 \{\text{print}(' 9 ')\}$



Translation Schemes

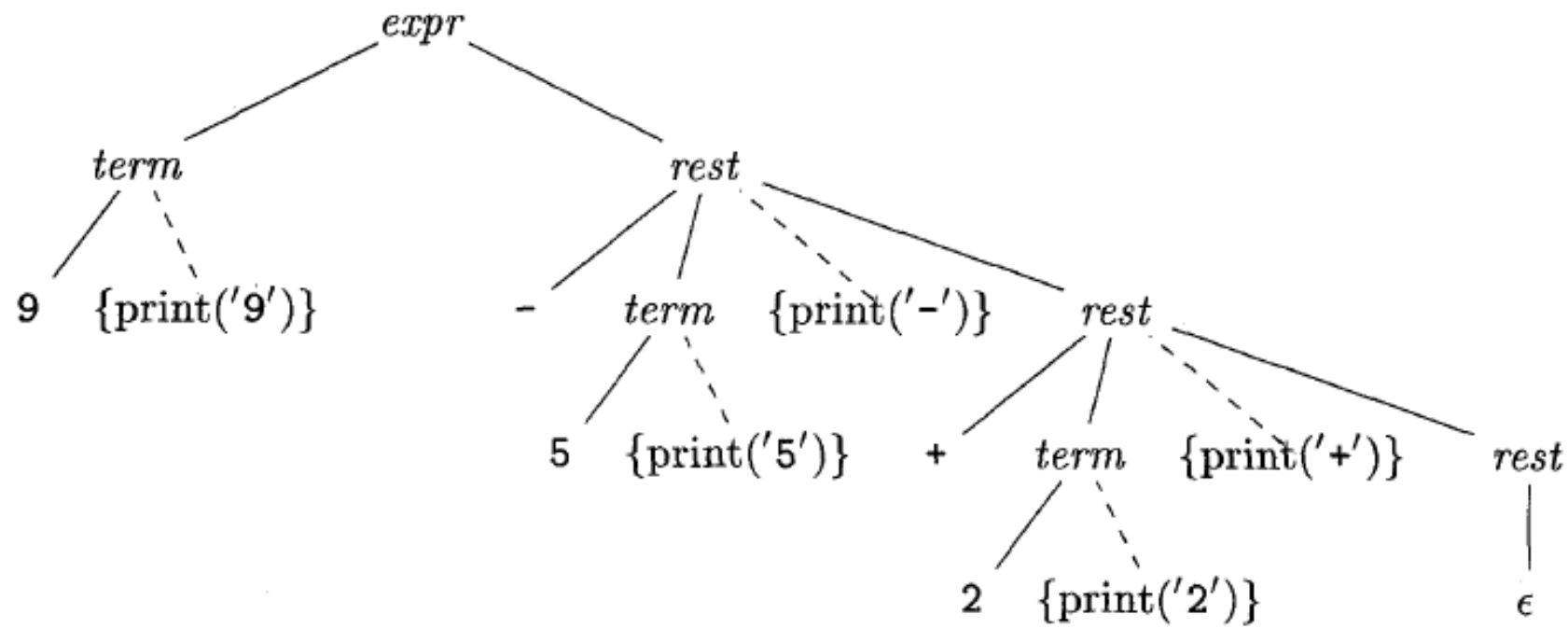
- Let

$$\begin{aligned} A &= \text{expr} \\ \alpha &= + \text{term} \{ \text{print}(' + ') \} \\ \beta &= - \text{term} \{ \text{print}(' - ') \} \\ \gamma &= \text{term} \end{aligned}$$

- Then the left-recursion-eliminating transformation produces the translation scheme

$$\begin{aligned} \text{expr} &\rightarrow \text{term rest} \\ \text{rest} &\rightarrow \begin{array}{l} + \text{term} \{ \text{print}(' + ') \} \text{rest} \\ - \text{term} \{ \text{print}(' - ') \} \text{rest} \\ \epsilon \end{array} \\ \text{term} &\rightarrow \begin{array}{l} 0 \{ \text{print}(' 0 ') \} \\ 1 \{ \text{print}(' 1 ') \} \\ \dots \\ 9 \{ \text{print}(' 9 ') \} \end{array} \end{aligned}$$

Translation Schemes



Semantic Analysis

- For the ac language, we focus on two aspects of semantic analysis: symbol table construction and type checking.

/★ 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
```

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

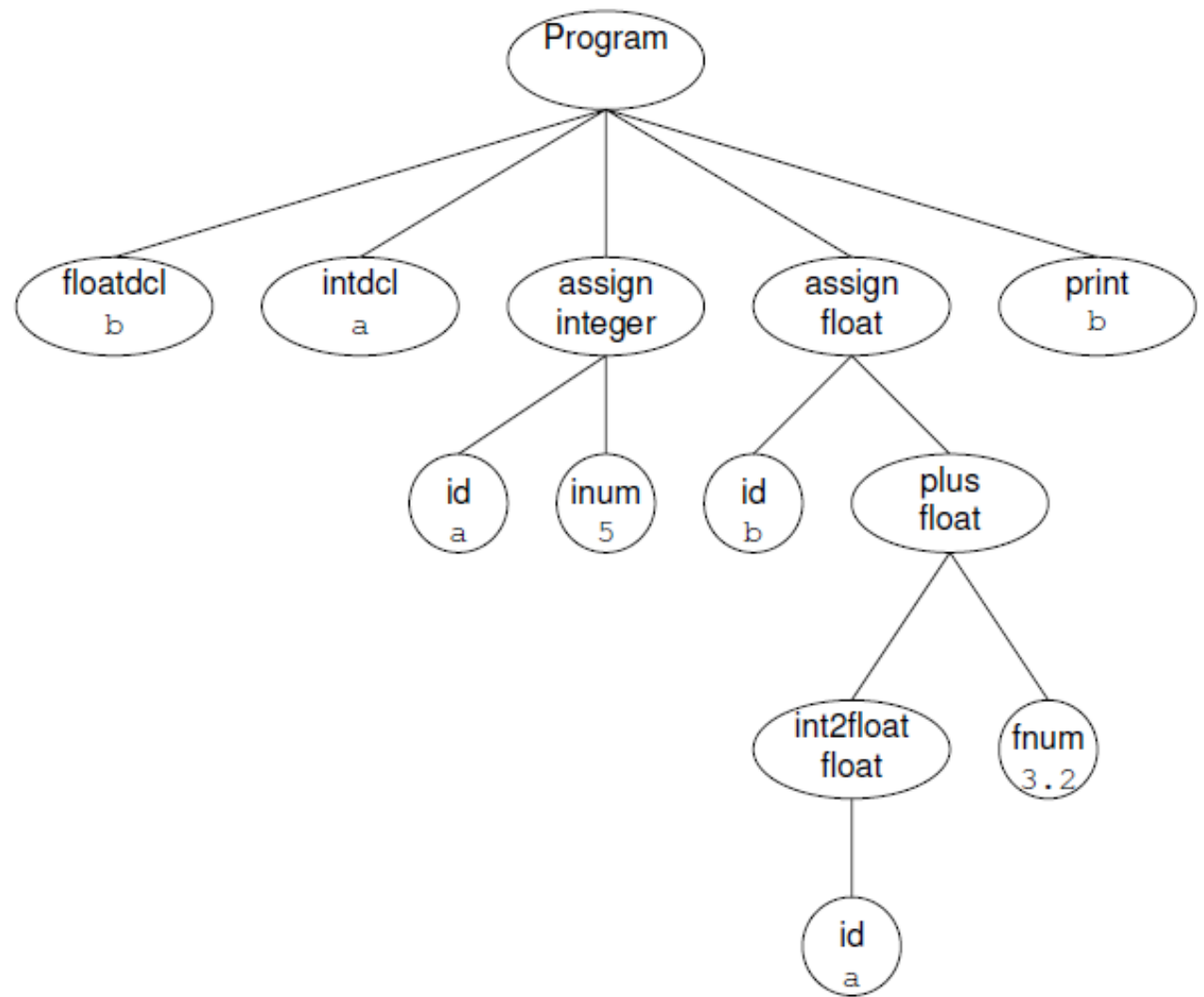
then

 /★ replace node *n* by convert-to-float of node *n* ★/

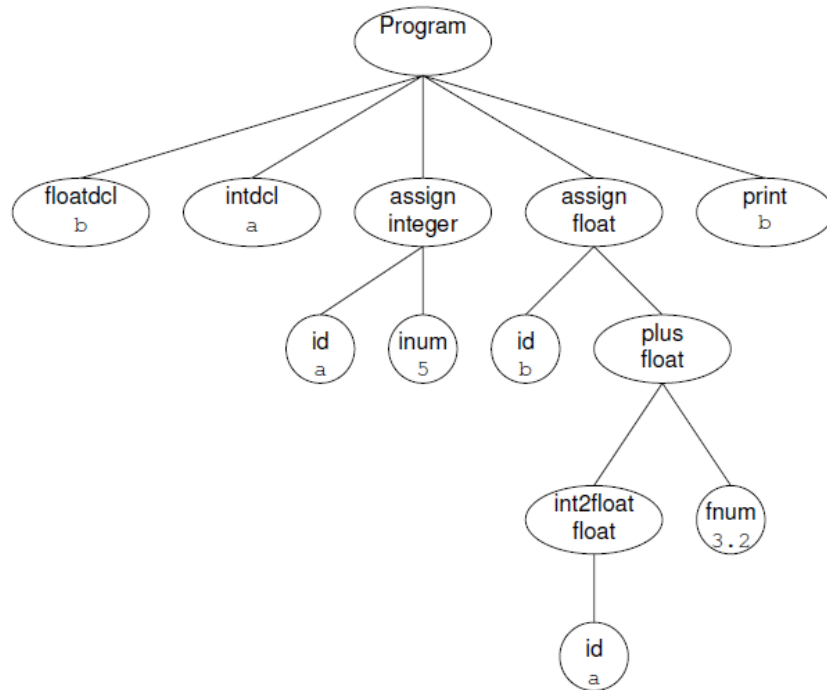
else /★ nothing needed ★/

end

AST after semantic analysis



Code Generation



```
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
```

Code Generation

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