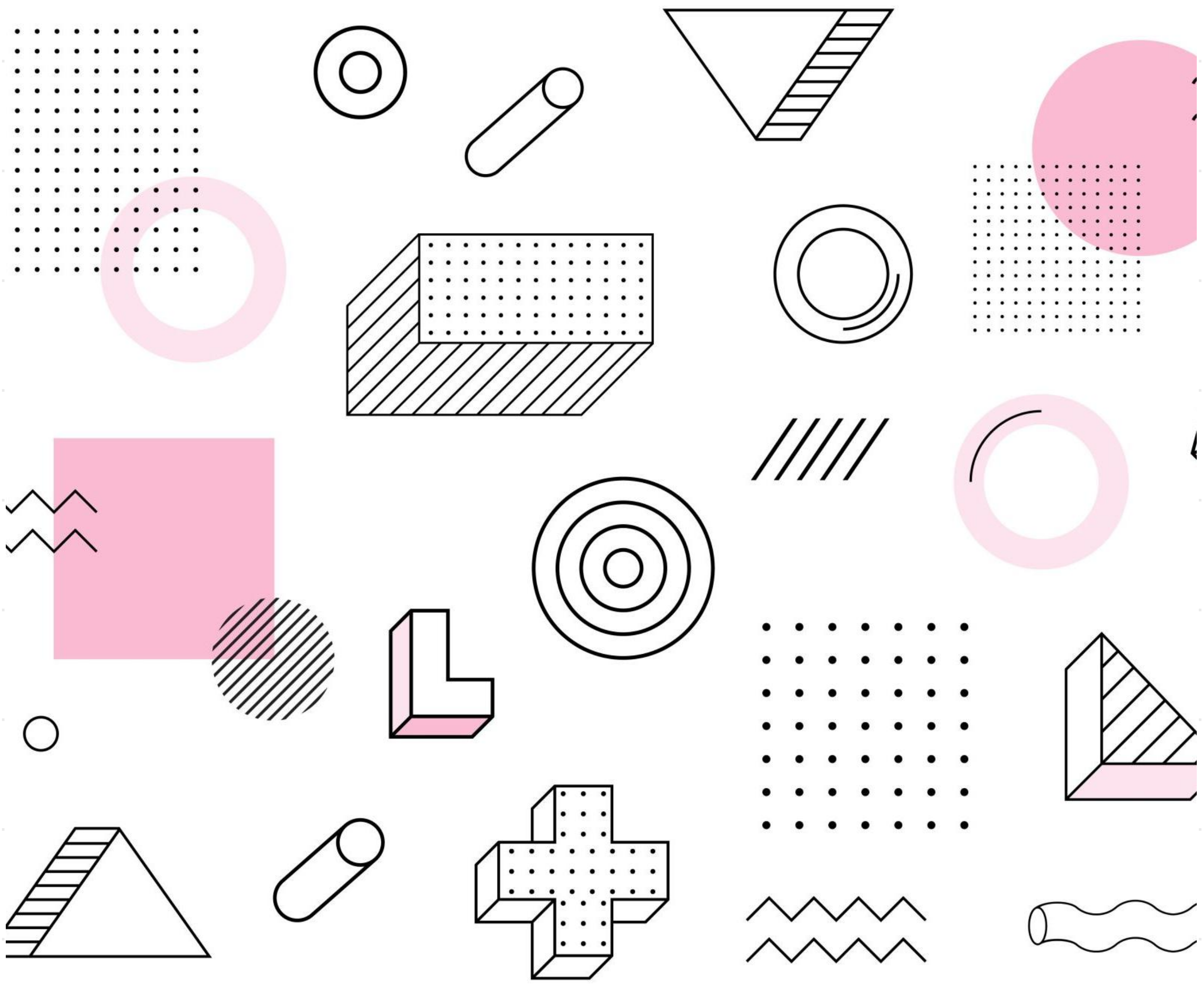
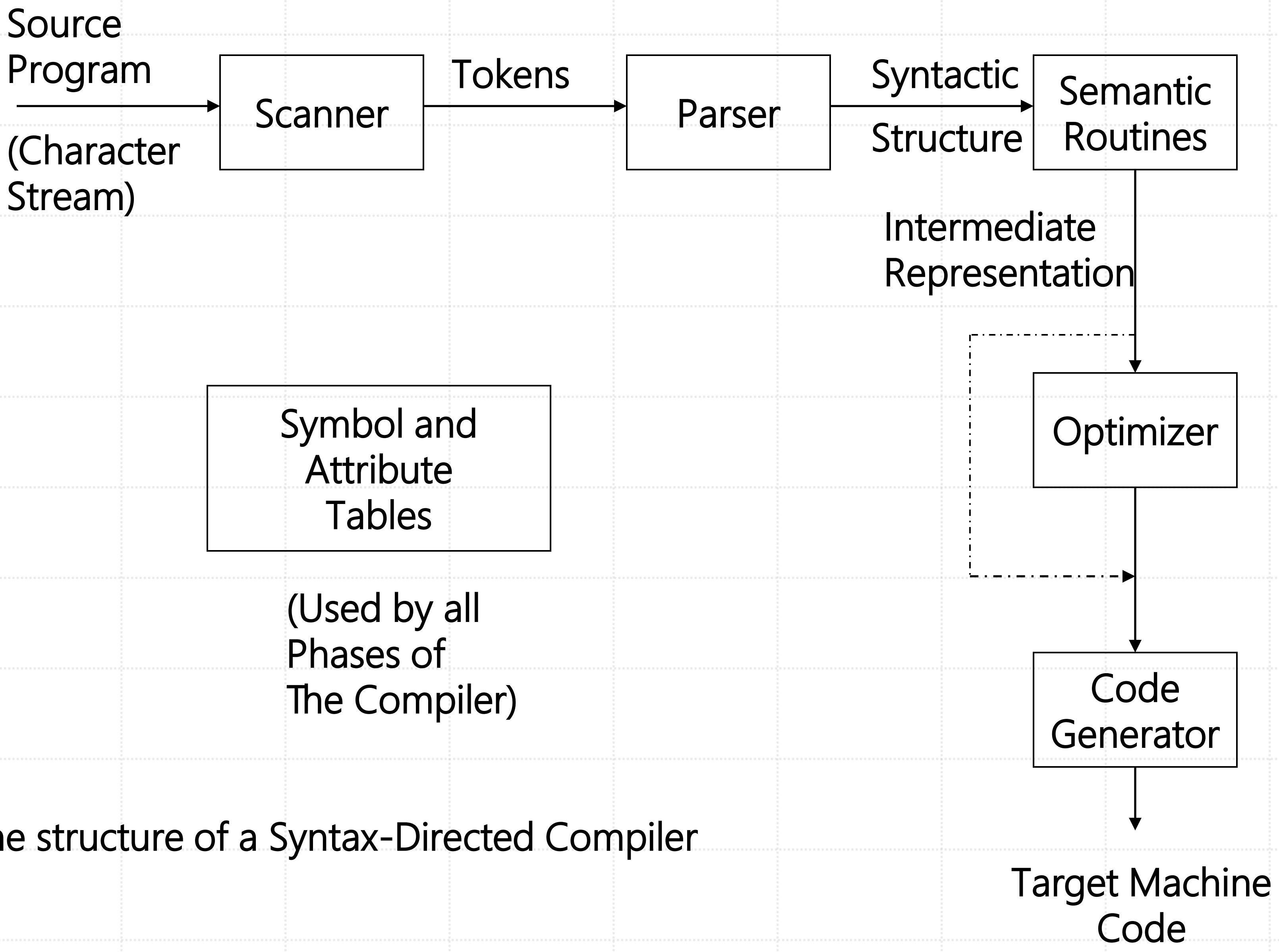
Chapter 2: A Simple Compiler

陳奇業 成功大學資訊工程系



Outlines



The structure of a Syntax-Directed Compiler

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).

- 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

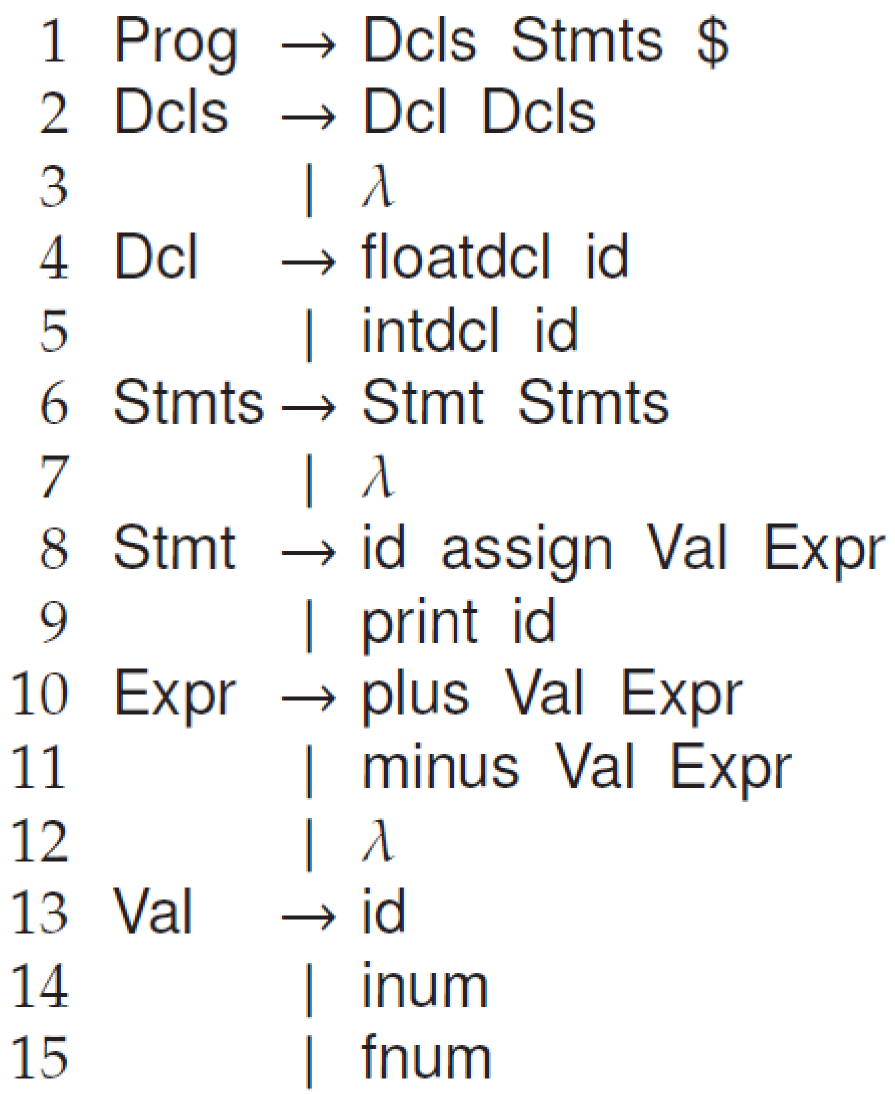
- Ac's syntax is defined by a context-free grammar (CFG)
- CFG is also called BNF (Backus-Naur Form 巴科斯範式) grammar
- CFG consists of a set of production rules,

$$\underbrace{A}_{\text{LHS}} \to \underbrace{B\ C\ D\ldots\ Z}_{\text{RHS}}$$

LHS must be a single nonterminal

RHS consists 0 or more terminals or nonterminals

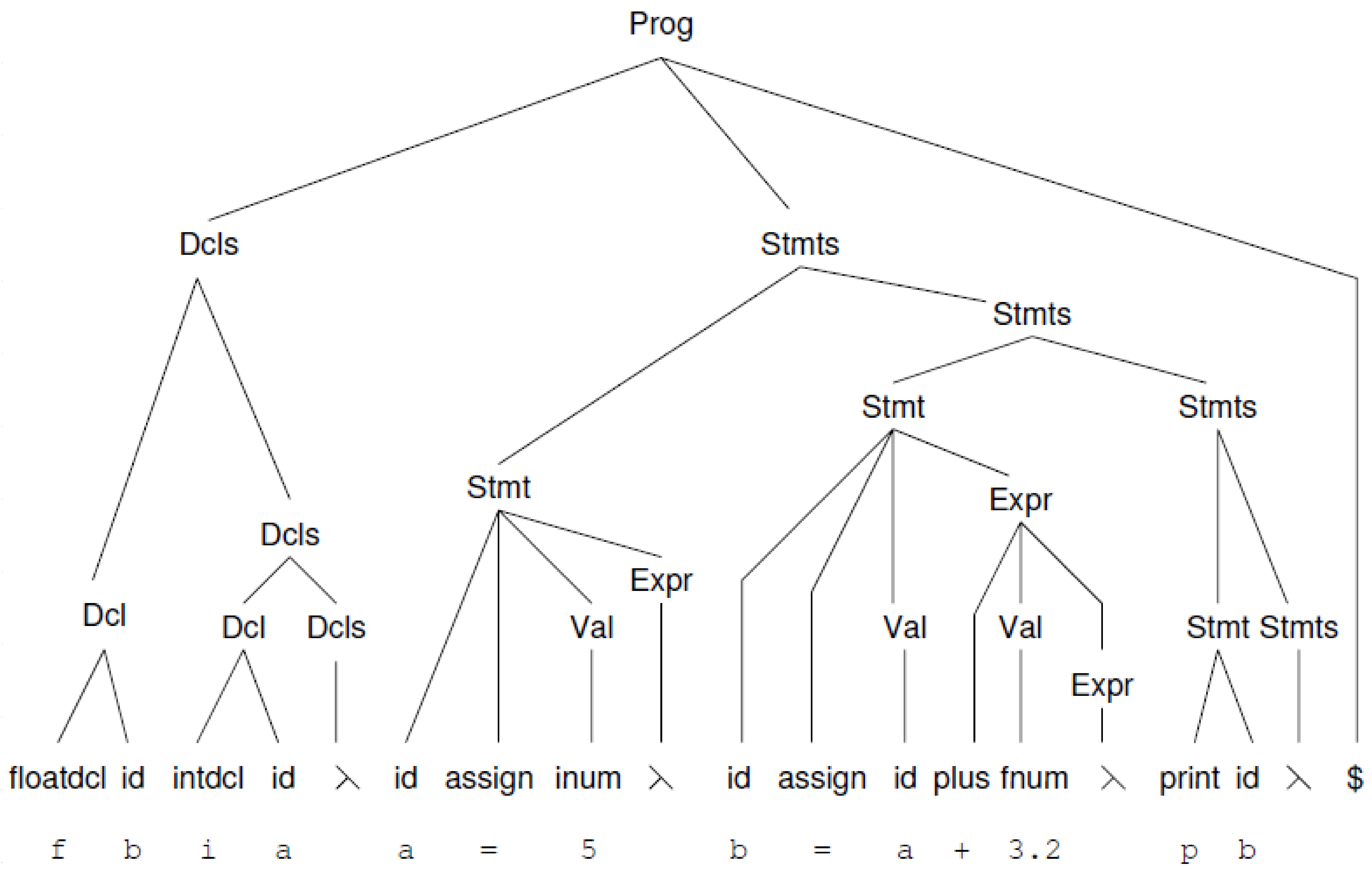
Syntax Specification



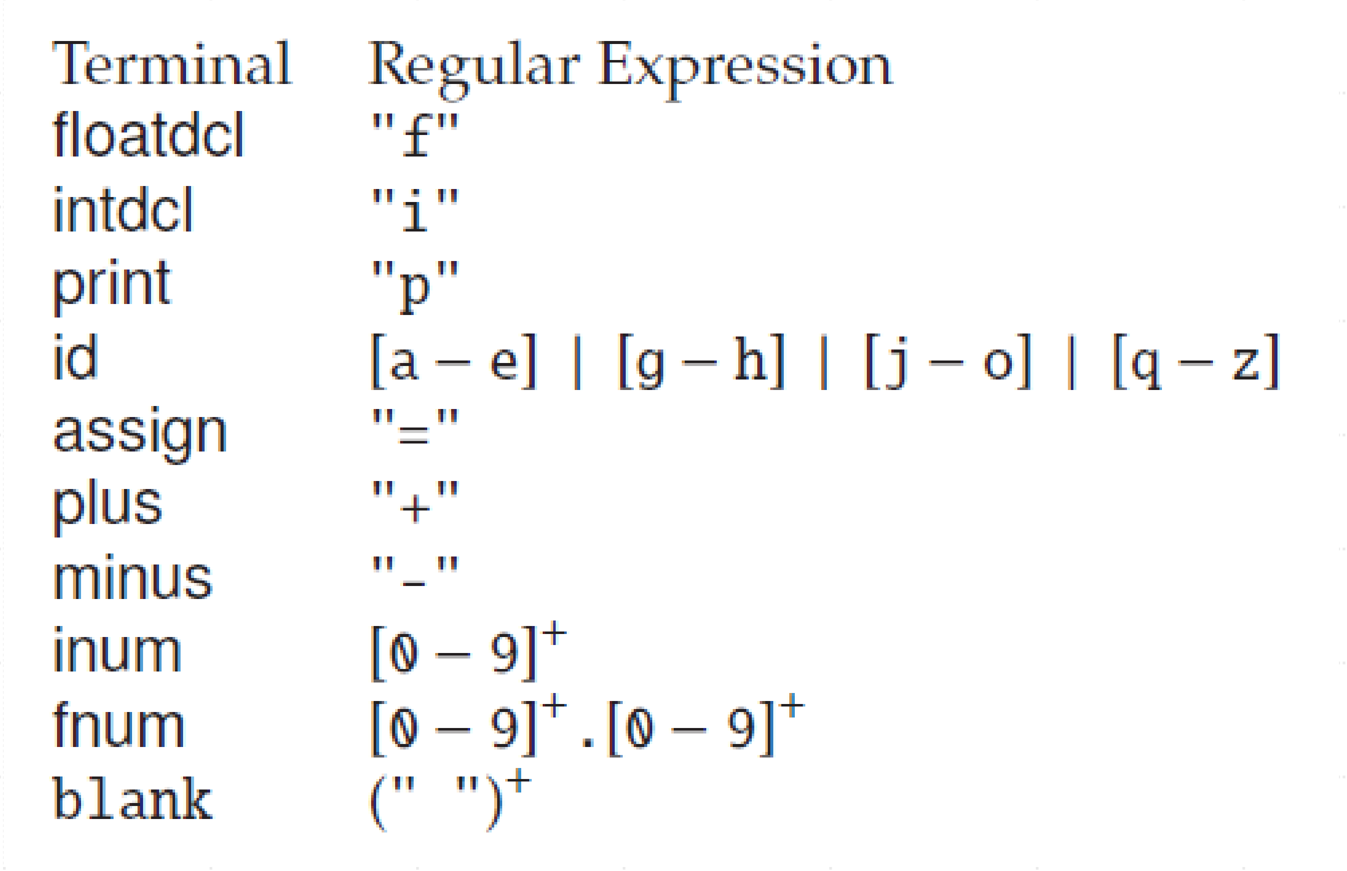
1	Prog	→	Dcls Stmts $
2	Dcls	→	Dcl Dcls
3		|	λ
4	Dcl	→	floatdcl id
5		|	intdcl id
6	Stmts	→	Stmt Stmts
7		|	λ
8	Stmt	→	id assign Val Expr
9		|	print id
10	Expr	→	plus Val Expr
11		|	minus Val Expr
12		|	λ
13	Val	→	id
14		|	inum
15		|	fnum

Step	Sentential Form	Production Number
1	⟨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

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


Prog
Dcls
Stmts
Dcls
Stmts
Stmt
Stmts
Stmt
Expr
Dcls
Dcl
Dcl
Dcls
Expr
Val
Val
Val
Stmt Stmts
Expr
floatdcl id intdcl id λ id assign inum λ id assign id plus fnum λ print id λ $
f b i a a = 5 b = a + 3.2 p b

Token Specification



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	$(\texttt{" "})^+$

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

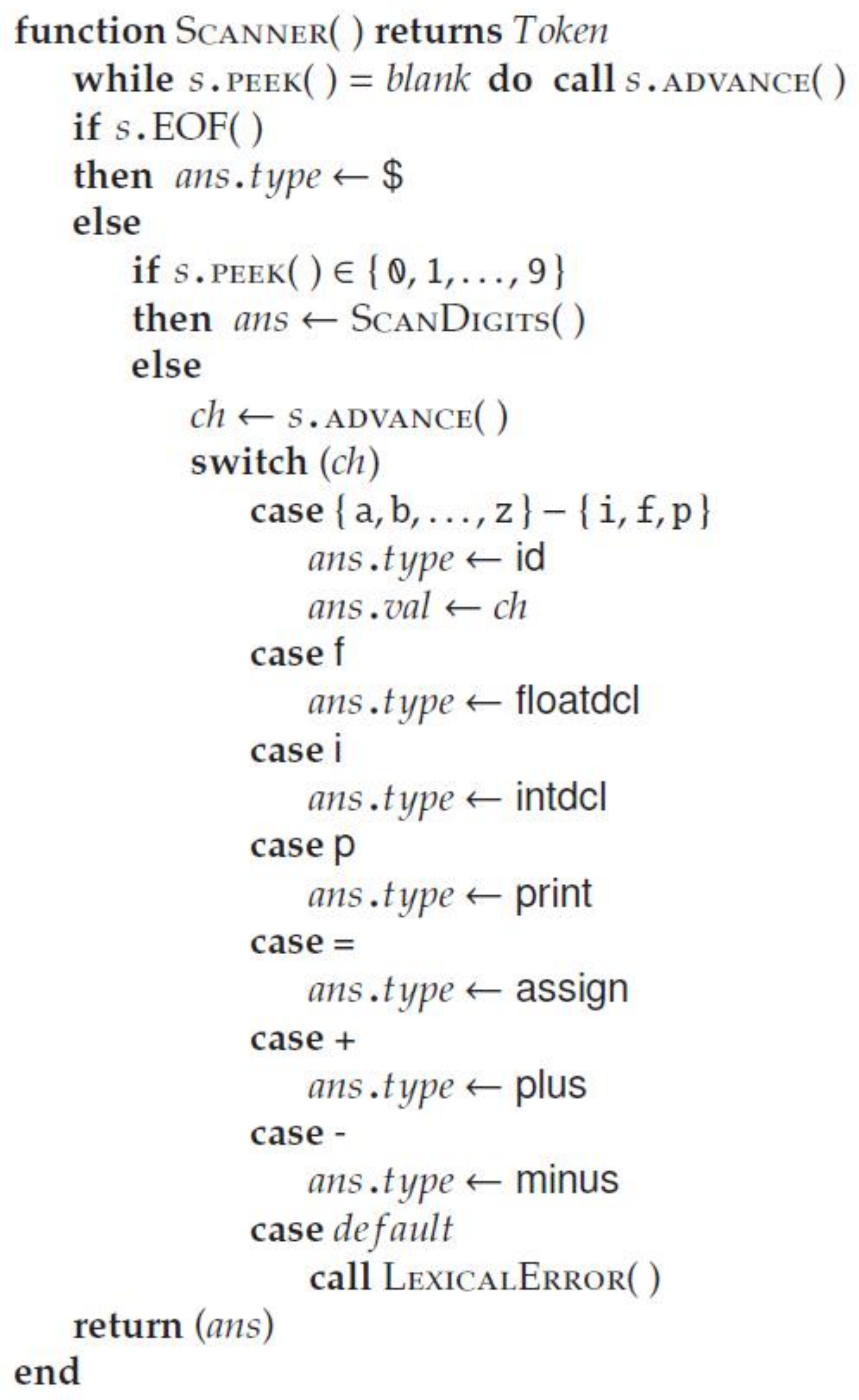
- The scanner's job is to translate a stream of characters into a stream of tokens, where each token represents an instance of some terminal symbol.
- Each token found by the scanner has the following two components:
  1. A token's type explains the token's membership in the terminal alphabet. All instances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

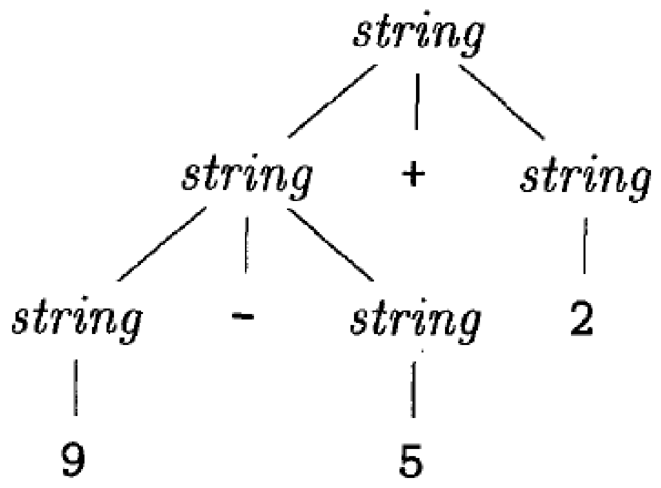
```
function ScanDigits( ) returns token
    tok.val ← " "
    while s.peek( ) ∈ {0, 1, ..., 9} do
        tok.val ← tok.val + s.advance( )
    if s.peek( ) ≠ "."
    then  tok.type ← inum
    else
        tok.type ← fnum
        tok.val ← tok.val + s.advance( )
        while s.peek( ) ∈ {0, 1, ...,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

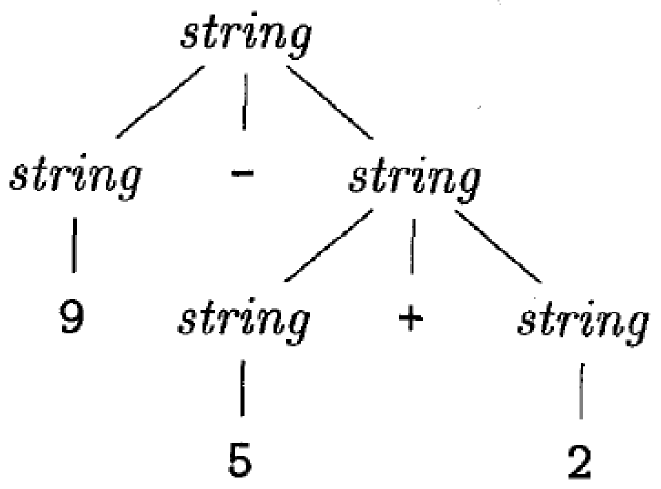
Parsing

- 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
  string → string + string | string - string | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



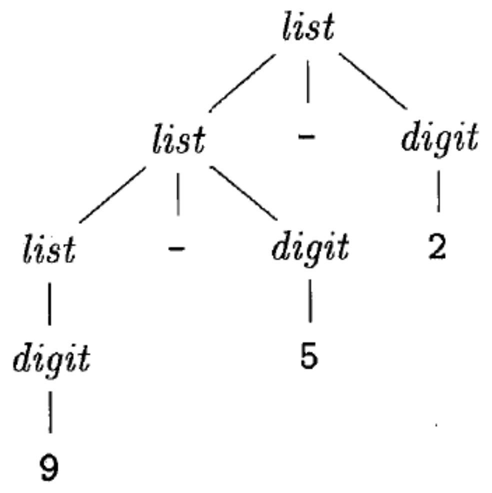


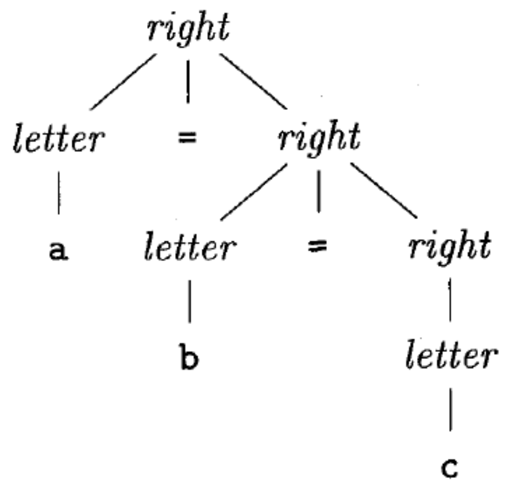
Associativity of Operators

- 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:
  right → letter = right | letter
  letter → a | b | … | z





Precedence of Operators

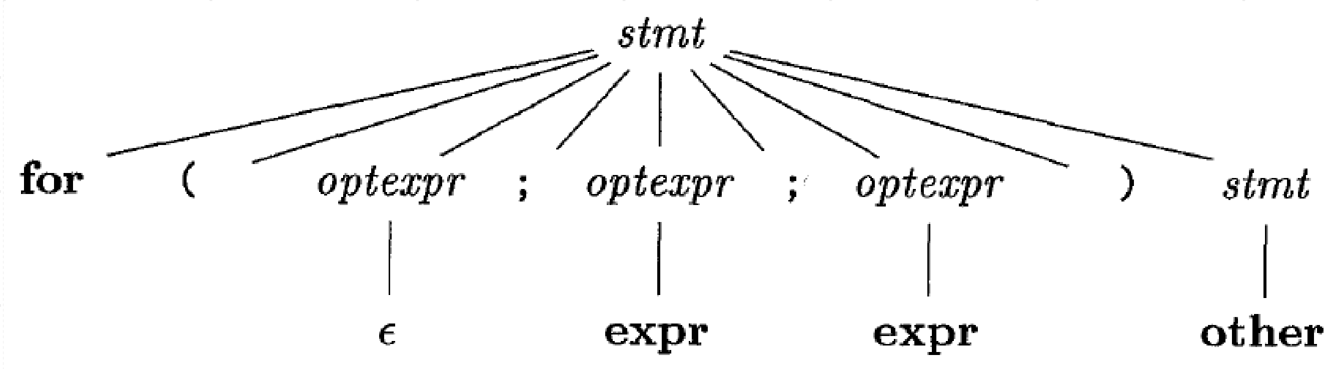
- Consider the expression 9+5*2. There are two possible interpretations of this expression: (9+5) *2 or 9+ (5*2).
- Ex. : A grammar for arithmetic expressions can be constructed from a table showing the associativity and precedence of operators.
  expr → expr + term | expr - term | term
  term → term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

- stmt → expr ;
  | if (expr) stmt
  | for (optexpr ; optexpr ; optexpr) stmt
  | other
- optexpr →ε | 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:

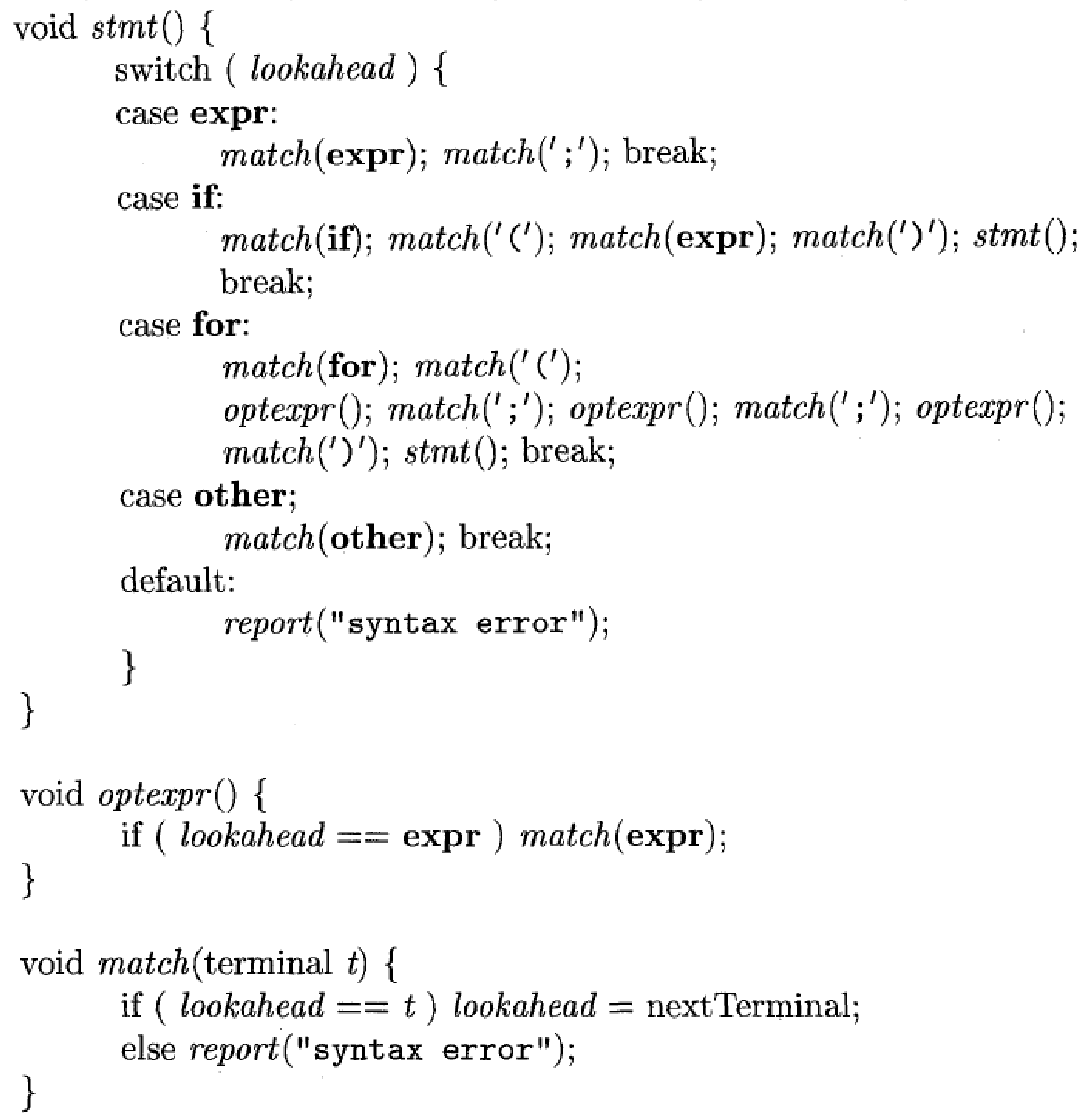
Stmt→id assign Val Expr

Stmt→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.
- FIRST(stmt) = {expr, if, for, other}



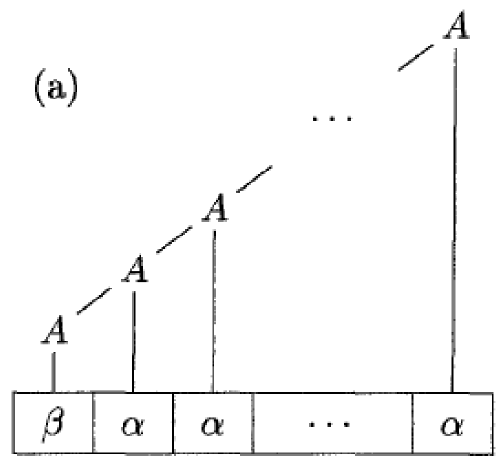
```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                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");
}
```

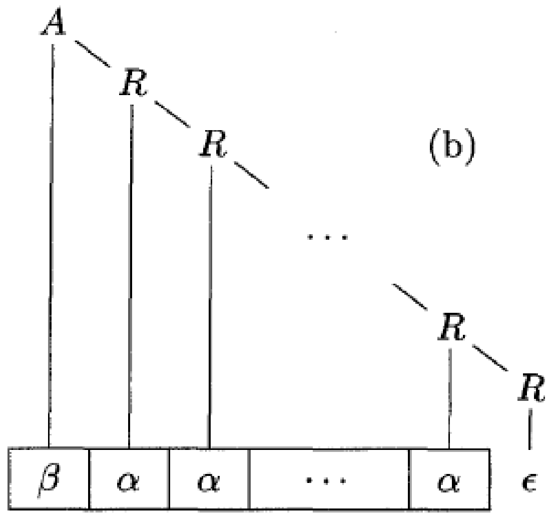
Left Recursion

- It is possible for a recursive-descent parser to loop forever. A problem arises with "left-recursive" productions like
  expr → expr + 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= expr, α= + term, β= term

Left Recursion

- We can convert left recursion to right recursion in the following manner, using a new nonterminal R:
  $A \rightarrow \beta R$
  $R \rightarrow \alpha R | \epsilon$



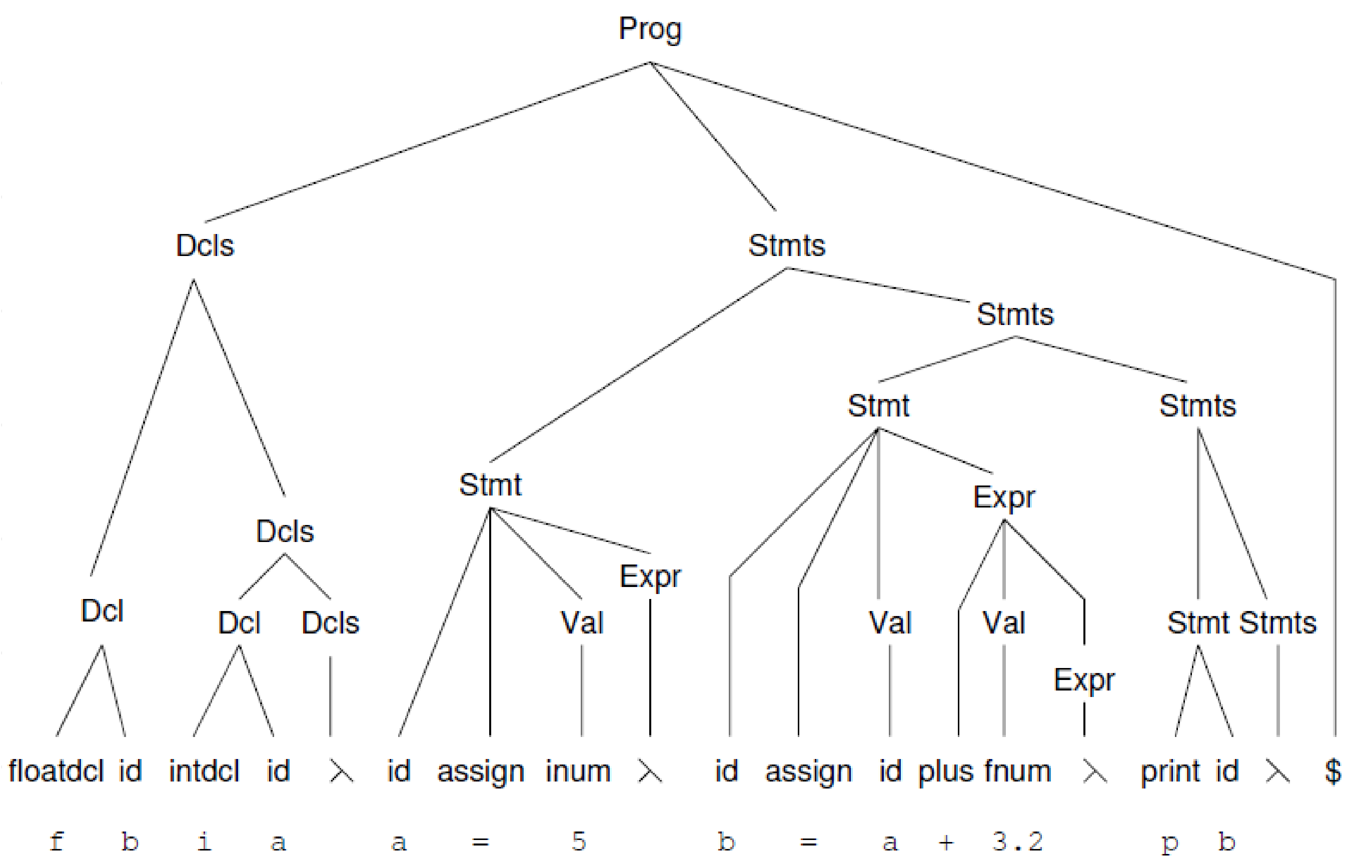


Abstract Syntax Trees

- 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.

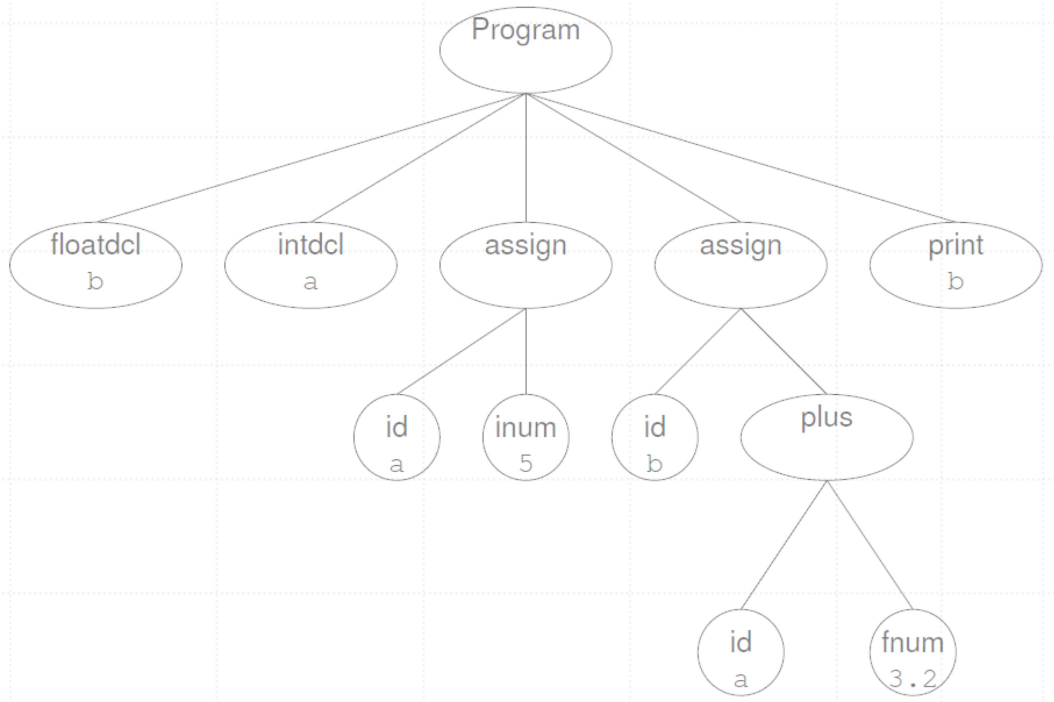
Abstract Syntax Trees

- 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.



Prog
Dcls
Stmts
Stmts
Stmt
Stmts
Stmt
Expr
Dcls
Dcl
Dcl
Dcls
Val
Expr
Val
Val
Stmt Stmts
Expr
floatdcl id intdcl id λ id assign inum λ id assign id plus fnum λ print id λ $
f b i a a = 5 b = a + 3.2 p b

Parse Tree

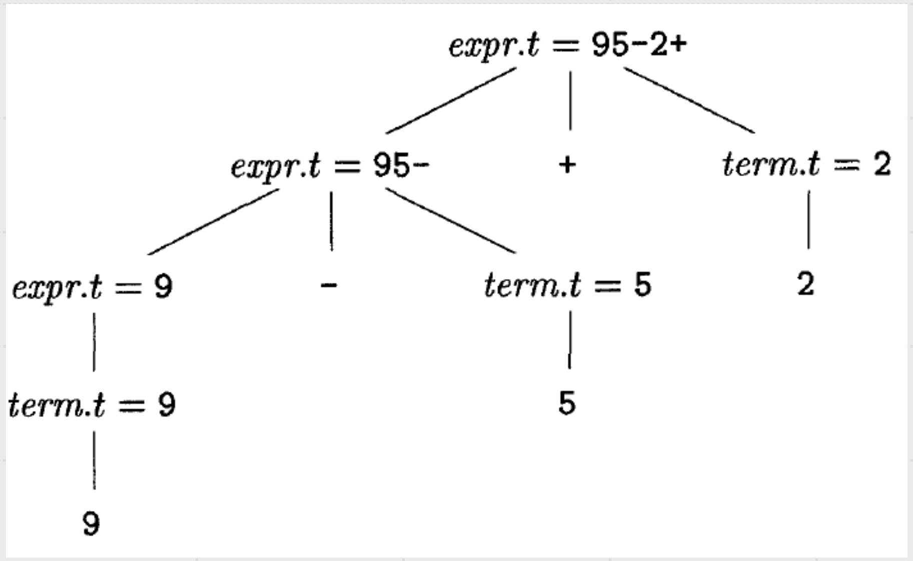


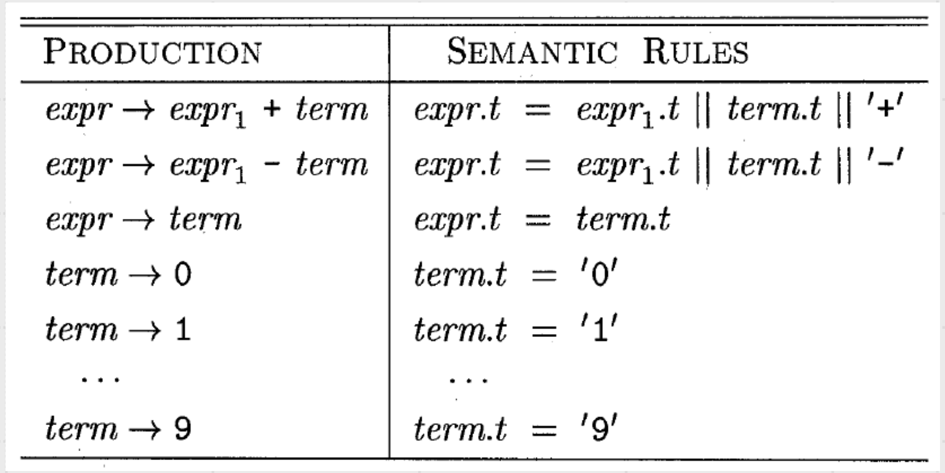
Abstract Syntax Tree

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

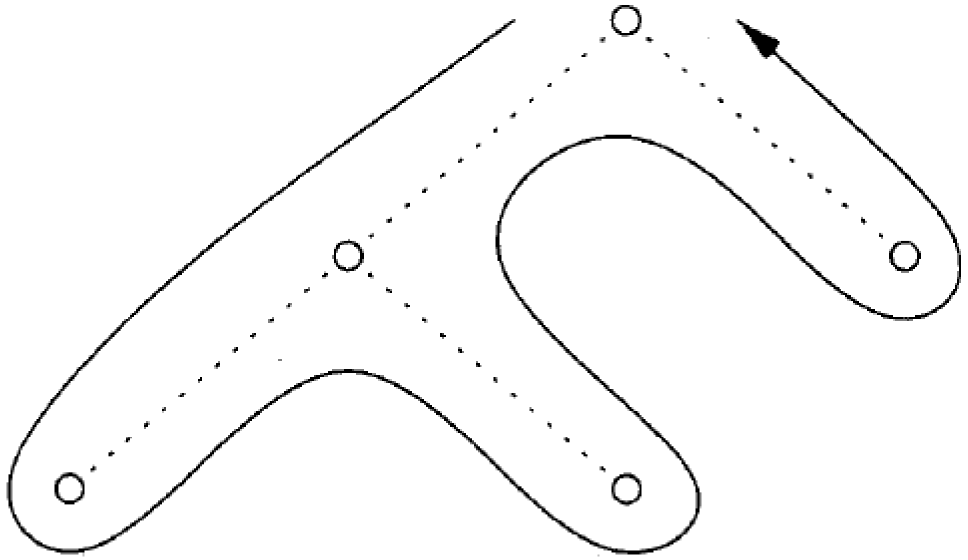




PRODUCTION	SEMANTIC RULES
$expr \to expr_1 + term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|\, '+'$
$expr \to expr_1 - term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|\, '-'$
$expr \to term$	$expr.t = term.t$
$term \to 0$	$term.t = '0'$
$term \to 1$	$term.t = '1'$
...	...
$term \to 9$	$term.t = '9'$

Tree Traversals

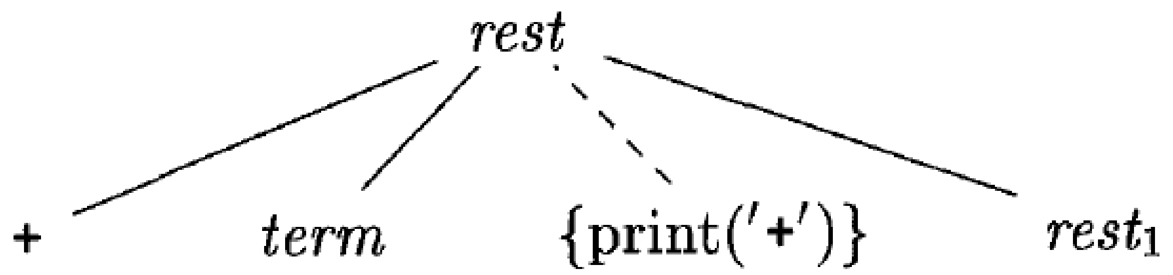
- Tree traversals will be used for describing attribute evaluation and for specifying the execution of code fragments in a translation scheme.



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
  rest → + term {print('+')} rest_1



Translation Schemes

expr → $expr_1$ + term {print('+')}

expr → $expr_1$ - term {print('-')}

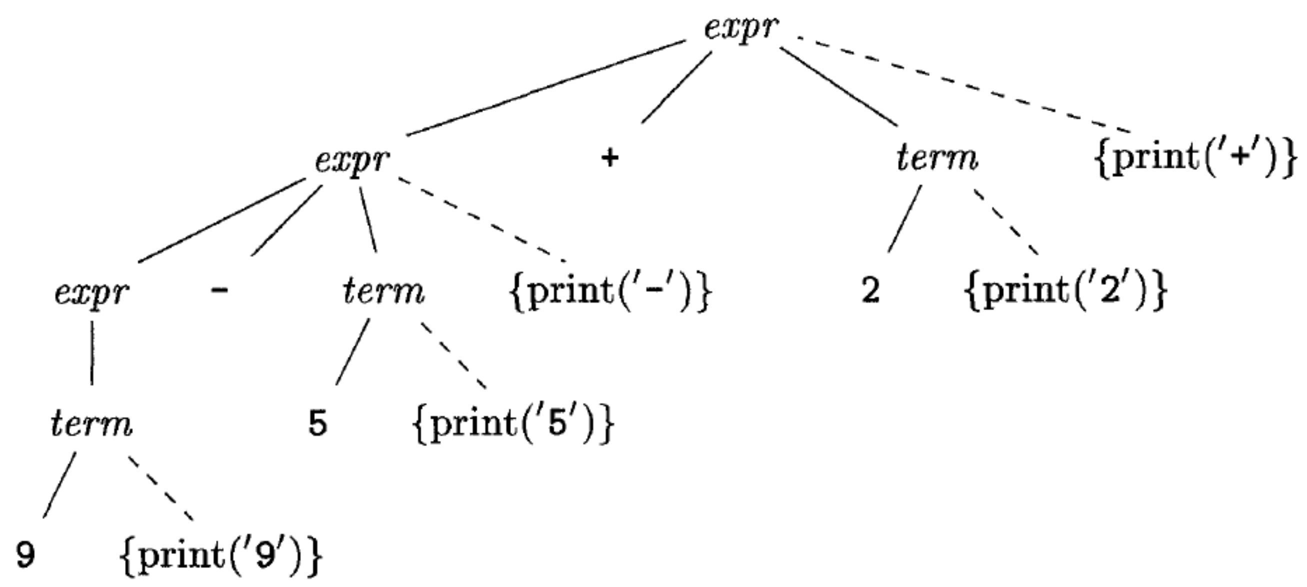
expr → term

term → 0 {print ('0')}

term → 1 {print (' 1')}

…

term → 9 {print ('9')}



Translation Schemes

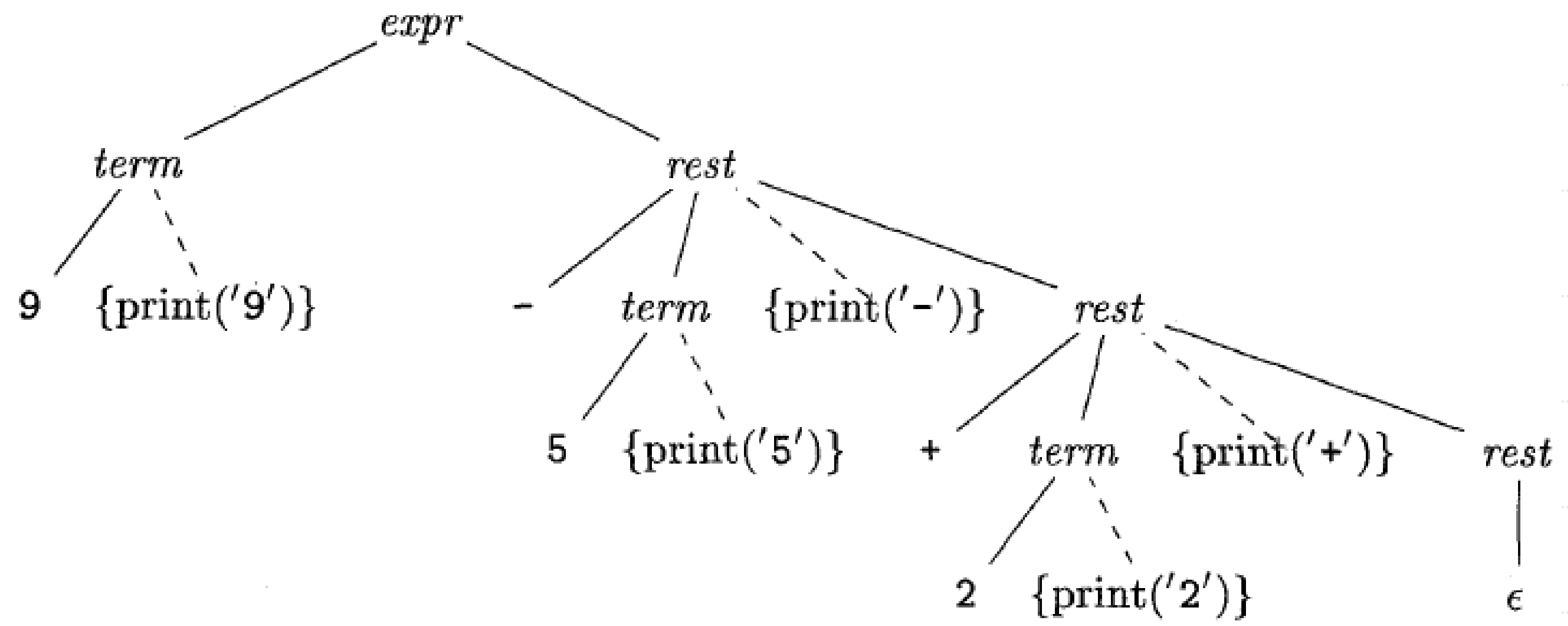
- Let

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

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

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

Translation Schemes



Semantic Analysis

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

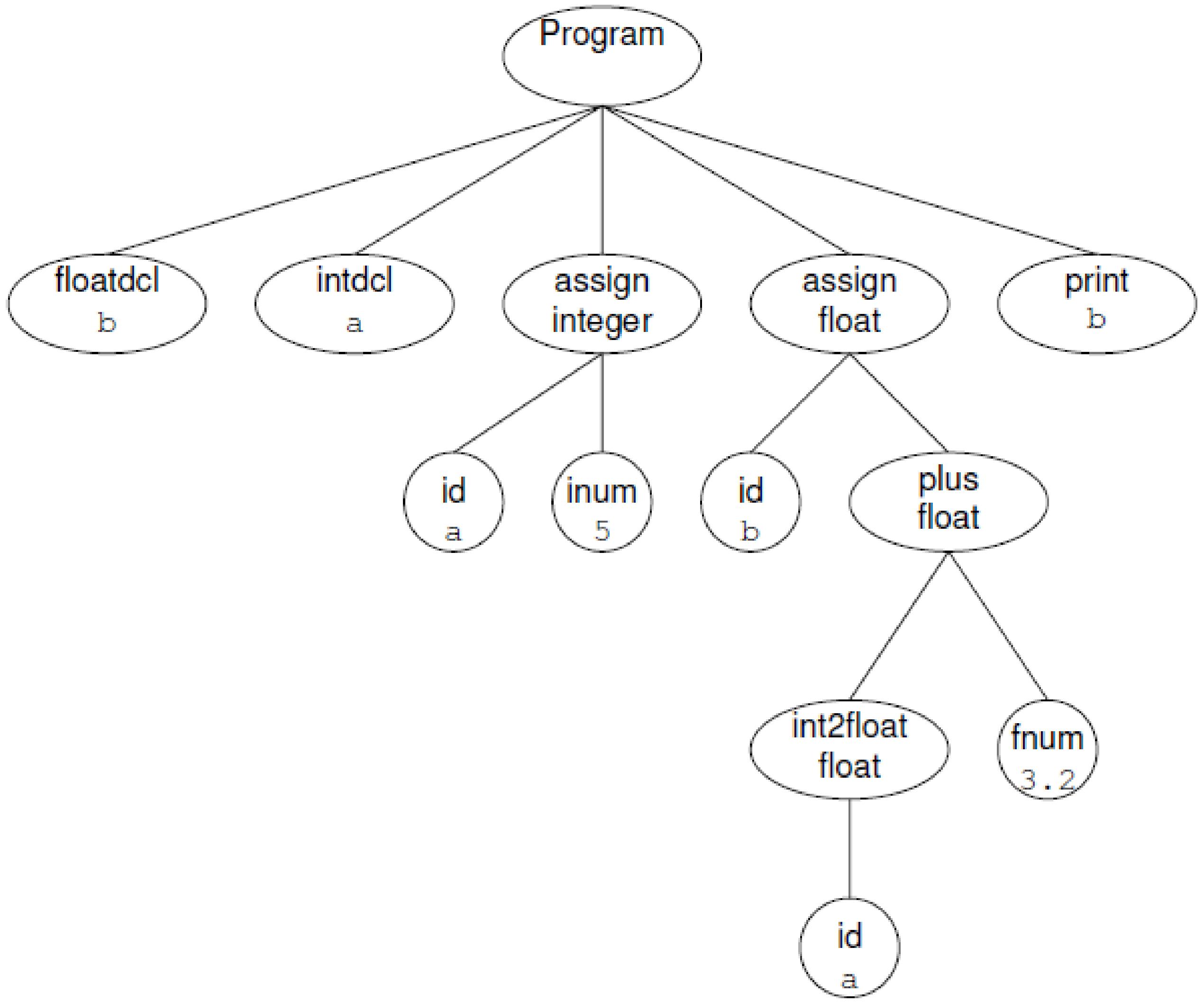
```
/★   Visitor methods
procedure visit( SymDeclaring n )
    if n.getType( ) = floatdcl
    then  call EnterSymbol(n.getId( ), float)
    else  call EnterSymbol(n.getId( ), integer)
end

/★   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 $n$ )
    $n.type \leftarrow$ 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
```

```
/*    Type-checking utilities                                    */
function Consistent( $c1, c2$ ) returns $type$
    $m \leftarrow$ 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  $ans \leftarrow$ float
    else  $ans \leftarrow$ 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
```

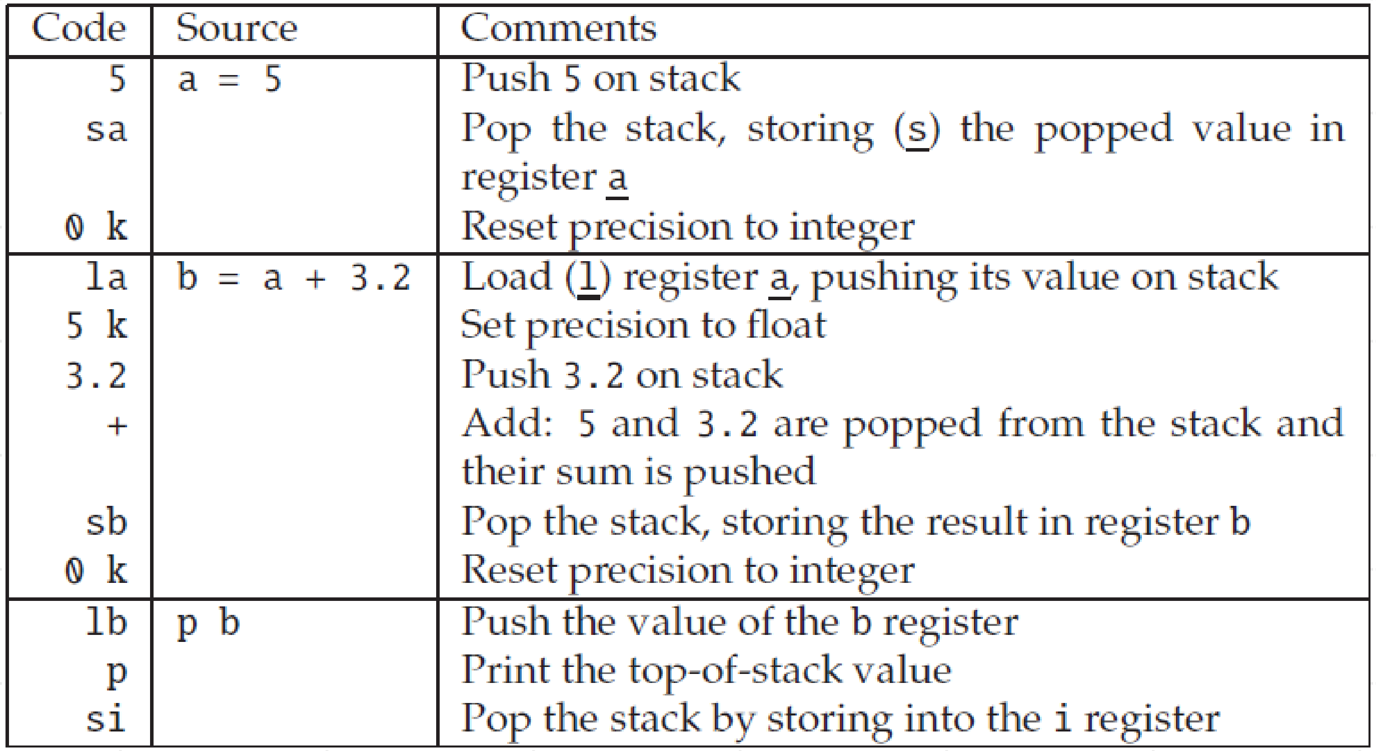
AST after semantic analysis



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`	`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