CHAPTER 3 Describing Syntax and Semantics

Language – collection of sentences satisfying given syntax rules.

Formal language: Fortran, C, Java, …
Natural language: English, German, …

- **Syntax (form)**
  - **rules governing the creation of valid sentences.**
- **Semantics (meaning)**
  - **meaning attached to a valid sentence.**

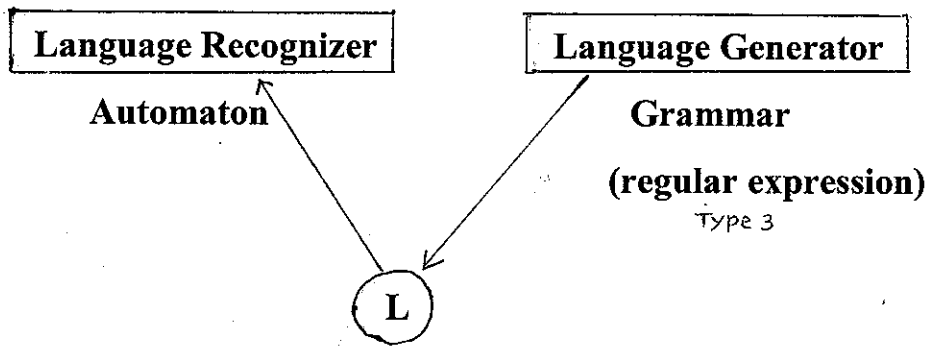
Example. a = b + c; c = (a + b) * c;

syntax:

```
<assign> ::= <id> = <expr> ;
<id>     ::= a | b | c
<expr>   ::= <id> + <expr>
           | <id> * <expr>
           | ( <expr> )
           | <id>
```

semantics of a = b + c;

Evaluate the RHS (expr) , and put the value on LHS (id)
Add b and c, and place the value in a.



Chomsky Hierarchy

Type	Name	Restrictions w1 --> w2	Acceptor
0	Phrase-Structure	w1 = any string with at least 1 nonterminal w2 = any string	Turing Machine
1	Context-Sensitive	w1 = any string with at least 1 nonterminal w2 = any string at least as long as w1	Bounded Turing Machine
2	Context-Free	w1 = one non-terminal w2 = any string	Push-Down Automaton
3	Regular	w1 = one non-terminal w2 = tA or t (t = terminal A = non-term.)	Finite State Automaton

Note: The empty production is included in each of the grammars.

Lexemes and Tokens

index = 2 * count + 17;

LEXEMES	TOKENS
index	identifier
=	equal_sign
2	int_constant
*	mult_op
count	identifier
17	int_contstant
+	plus_sign
;	semicolon

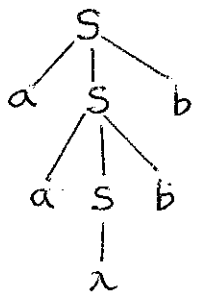
Context free language (Type 2 language)

- **pushdown (stack) capability**

Ex. 1 **G: $S \rightarrow aSb \mid \lambda$**

$L(G) = \{ a^n b^n \mid n \geq 0 \}$

String aabb



Ex. 2 **G: $S \rightarrow aSa \mid bSb \mid \lambda$**

$L(G) = \{ww^R : w \in \{a, b\}^*\}$ **parlindrome**

MADAM, I'M ADAM

A MAN A PLAN A CANAL PANAMA

Example 3.1 Grammar for Small Language

nonterminal derives terminal

Production **<program>** --> ***begin*** **<stmt_list>** ***end***

start symbol is replaced by

<stmt_list> --> <stmt>
| <stmt>; <stmt_list> right recursive

<stmt> --> <var> := <expression>

<var> --> A | B | C

<expression> --> <var> + <var>
| <var> - <var>
| <var>

$G = \langle N, T, S, P \rangle$

Derivation of program

```
begin
  A := B + C;
   B := C
end
```

```
<program> => begin <stmt_list> end
          => begin <stmt>; <stmt_list> end
          => begin <var> := <expression>;
                                <stmt_list> end
          => begin A := <expression>;
                                <stmt_list> end
          => begin A := <var> + <var>;
                                <stmt_list> end
          => begin A := B + <var>;
                                <stmt_list> end
          => begin A := B + C; <stmt_list> end
          => begin A := B + C; <stmt> end
          => begin A := B + C;
                        <var> := <expression> end
          => begin A := B + C;
                        B := <expression> end
          => begin A := B + C; B := <var> end
          => begin A := B + C; B := C end
```

Example 3.2

A grammar for simple assignment statements

```
<assign> → <id> := <expr>
   <id>  → A | B | C
 <expr>  → <id> + <expr>
         | <id> * <expr>
         | ( <expr> )
         | <id>  ■
```

This grammar describes assignment statements whose right sides are arithmetic expressions with multiplication and addition operators and parentheses. For example, the statement

```
A := B * ( A + C )
```

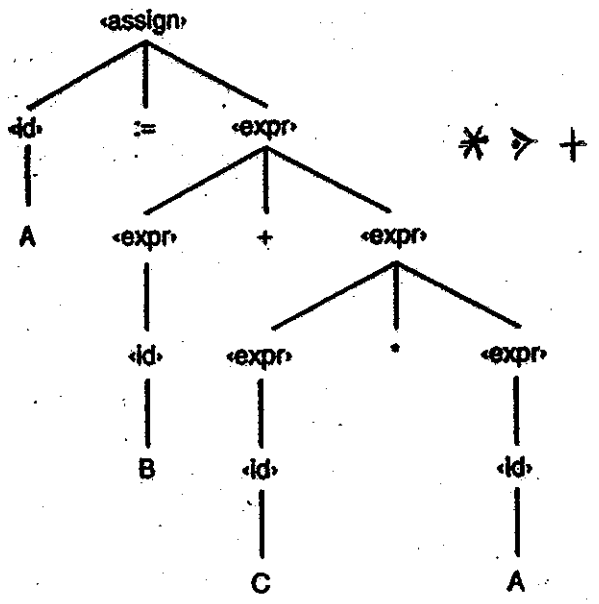
is generated by the following derivation:

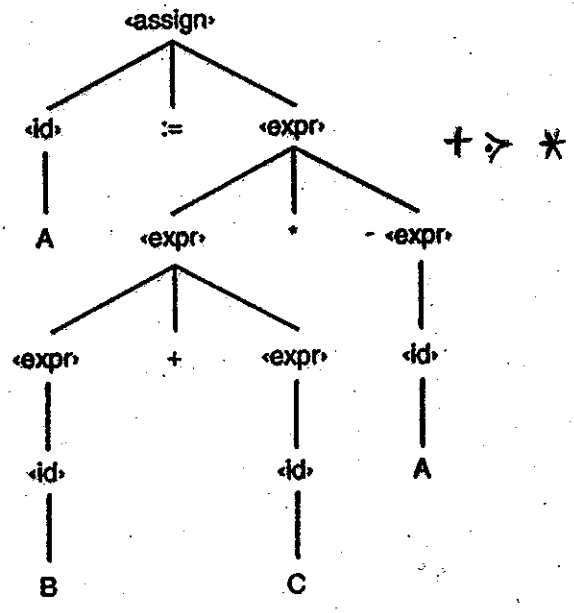
```
<assign> ⇒ <id> := <expr>
         ⇒ A := <expr>
         ⇒ A := <id> * <expr>
         ⇒ A := B * <expr>
         ⇒ A := B * ( <expr> )
         ⇒ A := B * ( <id> + <expr> )
         ⇒ A := B * ( A + <expr> )
         ⇒ A := B * ( A + <id> )
         ⇒ A := B * ( A + C )
```

Example 3.3 An ambiguous grammar for simple assignment statements

```
<assign> → <id> := <expr>
   <id>  → A | B | C
 <expr>  → <expr> + <expr>
         | <expr> * <expr>
         | ( <expr> )
         | <id>
```

A := B + C * A





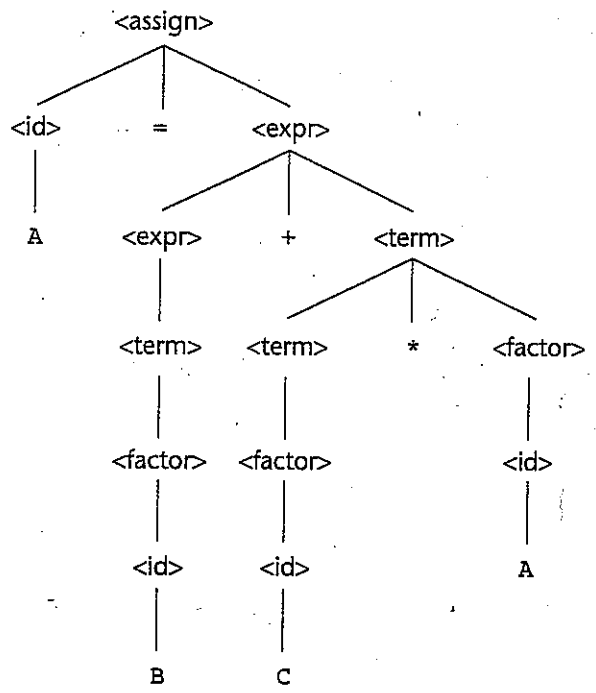
Parse tree
Derivation tree
Syntax tree

Example 3.4 An Unambiguous Grammar for Expressions

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

Operator Precedence Grammar
* > +
left-recursive ⇒ left associativity
Fortran **: F→E**F ⇒ right associativity

A = B + C * A



Left-most derivation

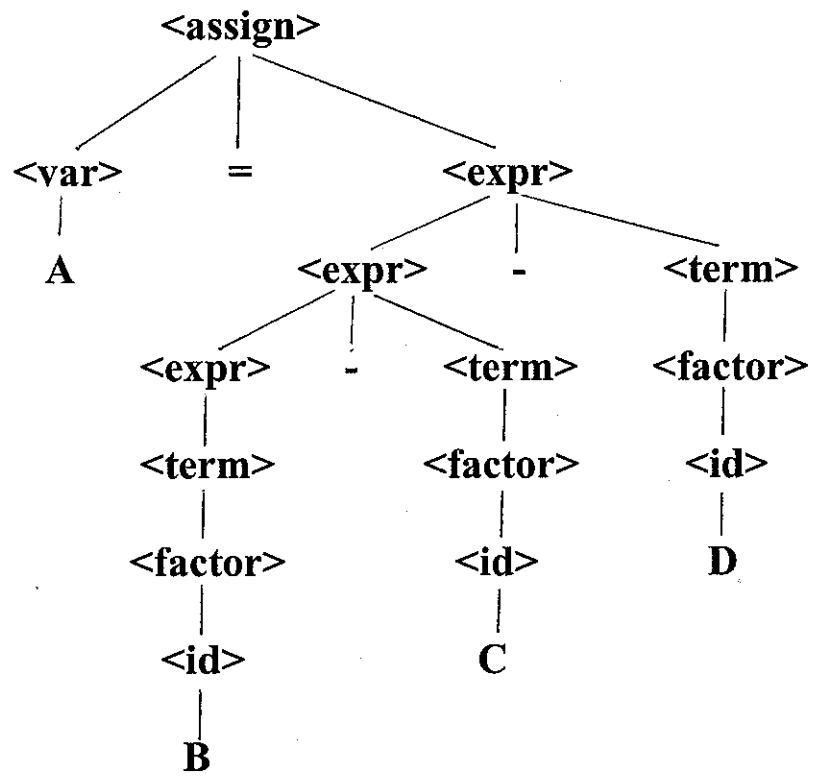
<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> + <term>
=> A = <term> + <term>
=> A = <factor> + <term>
=> A = <id> + <term>
=> A = B + <term>
=> A = B + <term> * <factor>
=> A = B + <factor> * <factor>
=> A = B + <id> * <factor>
=> A = B + C * <factor>
=> A = B + C * <id>
=> A = B + C * A

Right-most derivation

<assign> => <id> = <expr>
=> <id> = <expr> + <term>
=> <id> = <expr> + <term> * <factor>
=> <id> = <expr> + <term> * <id>
=> <id> = <expr> + <term> * A
=> <id> = <expr> + <factor> * A
=> <id> = <expr> + <id> * A
=> <id> = <expr> + C * A
=> <id> = <term> + C * A
=> <id> = <factor> + C * A
=> <id> = <id> + C * A
=> <id> = B + C * A
=> A = B + C * A

Associativity

A = B – C – D



A = (B - C) - D

Syntax and Program Reliability

"dangling-else" problem

if <cond> then S
if <cond> then S_1 else S_2

If S, S_1, or S_2 is another conditional statement, ambiguity may arise.

If <cond1> then if <cond2> then S_1 else S_2

1 if <cond1> then (if <cond2> then S1 else S2)
2 if <cond1> then (if <cond2> then S1) else S2

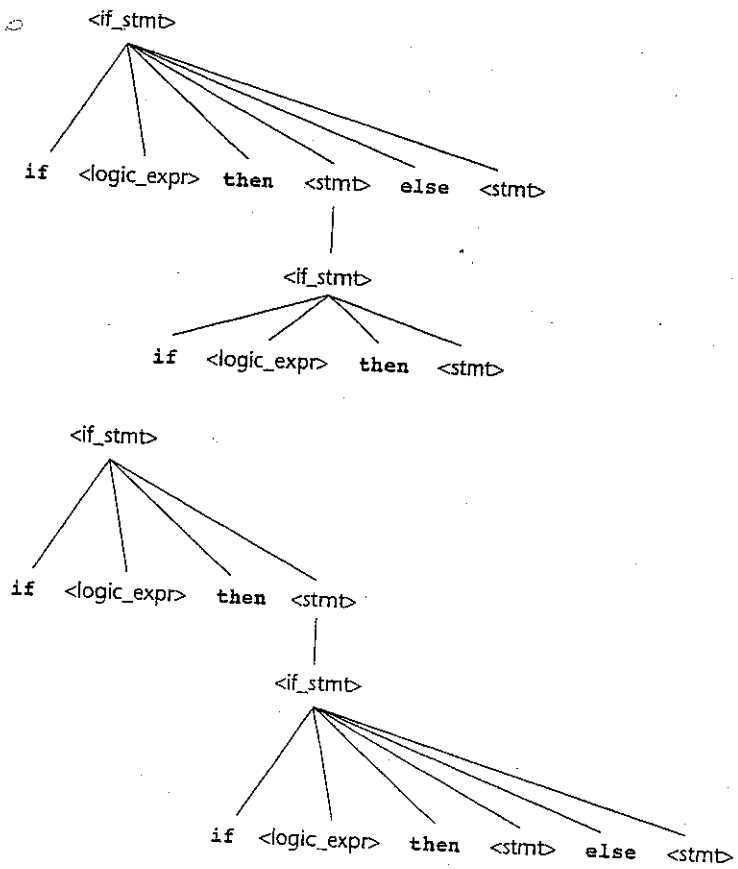


Figure 3.5
Two distinct parse trees for the same sentential form

An Unambiguous Grammar for `if-then-else`

```
<stmt> → <matched> | <unmatched>
<matched> → if <logic_expr> then <matched> else <matched>
          | any non-if statement
<unmatched> → if <logic_expr> then <stmt>
            | if <logic_expr> then <matched> else <unmatched>
```

Solution

Algol60: if <cond1> then begin if <cond2> then S1 end else S2

Algol68: if <cond1> then if <cond2> then S1 fi else S2 fi

Pascal: (else is matched with the innermost if)

if <cond1> then (if <cond2> then S1 else S2)
or if <cond1> then (if <cond2> then S1 else) else S2

Formal Syntax

BNF (Backus-Naur Form)

- mata-language for programming languages
- simple and natural way for context-free grammar
- easy mapping to parser
- Algol 60

- *<non-terminal>*
- ::= → is defined as, is replaced by

Extended BNF (EBNF)

- convenient, but not more powerful

- optional:
  <if-stmt> ::= if (<expr>) then <stmt> [else <stmt>]

- repetition:
  <constant> ::= <digit> { <digit> }
  <series> ::= <statement> { ; <statement> }

  Fortran Id
  <id> ::= <letter> { <alphanum> }$_0^5$

CBL (Cobol-like) Grammar

- **Cobol manual**

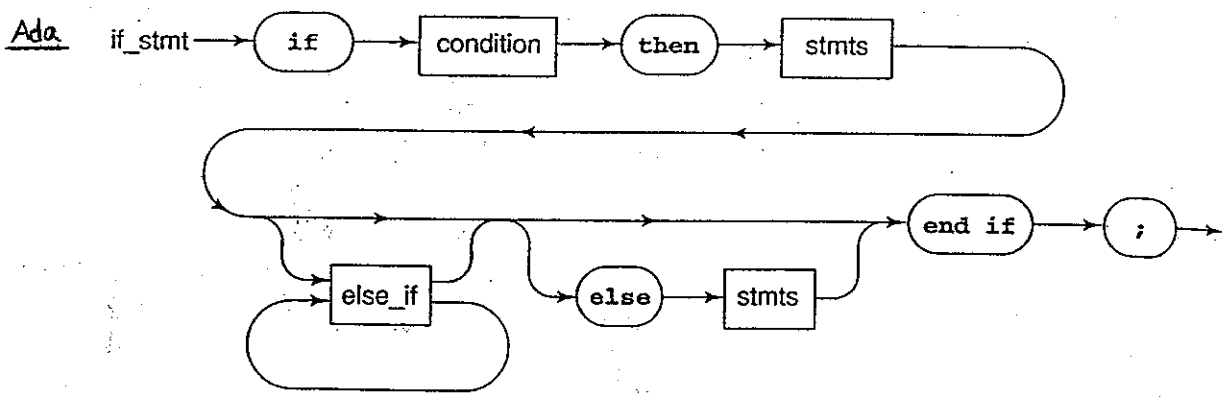
1. **<signed integer> ::= [+ -] <digit>**

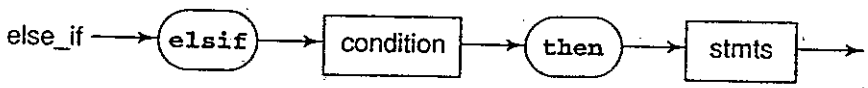
2. **<id> ::= <letter> [<letter> <digit>] ...**

3. **<ADD statement> ::= ADD { <id> <number> } [, <id> <number>] ...**
   TO <id> [ROUNDED] {, <id> [ROUNDED] ...
   [; ON SIZE ERROR <statement>]

Syntax Graph (Syntax Chart) (Bubble Diagram)

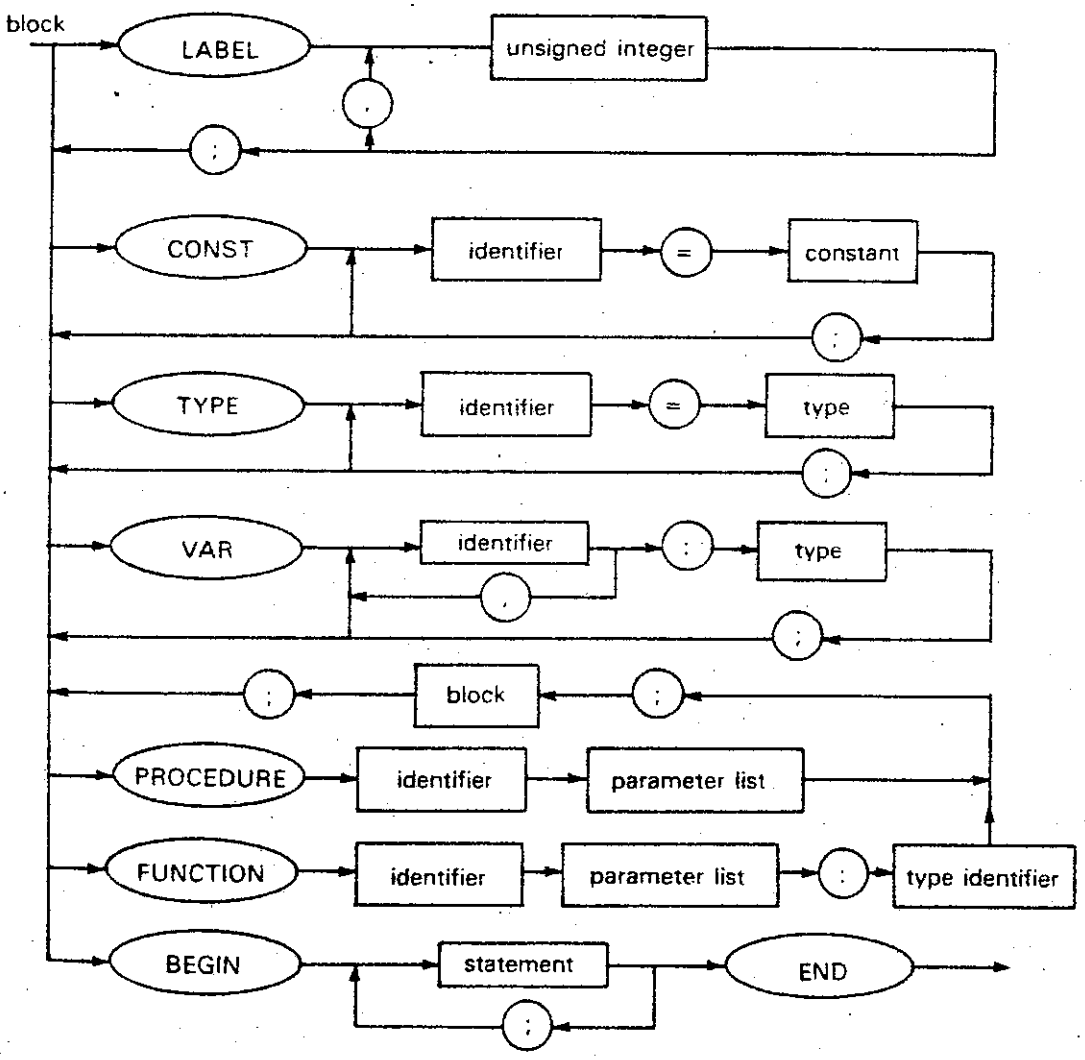
- **Pascal manual (Niklaus Wirth)**





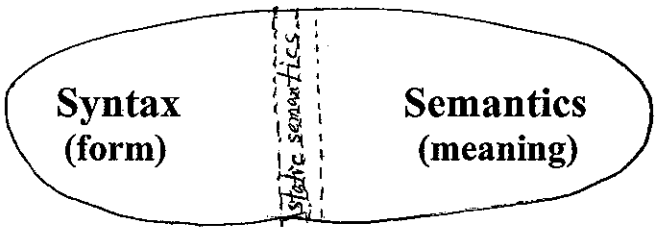
<if_stmt> ⟶ `if` <condition> `then` <stmts> {<else_if>}
[`else` <stmts>] `end if` `;`

<else_if> ⟶ `elsif` <condition> `then` <stmts>



Pascal Syntax Graph for a Block

Syntax and Semantics



Distinction is not clear-cut

Semantics – meaning

1. **Translational approach**
   equivalent program in machine (assembly) language
2. **Interpretive approach**
   set of all pairs of the form (inputfile, outputfile) such that the program produces the output file from the input file

input → [program] → output

o compile-time error

char x
string x

x := y + 1
(type-checking)

execution-time error

out-of-bound case

x := 0 (or read x)
y := 10/x

Static Semantic

Static semantics:

part of semantics that can be determined at compile time

Example.

(1) All variables must be declared before they are referenced.
(2) If the end of an Ada program is followed by a name, that name must match the name of the subprogram.

Attribute Grammars

context-free grammar + α (formal device)

Donald Knuth (1968)

Syntax + static semantics

- **Attribute**
  - **associated with nonterminals**

S(X): **synthesized – bottom-up**, actual_type
I(X): **inherited -- top-down**, expected_type

Intrinsic Attribute
- synthesized attribute of leaf node whose value is determined outside of parse tree. (Ex. symbol table)

- **Production rules**
  - **attribute evaluation functions**
  - **predicate function (static semantics)**

Example

Largest integer: $2^{31} - 1 = 2{,}147{,}483{,}647$

Val : synthesized attribute (bottom-up)

⟨numeral⟩ ::= ⟨digit⟩

Val(⟨numeral⟩) ← Val(⟨digit⟩)

Syntax rule | ⟨numeral⟩$_2$ ⟨digit⟩

Semantic rule Val(⟨numeral⟩) ← 10 × Val(⟨numeral⟩$_2$) + Val(⟨digit⟩)

Predicate Condition: Val(⟨numeral⟩) ≤ 2,147,483,647

⟨digit⟩ ::= 0

Val(⟨digit⟩) ← 0

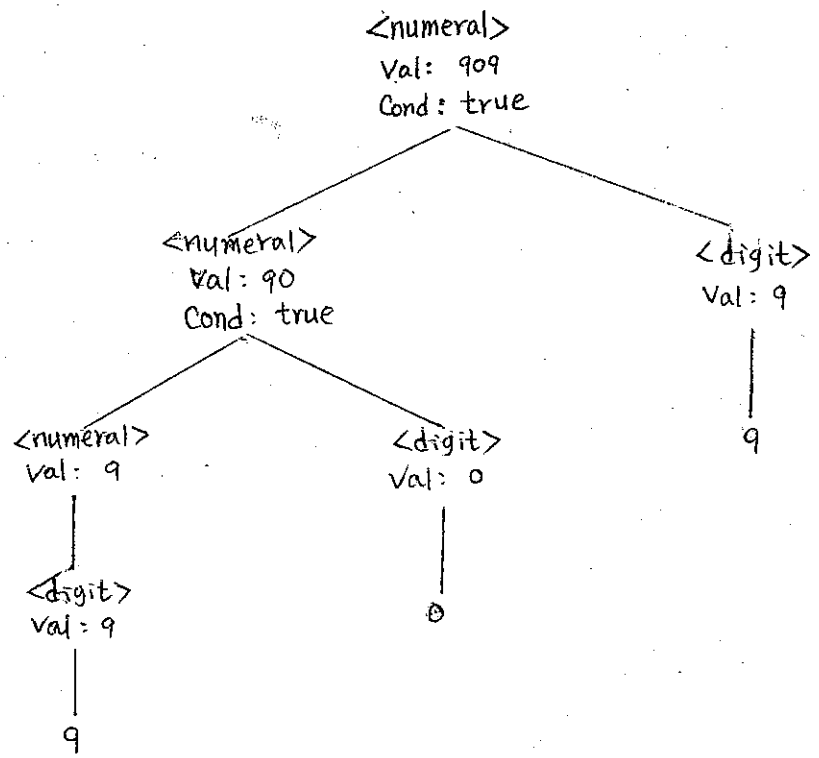
| ...

...

| 9

Val(⟨digit⟩) ← 9

909 : legal integer



Example Ada

<proc_def>::= procedure <proc_name>^{1} <proc_body> end <proc_name>^{2} ;

Predicate: <proc_name>^{1}.string = <proc_name>^{2}.string

Example. Hollerith literals in Fortran II

7 H MONTANA

numeral part	string part

Attribute grammar can check the length of 'MONTANA' is 7

Val - synthesized attribute of <numeral> and <digit>
Size - inherited attribute of the symbol <string>

<numeral> ::= <digit>
Val(<numeral>) ← Val(<digit>

<numeral>::= <numeral>$_2$ <digit>
Val(<numeral>) ← 10 x Val(<numeral>$_2$) + Val(<digit>)

<digit>::= 0 | 1 | ... | 9
Val(<digit>) ← 0 ... Val(<digit>) ← 9

<string>::= <char> | <string>$_2$ <char>
Cond: Size(<string>)=1 Size(<string>$_2$) ← Size(<string>) – 1

<char>::= <digit> | A | B | ... | Z

<literal>::= <numeral> H <string>
Size(<string>) ← Val (<numeral>)
Cond: Val(<numeral>) > 0

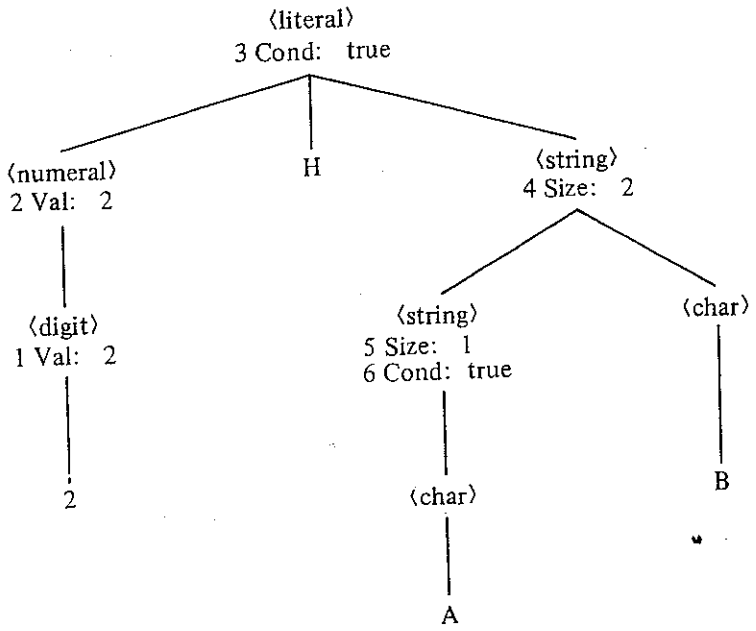


FIGURE 1.

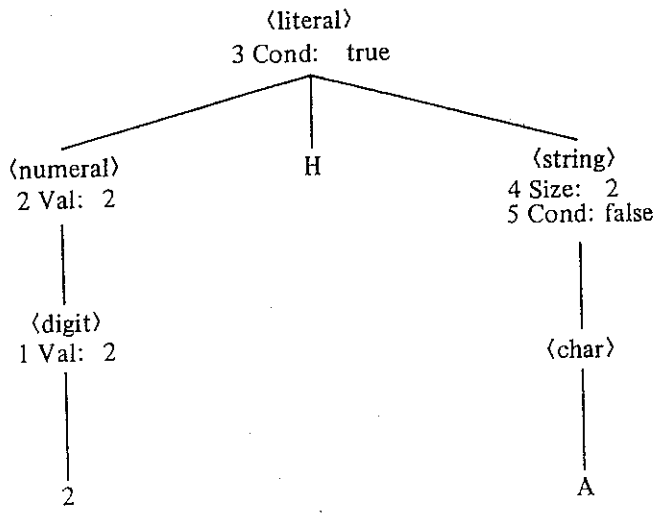


FIGURE 2.

Attribute Grammar for simple assignment statement

Use synthesized attribute *Type*

<assign>::= <var> := <expr>
Cond: Type(<var>) = Type (<expr>)

<expr>::= <var>1 + <var>2
Type(<expr>) ← if (Type(<var>1) = integer) and (Type(<var>2) = integer)
then integer
else real

<expr>::= <var>
Type(<expr>) ← Type(<var>)

<var>::= A | B | C
Type(<var>) ← Type of 'A' from the symbol table
Type(<var>) ← Type of 'B' from the symbol table
Type(<var>) ← Type of 'C' from the symbol table

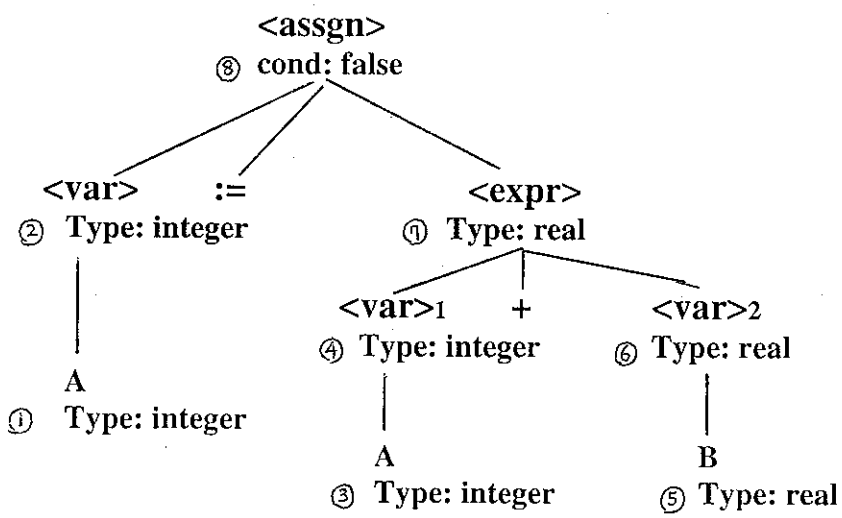
Ex. A:= A + B

Symbol Table

Intrinsic Attribute

A	integer
B	real

HW A: real
B: integer



Attribute Grammar for Context-Sensitive Language

$$L = \{ x^n y^n z^n : n \geq 1 \}$$

Idea – Find the size of x string (bottom-up). Then check whether y and z string have the same size (top-down).

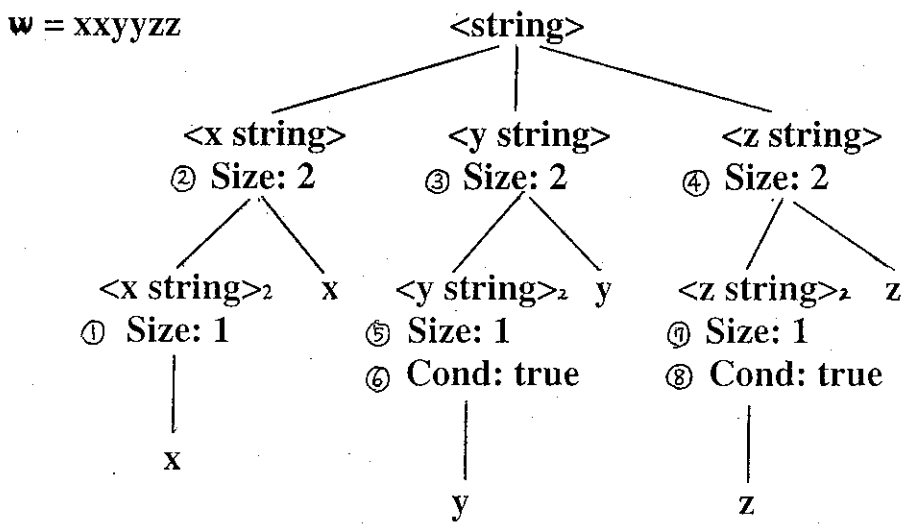
Size is synthesized attribute for x string, but inherited attribute for y string and z string.

⟨string⟩ ::= ⟨x string⟩ ⟨y string⟩ ⟨z string⟩
  cond: Size(⟨y string⟩) ⟵ Size(⟨x string⟩)
  Size(⟨z string⟩) ⟵ Size(⟨x string⟩)

⟨x string⟩ ::= *x*
  Size(⟨x string⟩) ⟵ 1
 | ⟨x string⟩$_2$ *x*
  Size(⟨x string⟩) ⟵ Size(⟨x string⟩$_2$) + 1

⟨y string⟩ ::= *y*
  Condition: Size(⟨y string⟩) = 1
 | ⟨y string⟩$_2$ *y*
  Size(⟨y string⟩$_2$) ⟵ Size(⟨y string⟩) − 1

⟨z string⟩ ::= *z*
  Condition: Size(⟨z string⟩) = 1
 | ⟨z string⟩$_2$ *z*
  Size(⟨z string⟩$_2$) ⟵ Size(⟨z string⟩) − 1



Context Sensitive Language (Type 1 language)

Ex 1. $L = \{ a^n b^n c^n \mid n \geq 1 \}$

$S \rightarrow aSBc \mid aBc$

$aB \rightarrow ab$

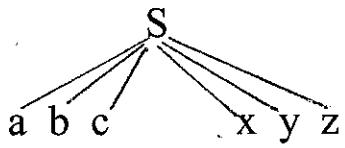
$cB \rightarrow Bc$

$bB \rightarrow bb$

Ex 2. $L = \{wcw \mid w \in (a,b)^+\}$

Context:

- define variables before using them
- argument-parameter matching



argument parameter

Dynamic Semantics

- no universally accepted notation

Operational	- TM, state	: language implementer	↓ abstract
Denotational	- state	: language designer	
Axiomatic	- function	: language users	

Operational Semantics

- based on machine and algorithm

Idea: meaning of a statement ≡ change of machine states caused by the execution of the instruction

VDL (Vienna Definition Language)

- interpret program (i.e., sequence of actions specified by program)

  abstract machine
  abstract syntax
  instruction definitions – with control tree, describe state transitions

- semantics of PL/I (1972)
- so complex, serves no practical purpose

Ex. c for-loop

```
for (expr1; expr2; expr3 ) {
        :
}
```

high-level

interpretation

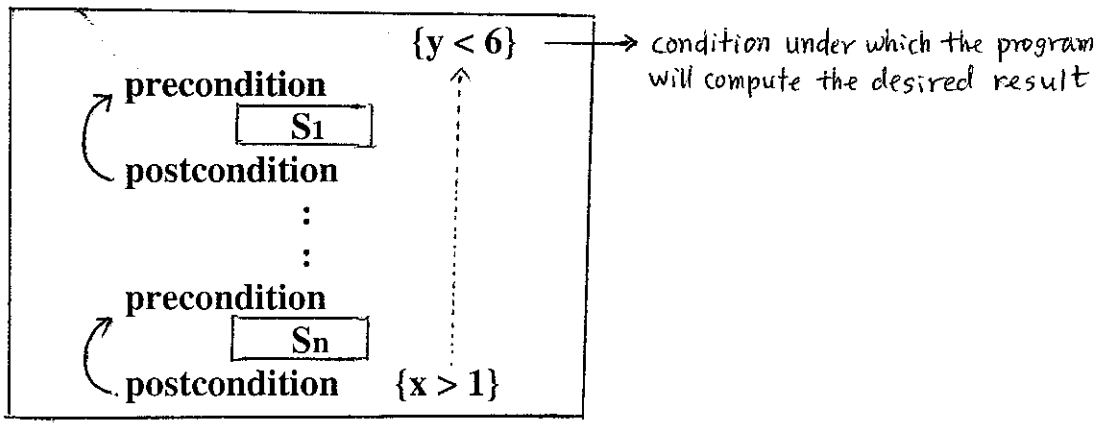
```
      expr1;
loop: if expr2 = 0 goto out
        ...
      expr3;
      goto loop
out:
```

low-level

Axiomatic Semantics

- abstract
- based on mathematical logic (predicate calculus) (assertion)
- correctness proof of a program

Idea: use variables to specify the meaning of the statement
i.e., how the value of a variable changes before and after a statement



Axiom - logical statement that is assumed to be true
Inference rule – method of inferring the truth of an assertion on the basis of values of other assertions

Theorem: {P} S {Q} (axiomatic semantics)
If P is true before the execution of S and if execution of S terminates, *then Q is true after execution of S*

Weakest precondition:
Ex. sum = 2 * X + 1 {sum > 1}
{x > 10} {x > 100} ... {x > 0}
↑ weakest precondition

(0) Dummy statement

Fortran 'CONTINUE'

{P} CONTINUE {P}

$\{P\}\ S\ \{Q\}$

(1) Assignment Statements

$$P = Q_{x \to E}$$
$$\boxed{X := E}$$
$$Q$$

$\{Q_{x \to E}\}\ x := E\ \{Q\}$

Ex. $a = b / 2 - 1$ $\underline{\{a < 10\}}$ Q

P: $\{b / 2 - 1 < 10\}$
$\{b < 22\}$

Proof
$\{x > 3\}\ x = x - 3\ \{x > 0\}$
use assignment axiom on $x = x-3\ \{x>0\} \Rightarrow \{x>3\}$, given precondition
what if
$\{x > 5\}\ x = x - 3\ \{x > 0\}$

rule of consequence(inference): $\underline{S_1, S_2, \ldots S_n}$
(deduction rule) S

If S1, S2, … are true, then the truth of S can be inferred.

$$\frac{\{P\}\ S\ \{Q\},\ P' \overset{\text{implies}}{\to} P,\ Q \to Q'}{\{P'\}\ S\ \{Q'\}}$$

Note: precondition can be strengthened
postcondition can be weakened

Ex. $P\ \{x > 3\}, P'\ \{x > 5\}, Q\ \{x > 0\}, Q'\ \{x > 0\}$

$$\frac{\{x > 3\}\ x = x - 3\ \{x > 0\},\ \{x > 5\} \to \{x > 3\},\ \{x > 0\} \to \{x > 0\}}{\{x > 5\}\ x = x - 3\ \{x > 0\}}$$

x := 2 * y - 3
{x > 25}

compute the weakest precondition

2 * y - 3 > 25

2 * y > 25 + 3 = 28

y > 28/2 = 14

this yields the following group

{y > 14}

x := 2 * y - 3

{x > 25}

(2) Sequence

$\{P1\}\ S1\ \{P2\}$
$\{P2\}\ S2\ \{P3\}$

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

If $S1:\ X1 = E1$
$S2:\ X2 = E2$
then
$\{P3_{X2 \rightarrow E2}\}\ X2 = E2\ \{P3\}$
$\{(P3_{X2 \rightarrow E2})_{X1 \rightarrow E1}\}\ X1 = E1\ \{P3_{X2 \rightarrow E2}\}$
↑ weakest precondition for $S1; S2$

Ex.
```
y := 3 * x + 1;
x := y + 3
{x < 10}
```

compute weakest precondition for 2nd statement

$y + 3 < 10$
$y < 7$

using {y < 7} as post condition for 1st statement compute the weakest precondition for the 1st statement

$3 * x + 1 < 7$
$3 * x < 6$
$x < 2$

(3) Selection

Inference rule

$$\frac{\{B \text{ and } P\}\ S_1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S_2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } S_1 \text{ else } S_2\ \{Q\}}$$

Ex.

suppose

if (x > 0) then y = y – 1 else y = y + 1 {y > 0}

then: {y > 1} y = y – 1 {y > 0}
else: {y > -1} y = y + 1 {y > 0}

$\{y > 1\} \Rightarrow \{y > -1\}$

$\therefore P = \{y > 1\}$

(4) Logical Pretest (while loop)

- loop invariant I

$$\frac{(I \text{ and } B)\ S\ \{I\}}{\underbrace{\{I\}}_{P}\ \text{while B do S end}\ \underbrace{\{I \text{ and (not B)}\}}_{Q}}$$

Requirements:

1. P ➔ I
2. {I and B} S {I}
3. (I and (not B)) ➔ Q
4. the loop terminates

Define function wp that produces weakest precondition.

wp (statement, postcondition) = precondition

Example. while $y <> x$ do $y = y + 1$ end $\{y = x\}$

0 iteration: $wp = \{y = x\}$
1 iteration: $wp\ (y = y + 1, \{y = x\}) = \{y = x - 1\}$
2 iteration: $wp\ (y = y + 1, \{y = x-1\}) = \{y = x - 2\}$
: : :

} I: $\{y \leq x\}$

$P = I$

1. $P = I$, therefore $P \rightarrow I$

2. Show $\{y \leq x \text{ and } y <> x\}\ y = y + 1\ \{y \leq x\}$
   $y = y + 1\ \{y \leq x\}$
   $\{y + 1 \leq x\}$
   $\{y < x\}$
   $\{y \leq x \text{ and } y <> x\}$

3. Show $\{(y \leq x) \text{ and not } (y <> x)\} \rightarrow \{y = x\}$
   $\{(y \leq x) \text{ and } (y = x)\} \rightarrow \{y = x\}$
   $\{y = x\} \rightarrow \{y = x\}$

4. Termination of $\{(y \leq x)\}$ while $y <> x$ do $y = y+1$ end $\{y=x\}$
   Case $y = x$: no iteration, $y = x$
   Case $y < x$: eventually y value becomes x, $y = x$
   } terminate

Denotational Semantics

① Define object
② Find mapping function

- Scott and Strachey (1971)
- Recursive function theory
- Denoted object → meaning

Denotational semantics define mathematical object and a function that maps instances of that entity onto instances of the mathematical object.

Ex. <binary> → 0 | 1 | <binary> 0 | <binary> 1

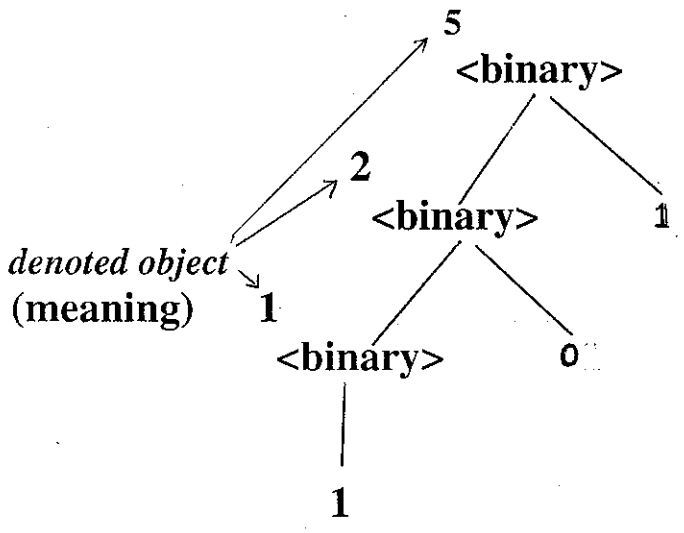
Let the domain of semantic values of the objects be N.
Semantic function Mbin maps syntactic objects to N.

semantic function ↓ ↓ syntactic digit

$M_{bin}(\text{'0'}) = 0, \quad M_{bin}(\text{'1'}) = 1$

$M_{bin}(\langle binary\rangle\ \text{'0'}) = 2 * M_{bin}(\langle binary\rangle)$

$M_{bin}(\langle binary\rangle\ \text{'1'}) = 2 * M_{bin}(\langle binary\rangle) + 1$



syntax-directed semantics

semantics of a program ≡ state changes of a computer

note: operational: coded algorithms
denotational: mathematical functions

state of a program : $\langle i_1, v_1 \rangle, \langle i_2, v_2 \rangle, \ldots \langle i_n, v_n \rangle$

(i_1: variable, v_1: value)

mapping function $VARMAP(i_j, s) = v_j$

// at state s, variable i_j has value v_j //

undef is a special value.

Expressions

<expr> → <dec_num> | <var> | <binary_expr>
<binary_expr> → <left_expr> <operator> <right_expr>
<left_expr> → <dec_num> | <var>
<right_expr> → <dec_num> | <var>
<operator> → + | *

semantic function of expression

M_e(<expr>, s) Δ=
case <expr> of
<dec_num> => M_{dec}(<dec_num>, s)
<var> => if VARMAP(<var>, s) == **undef**
then **error**
else VARMAP(<var>, s)
<binary_expr> =>
if (M_e(<binary_expr>.<left_expr>, s) == **undef** OR
M_e(<binary_expr>.<right_expr>, s) == **undef**)
then **error**
else if (<binary_expr>.<operator> == '+') then
M_e(<binary_expr>.<left_expr>, s) +
M_e(<binary_expr>.<right_expr>, s)
else M_e(<binary_expr>.<left_expr>, s) *
M_e(<binary_expr>.<right_expr>, s)

a + b
= left right

Assignment Statements

if rhs expression is error, ... or = error / b+c

$M_a(x = E, s) \Delta=$ if $M_e(E, s) ==$ **error**
then **error**
else $s' = \{<i_1', v_1'>, <i_2', v_2'>, \ldots, <i_n', v_n'>\}$, where
for $j = 1, 2, \ldots, n$
if $i_j ==$ (x) ← LHS var
then $v_j' = M_e(E, s)$ // x is replaced by expr value
else $v_j' = VARMAP(i_j, s)$ // same values

Logical Pretest Loops (while loop)

$M_l($**while** B **do** L, s$) \Delta=$ if $M_b(B, s) ==$ **undef**
then **error**
else if $M_b(B, s) ==$ false
then s ——— no state change
else if $M_{sl}(L, s) ==$ **error**
then **error**
else $M_l($**while** B **do** L, $M_{sl}(L, s))$ // recursion

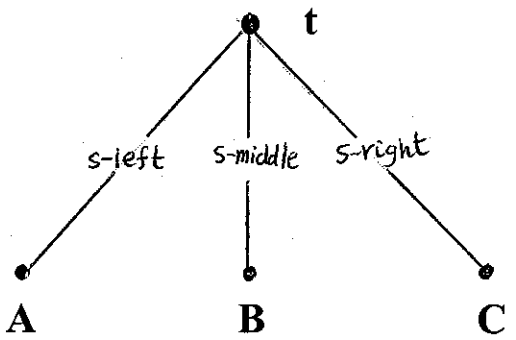
M_{sl} ≡ statement list to states
M_b ≡ Boolean expression to Boolean values

Vienna Definition Language: (Operational approach)

- **Elementary object: A, B, C**
- **Structured object: <s-: a>** **// selector : object //**

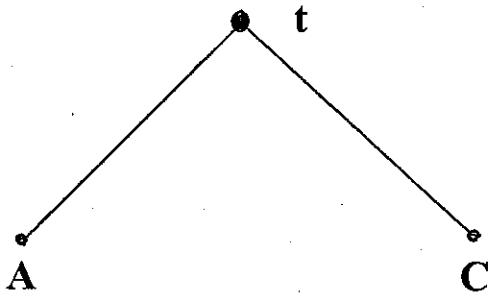
selector object

Ex. t: {<s-left: A>, <s-middle: B>, <s-right: C>}

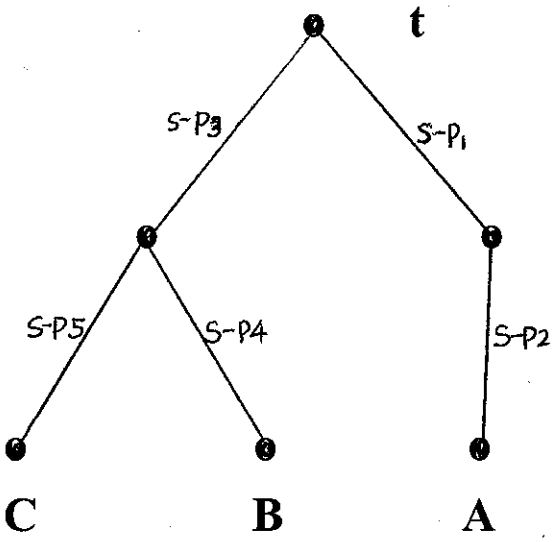


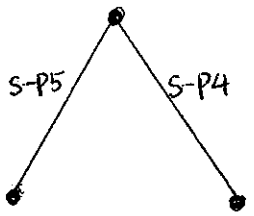
- **Null object : Ω**

{<s-left: A>, <s-middle: Ω >, <s-right: C>}



Ex.



① **s-p3(t)** → 

② **s-p5 ((s-p3(t))** → **C**

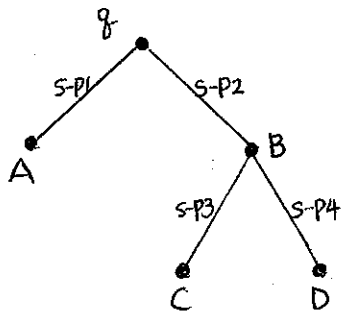
or s-p5 o s-p3(t)

③ **s-p6(t)** → Ω

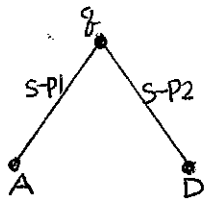
- **Mutation operator:** μ

μ (a; <s : b>)

Ex.

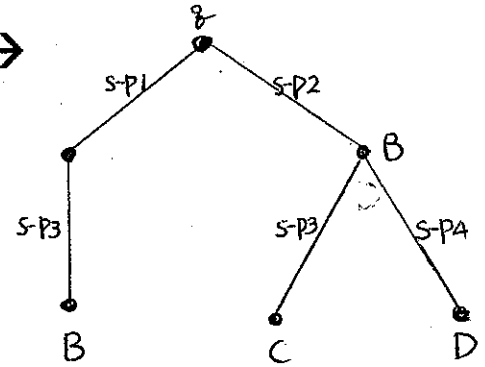


① μ(q; <s-p2: D>) →



② μ(q; <s-p2: Ω>) →

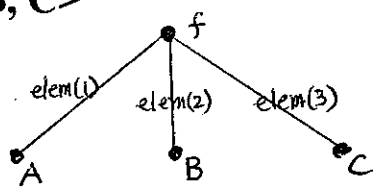
③ μ(q; <s-p1: {<sp3: B}>) →



Note. two S-P3.

- **List**

  f = {<elem(1): A>, <elem(2): B>, <elem(3): C>}
  or <A, B, C>



  - **length(f) → 3**
  - **head(f) → A**
  - **tail(f) → {<elem(1): B>, <elem(2): C>} or <B, C>**

- **Concatenation (⌢)**

  f ⌢ tail → <A, B, C, B, C>

- **Predicate (true or false)**

is-asmt-st = (<s-lhs: is-var>, <s-rhs: <u>is-expr</u>>) //conjunction
<u>is-expr</u> = is-infix-expr ∨ is-var ∨ is-intg //disjunction

⇓

is-asmt-st = (<s-lhs: is-var>,<s-rhs: is-infix-expr ∨ is-var ∨ is-intg>)

VDL

Abstract machine

state: ξ

is-ξ = (<s-c: is-c>,<is-stg: is-value-list>,
<s-input: is-intg-list> <is-output: is-intg-list>)

↑ input file ↑ output file

is-value = is-intg ∨ is-UNDEFINED

Ex.

elem(i) o s-stg(ξ) ≡ current value of the i-th variable

Abstract Syntax for Pam

is-series = is-st-list

is-st = is-read-st ∨ is-write-st ∨ is-asmt-st ∨ is-cond-st ∨
    is-def-loop ∨ is-indef-loop

is-read-st = (⟨s-r : is-var-list⟩)

is-write-st = (⟨s-w : is-var-list⟩)

is-asmt-st = (⟨s-lhs : is-var⟩, ⟨s-rhs : is-expr⟩)

is-cond-st = (⟨s-ifpart : is-comp⟩, ⟨s-thenpart : is-series⟩,
    ⟨s-elsepart : is-series⟩)

is-def-loop = (⟨s-limit : is-expr⟩, ⟨s-body : is-series⟩)

is-indef-loop = (⟨s-test : is-comp⟩, ⟨s-body : is-series⟩)

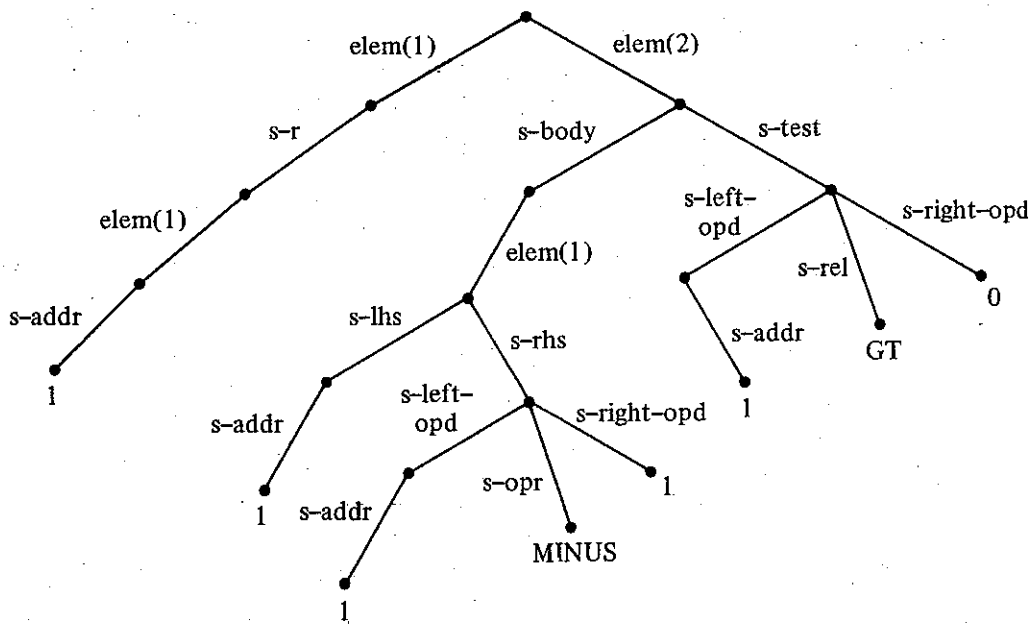
is-comp = (⟨s-left-opd : is-expr⟩, ⟨s-right-opd : is-expr⟩,
    ⟨s-rel : is-EQ ∨ is-GT ∨ is-LE ∨ is-LT ∨ is-GE ∨ is-NE⟩)

is-expr = is-infix-expr ∨ is-var ∨ is-intg

is-infix-expr = (⟨s-left-opd : is-expr⟩, ⟨s-right-opd : is-expr⟩,
    ⟨s-opr : is-PLUS ∨ is-MINUS ∨ is-TIMES ∨ is-OVER⟩)

is-var = (⟨s-addr : is-intg⟩)

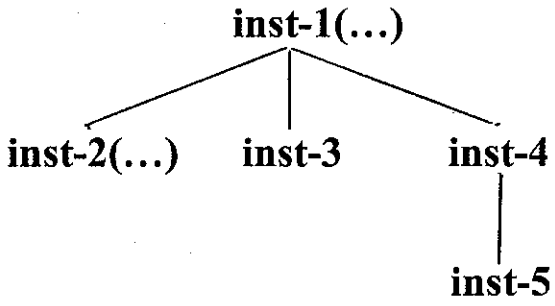
Example. **read** k ; **while** $k > 0$ **do** $k := k - 1$ **end**



Control Mechanism and Notation for Instruction Definitions – A Semantic Specification for Pam

s-c (ξ)

Control tree:



instruction definition:

inst-1 (a, b) =

Condition$_1$ → Group$_1$
Condition$_2$ → Group$_2$
: :
Condition$_n$ → Group$_n$

inst-3 =

Condition$_1$ → Group$_1$
Condition$_2$ → Group$_2$
: :
Condition$_n$ → Group$_n$

Group

- **value-returning group**

  PASS: Expression or

  Selector: Expression

  Ex. PASS: head o s-input(ξ)

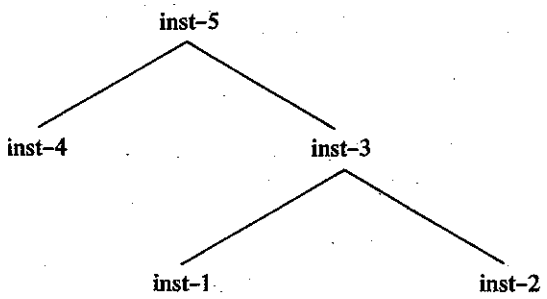
  s-input: tail o s-input(ξ)

  // 1st element of the 'input'-component of the state is passed

  // elsewhere and deleted from the 'input'-component

  Ex. null = PASS: // delete itself

- **self-replacing group**




```
inst-5;
    inst-4,
    inst-3;
        inst-1,
        inst-2
```

```
inst-5;
    inst-3;
        inst-2,
        inst-1
    inst-4
```

Ex.

inst-2 (…, a, …)

a: inst –1 (…)

// argument a is supplied as a result of the inst-1

TABLE 4.1 VDL Definition of the Semantics of Pam

Abstract Machine

is-ξ = (⟨s-c : is-c⟩, ⟨s-stg : is-value-list⟩,
⟨s-input : is-intg-list⟩, ⟨s-output : is-intg-list⟩)
is-value = is-intg ∨ is-UNDEFINED

Abstract Syntax

is-series = is-st-list
is-st = is-read-st ∨ is-write-st ∨ is-asmt-st ∨ is-cond-st ∨
is-def-loop ∨ is-indef-loop
is-read-st = (⟨s-r : is-var-list⟩)
is-write-st = (⟨s-w : is-var-list⟩)
is-asmt-st = (⟨s-lhs : is-var⟩, ⟨s-rhs : is-expr⟩)
is-cond-st = (⟨s-ifpart : is-comp⟩, ⟨s-thenpart : is-series⟩,
⟨s-elsepart : is-series⟩)
is-def-loop = (⟨s-limit : is-expr⟩, ⟨s-body : is-series⟩)
is-indef-loop = (⟨s-test : is-comp⟩, ⟨s-body : is-series⟩)
is-comp = (⟨s-left-opd : is-expr⟩, ⟨s-right-opd : is-expr⟩,
⟨s-rel : is-EQ ∨ is-GT ∨ is-LE ∨ is-LT ∨ is-GE ∨ is-NE⟩)
is-expr = is-infix-expr ∨ is-var ∨ is-intg
is-infix-expr = (⟨s-left-opd : is-expr⟩, ⟨s-right-opd : is-expr⟩,
⟨s-opr : is-PLUS ∨ is-MINUS ∨ is-TIMES ∨ is-OVER⟩)
is-var = (⟨s-addr : is-intg⟩)

Instruction Definitions

exec-series (ser) =
is-⟨⟩ (ser) ⟶ **null**
T ⟶ **exec-series** (tail (ser));
exec-stmt (head (ser))

```
exec-stmt (stmt) =
    is-read-st (stmt) ⟶ read (s-r (stmt))
    is-write-st (stmt) ⟶ write (s-w (stmt))
    is-asmt-st (stmt) ⟶
        store (s-addr ∘ s-lhs (stmt), val);
            val: eval-expr (s-rhs (stmt))
    is-cond-st (stmt) ⟶
        choose (c, s-thenpart (stmt), s-elsepart (stmt));
            c: eval-comp (s-ifpart (stmt))
    is-def-loop (stmt) ⟶
        repeat-series (s-body (stmt), r);
            r: eval-expr (s-limit (stmt))
    is-indef-loop (stmt) ⟶
        choose (c, s-body (stmt)⌢⟨stmt⟩, ⟨⟩);
            c: eval-comp (s-test (stmt))

repeat-series (ser, n) =
    n ≤ 0 ⟶ null
    T ⟶ repeat-series (ser, n − 1);
            exec-series (ser)

choose (flag, ser-1, ser-2) =
    flag ⟶ exec-series (ser-1)
    T ⟶ exec-series (ser-2)

eval-comp (comp) =
    compare (a, b, s-rel (comp));
        a: eval-expr (s-left-opd (comp)),
        b: eval-expr (s-right-opd (comp))

compare (val-1, val-2, rel) =
    is-EQ (rel) ∧ (val-1 = val-2) ∨
        is-GT (rel) ∧ (val-1 > val-2) ∨
        is-LE (rel) ∧ (val-1 ≤ val-2) ∨
        is-LT (rel) ∧ (val-1 < val-2) ∨
        is-GE (rel) ∧ (val-1 ≥ val-2) ∨
        is-NE (rel) ∧ (val-1 ≠ val-2) ⟶ PASS: T
    T ⟶ PASS: F
```

```
eval-expr (expr) =
    is-intg (expr) ⟶ PASS: expr
    is-var (expr) ⟶ eval-var (expr)
    is-infix-expr (expr) ⟶
        calculate (a, b, s-opr (expr));
            a: eval-expr (s-left-opd (expr)),
            b: eval-expr (s-right-opd (expr))

eval-var (var) =
    is-UNDEFINED (elem (s-addr (var)) (s-stg (ξ))) ⟶ error
    T ⟶ PASS: elem (s-addr (var)) (s-stg (ξ))

calculate (val-1, val-2, opr) =
    is-PLUS (opr) ⟶ PASS: val-1 + val-2
    is-MINUS (opr) ⟶ PASS: val-1 − val-2
    is-TIMES (opr) ⟶ PASS: val-1 × val-2
    is-OVER (opr) ⟶ PASS: val-1 ÷ val-2

store (loc, val) =
    s-stg: μ(s-stg(ξ); ⟨elem (loc) : val⟩)

read (vars) =
    is-⟨⟩ (vars) ⟶ null
    T ⟶ read (tail (vars));
            store (s-addr ∘ head (vars), val);
                val: input-val

input-val =
    is-⟨⟩ ∘ s-input (ξ) ⟶ error
    T ⟶ PASS: head ∘ s-input (ξ)
        s-input: tail ∘ s-input (ξ)

write (vars) =
    is-⟨⟩ (vars) ⟶ null
    T ⟶ write (tail (vars));
            output-val (val);
                val: eval-expr (head (vars))

output-val (val) =
    s-output: s-output(ξ)⌢val
```

Control example: [read k; while k > 0 do k:= k – 1 end]

Assume k is 1.

1 • **exec-series** (t)

2 • **exec-series** ([**while** $k > 0$ **do** $k := k - 1$ **end**])
• **exec-stmt** ([**read** k])

3 • **exec-series** ([**while** $k > 0$ **do** $k := k - 1$ **end**])
• **read** ([k])

4 • **exec-series** ([**while** $k > 0$ **do** $k := k - 1$ **end**])
• **read** ($\diamond$)
• **store** (1, val)
• val: **input-val**

5 • **exec-series** ([**while** $k > 0$ **do** $k := k - 1$ **end**])
• **read** ($\diamond$)
• **store** (1, 1)

// k = 1

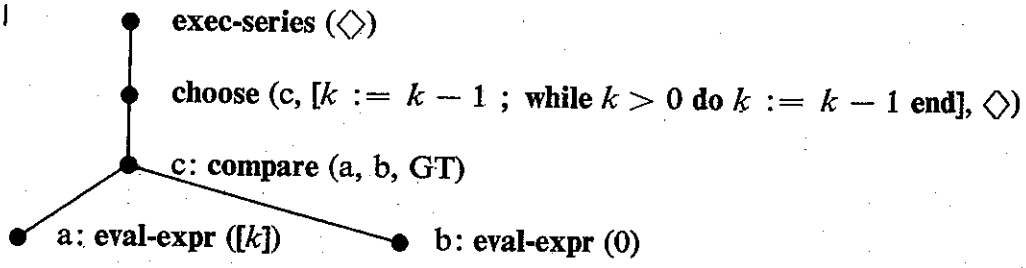
6 • **exec-series** ([**while** $k > 0$ **do** $k := k - 1$ **end**])
• **read** ($\diamond$)

7 • **exec-series** ([**while** $k > 0$ **do** $k := k - 1$ **end**])
• **null**

8 • **exec-series** ([**while** $k > 0$ **do** $k := k - 1$ **end**])

9 • **exec-series** ($\diamond$)
• **exec-stmt** ([**while** $k > 0$ **do** $k := k - 1$ **end**])

10 • **exec-series** ($\diamond$)
• **choose** (c, [$k := k - 1$; **while** $k > 0$ **do** $k := k - 1$ **end**], $\diamond$)
• c: **eval-comp** ([$k > 0$])

11 

12 • **exec-series** ($\diamond$)
• **choose** (c, [$k := k - 1$;
while $k > 0$ **do** $k := k - 1$ **end**], $\diamond$)
• c: **compare** (a, 0, GT)
• a: **eval-expr** ([k])

13 • **exec-series** ($\diamond$)
• **choose** (c, [$k := k - 1$;
while $k > 0$ **do** $k := k - 1$ **end**], $\diamond$)
• c: **compare** (a, 0, GT)
• a: **eval-var** ([k])

// K = 1

14 **exec-series** (◇)
choose (c, [k := $k - 1$;
while $k > 0$ **do** k := $k - 1$ **end**], ◇)
c: **compare** (t, 0, GT) // $1 > 0 \Rightarrow$ T

15 **exec-series** (◇)
choose (T, [k := $k - 1$; // Choose T
while $k > 0$ **do** k := $k - 1$ **end**], ◇)

16 **exec-series** (◇)
exec-series ([k := $k - 1$; **while** $k > 0$ **do** k := $k - 1$ **end**])

17 **exec-series** (◇)
exec-series ([**while** $k > 0$ **do** k := $k - 1$ **end**])
exec-stmt ([k := $k - 1$])

18 **exec-series** (◇)
exec-series ([**while** $k > 0$ **do** k := $k - 1$ **end**])
store (1, val)
val: **eval-expr** ([$k - 1$])

19 **exec-series** (◇)
exec-series ([**while** $k > 0$ **do** k := $k - 1$ **end**])
store (1, val)
val: **calculate** (a, b, MINUS)
a: **eval-expr** ([k]) b: **eval-expr** (1)

20
- exec-series (⟨⟩)
- exec-series ([while K > 0 do K := K - 1 end])
- store (1, val)
- val: calculate (a, 1, MINUS)
- a: eval-expr ([K])

21
- exec-series (⟨⟩)
- exec-series ([while K > 0 do K := K - 1 end])
- store (1, val)
- val: calculate (a, 1, MINUS)
- a: eval-var ([K])

22
- **exec-series** (⟨⟩)
- **exec-series** ([**while** $k > 0$ **do** $k := k - 1$ **end**])
- **store** (1, val)
- **val: calculate** (1, 1, MINUS) // 1-1 ⇒ 0

23
- exec-series (⟨⟩)
- exec-series ([while K > 0 do K := K - 1 end])
- store (1, 0) // K := 0

24
- **exec-series** (⟨⟩)
- **exec-series** ([**while** $k > 0$ **do** $k := k - 1$ **end**])

25
- **exec-series** (⟨⟩)
- **exec-series** (⟨⟩)
- **exec-stmt** ([**while** $k > 0$ **do** $k := k - 1$ **end**])

26

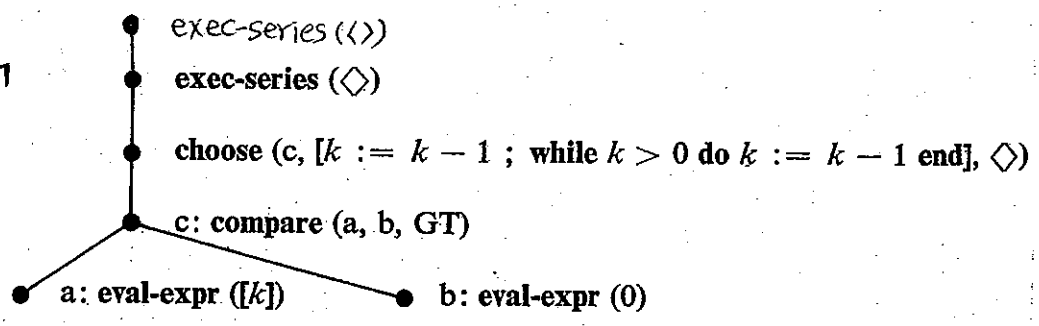
exec-series (⟨⟩)

exec-series (◇)

choose (c, [$k := k - 1$; **while** $k > 0$ **do** $k := k - 1$ **end**], ◇)

c: **eval-comp** ([$k > 0$])

27



28

exec-series (⟨⟩)

exec-series (◇)

choose (c, [$k := k - 1$;
while $k > 0$ **do** $k := k - 1$ **end**], ◇)

c: **compare** (a, 0, GT)

a: **eval-expr** ([k])

29

exec-series (⟨⟩)

exec-series (◇)

choose (c, [$k := k - 1$;
while $k > 0$ **do** $k := k - 1$ **end**], ◇)

c: **compare** (a, 0, GT)

a: **eval-var** ([k])

// current k value is 0.

30

exec-series (⟨⟩)

exec-series (◇)

choose (c, [$k := k - 1$;
while $k > 0$ **do** $k := k - 1$ **end**], ◇)

c: **compare** (0, 0, GT)

// 0 is not GT 0 ⇒ choose F

31

exec-series (⟨⟩)

exec-series (⟨⟩)

exec-series (⟨⟩)

⋮