3 Describing Syntax and Semantics

3.1 Introduction

- **syntax –** expressions, statements, program units
- **semantics** – meaning

3.2 The General Problem of Describing Syntax

- alphabet
- strings from the alphabet
- sentences composed of strings
- **lexemes** or **tokens** Lexemes are described separately from the grammar of the language. Lexemes are identifiers, literals, punctuation
- **tokens** are represented as a pair
    - o the string recognized
    - o a unique integer code often expressed symbolically as an enumeration constant

Example

statement index = 2 * count + 17;

Lexeme	*Token*(*integer code*)
index	ID(1)
=	ASSIGN(2)
2	INTLIT(3)
*	STAR(4)
count	ID(1)
+	PLUS(5)
17	INTLIT(3)
;	SEMICOLON(6)

3.2.1 Language Recognizers

- Language can be formally defined in two distinct ways:
    1) recognition
    2) generation

3.2.2 Language Generators

- A Language Generator is a device that can be used to generate sentences in a language. Recall that a sentence in a programming language is a *program*.
- Perhaps a language generator could be an aid to testing an implementation of a language but has little use in other areas.
- A language generator is interesting because programming language grammars can be used both to recognize and generate sentences in the grammar of a programming language.

3.3 Formal Methods of Describing Syntax

3.3.1 Backus-Naur Form and Context-Free Grammars

3.3.1.1 Context-free Grammars

- Chomsky defined four classes of grammars. Two of these classes are context-free and regular.
- Tokens can be described by regular expressions
- The syntax of most programming languages can be described by context-free grammars.

3.3.1.2 Origins of Backus-Naur Form

- ALGOL 58
- ACM-GAMM Conference 1959 John Backus
- Later modified by Peter Naur. Backus-Naur Form

3.3.1.3 Fundamentals

Metalanguage

A metalanguage is employed to describe the grammar of a programming language. A metalanguage contains four attributes as described below.

Context-free grammars

1. Terminals are the basic symbols from which strings are formed. The token is a synonym for "terminal" when we are talking about grammars for programming languages. Terminal strings are presented in bold. Terminal strings include reserve words **if**, **then**, **else**, **while**.
2. Nonterminals are syntactic variables the denote sets of strings. Example.
   statement →**if** *expression* **then** *statement* **else** *statement*
   <statement> → **if** <expression> **then** <statement> **else** <statement>
3. In a grammar, one nonterminal is distinguished as the start symbol, and the set of strings it denotes is the language defined by the grammar.
4. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of a nonterminal, followed by an arrow (sometimes the symbol ::= is used in place of the arrow), followed by a string of nonterminals and terminals.

3.3.1.4 Describing Lists

identifier-list →**id**
identifier-list →**id**, *identifier-list*

3.3.1.5 Grammars and Derivations

Id	LHS		RHS
1	*program*	→	**begin** *statement-list* **end**
2	*statement-list*	→	*statement*
3	*statement-list*	→	*statement* **;** *statement-list*
4	*statement*	→	*var* = *expression*
5	*var*	→	**A**
6	*var*	→	**B**
7	*var*	→	**C**
8	*expression*	→	*var* **+** *var*
9	*expression*	→	*var* **–** *var*
10	*expression*	→	*var*

Consider a sentence, a program, in the grammar given above.
begin A = B + C ; B = C end

A sentence in the grammar consists solely of terminal symbols. A derivation starts with the distinguished symbol and proceeds by substituting productions of the grammar for nonterminals to derive the sentence.

Sentential form (rightmost derivation)	**Id**	**LHS**		**RHS**
program		Start with the start symbol		
begin *statement-list* **end**	1	*program*	→	**begin** *statement-list* **end**
begin *statement* **;** *statement-list* **end**	**3**	*statement-list*	→	*statement* **;** *statement-list*
begin *statement* **;** *statement* **end**	**2**	*statement-list*	→	*statement*
begin *statement* **;** *var* = *expression* **end**	**4**	*statement*	→	*var* = *expression*
begin *statement* **;** *var* = *var* **end**	**10**	*expression*	→	*var*
begin *statement* **;** *var* = **C end**	**7**	*var*	→	**C**
begin *statement* **; B = C end**	**6**	*var*	→	**B**
begin *var* = *expression* **; B = C end**	**4**	*statement*	→	*var* = *expression*
begin *var* = *var* **+** *var* **; B = C end**	**8**	*expression*	→	*var* **+** *var*
begin *var* = *var* **+ C ; B = C end**	**7**	*var*	→	**C**
begin *var* **= B + C ; B = C end**	**6**	*var*	→	**B**
begin A = B + C ; B = C end	**5**	*var*	→	**A**

3.3.1.6 Parse Trees

Id	LHS		RHS
1	*assignment*	→	*id* = *expression*
2	*id*	→	**A**
3	*id*	→	**B**
4	*id*	→	**C**
5	*expression*	→	*id* **+** *expression*
6	*expression*	→	*id* ***** *expression*
7	*expression*	→	**(** *expression* **)**
8	*expression*	→	*id*

Sentential form (leftmost derivation)	**Id**	**LHS**		**RHS**
assignment		Start with the start symbol		
id = *expression*	1	*assignment*	→	*id* = *expression*
A = *expression*	2	*id*	→	**A**
A = *id* ***** *expression*	6	*expression*	→	*id* ***** *expression*
A = **B *** *expression*	3	*id*	→	**B**
A = **B * (** *expression* **)**	7	*expression*	→	**(** *expression* **)**
A = **B * (***id* **+** *expression***)**	5	*expression*	→	*id* **+** *expression*
A = **B * (A +** *expression***)**	2	*id*	→	**A**
A = **B * (A +** *id***)**	8	*expression*	→	*id*
A = **B * (A + C)**	4	*id*	→	**C**

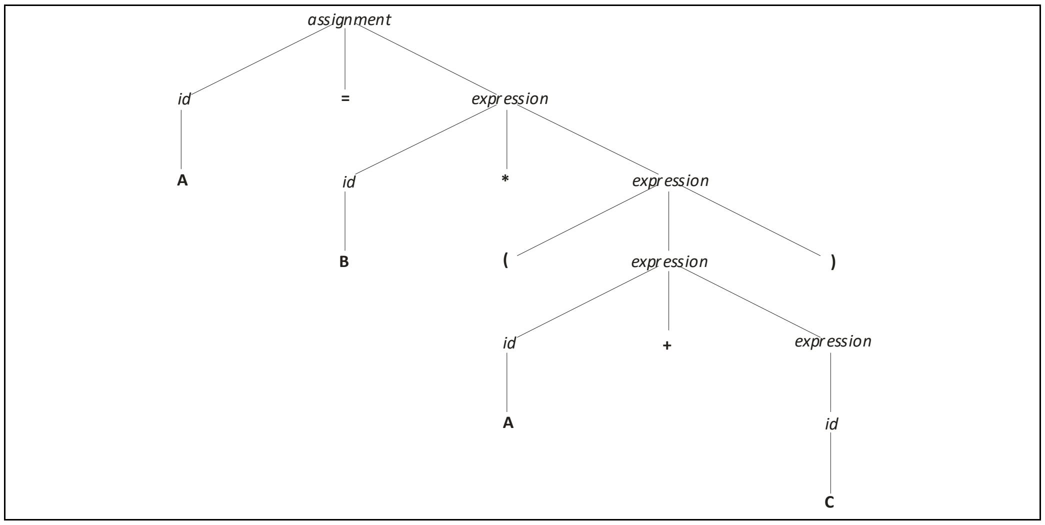


Figure 3.1 Parse Tree for the statement A=B*(A+C)

3.3.1.7 Ambiguity

Definition	A grammar that generates a sentential form for which there are two or more distinct parse trees is said to be **ambiguous**.

The following grammar is ambiguous.

Id	**LHS**		**RHS**
1	*assignment*	→	*id* **=** *expression*
2	*id*	→	**A**
3	*id*	→	**B**
4	*id*	→	**C**
5	*expression*	→	*expression* **+** *expression*
6	*expression*	→	*expression* ***** *expression*
7	*expression*	→	**(** *expression* **)**
8	*expression*	→	*id*

Consider the two parse trees for the sentence
A=B+C*A

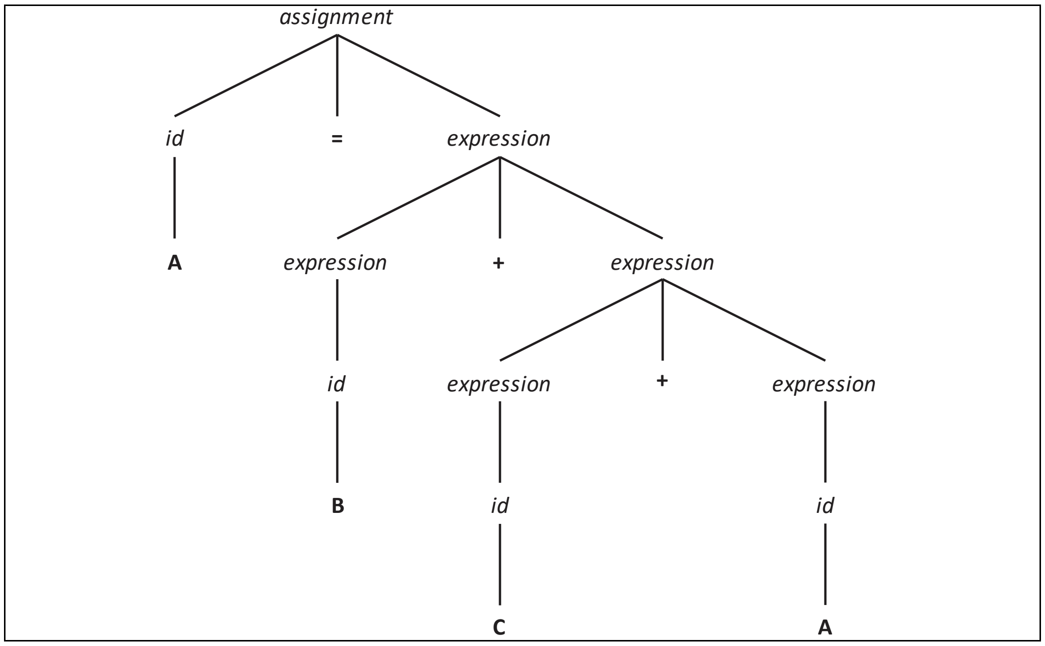


Figure 3.2.1 Parse Tree 1 for the sentence A=B+C+A

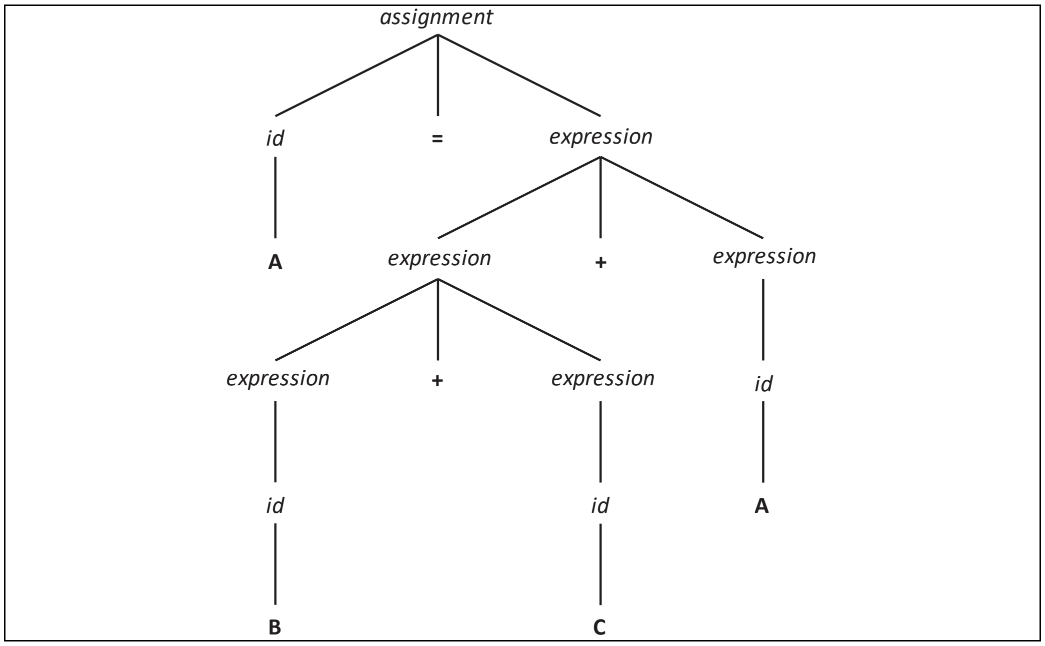


Figure 3.2.2 Parse Tree 2 for the sentence A=B+C+A

3.3.1.8 Operator Precedence

Definition	**Operator precedence** specifies the order of execution for different operators in an expression.

Example: The following unambiguous grammar specifies the normal order in which operators are evaluated in arithmetic expressions.

Id	LHS		RHS
1	*assignment*	→	*id* = *expression*
2	*id*	→	**A**
3	*id*	→	**B**
4	*id*	→	**C**
5	*expression*	→	*term*
6	*expression*	→	*expression* **+** *term*
7	*term*	→	*factor*
8	*term*	→	*term* ***** *factor*
9	*factor*	→	**(** *expression* **)**
10	*factor*	→	*id*

Operator	Precedence	Associativity
()	4 (Highest)	
*	3	left
+	2	left
=	1	

Sentential form (leftmost derivation)	Id	LHS		RHS
assignment		Start with the start symbol		
id = *expression*	**1**	*assignment*	→	*id* = *expression*
A = *expression*	**2**	*id*	→	**A**
A = *expression* **+** *term*	**6**	*expression*	→	*expression* **+** *term*
A = *term* **+** *term*	**5**	*expression*	→	*term*
A = *factor* **+** *term*	**7**	*term*	→	*factor*
A = *id* **+** *term*	**10**	*factor*	→	*id*
A = B + *term*	**3**	*id*	→	**B**
A = B + *term* ***** *factor*	**8**	*term*	→	*term* ***** *factor*
A = B + *factor* ***** *factor*	**7**	*term*	→	*factor*
A = B + *id* ***** *factor*	**10**	*factor*	→	*id*
A = B + C * *factor*	**4**	*id*	→	**C**
A = B + C * *id*	**10**	*factor*	→	*id*
A = B + C * A	**2**	*id*	→	**A**

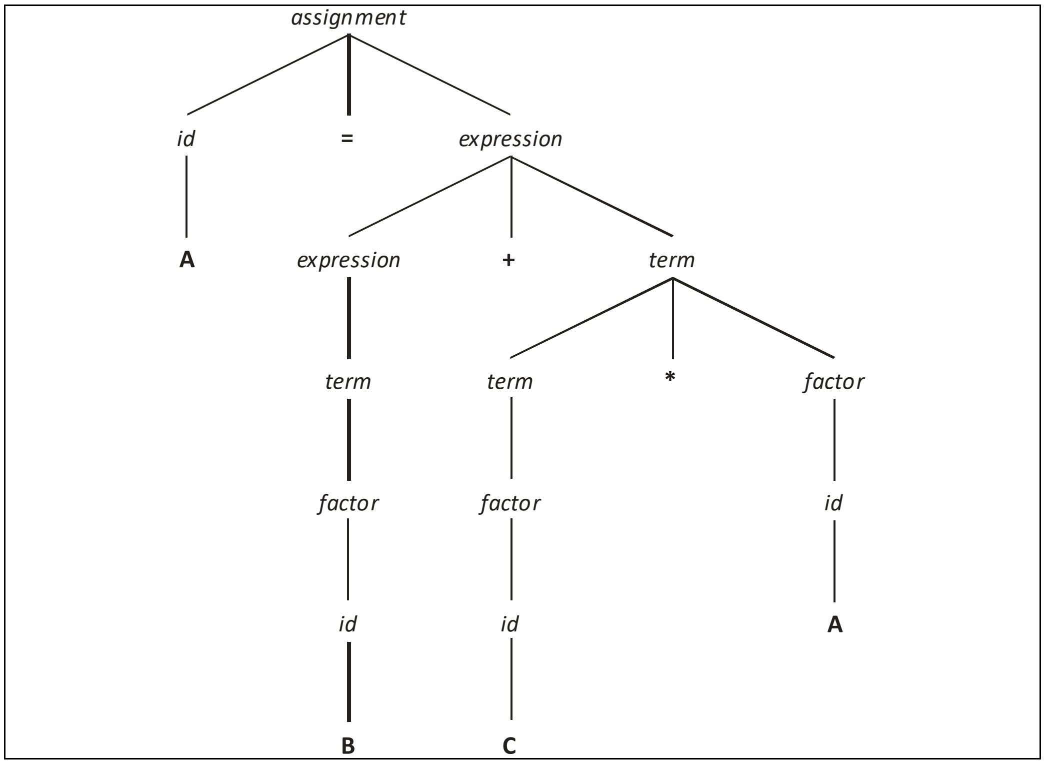


Figure 3.3 Unique parse tree for A=B+C*A using an unambiguous grammar

3.3.1.9 Associativity of Operators

Definition	**Associativity** defines the order of operations in an expression when operators have the same precedence.

Example: **A=B+C+A**
There are two operations in the expression B+C+A. Since both operations are addition, they have the same precedence. Which addition is performed first? Is the sum of B+C computed first and then the value of A added to the sum. Or, alternatively, is the sum of C+A found first followed by adding B to the sum of C+A?

The grammar defines the order in which operations are performed. An operator can left-associative or right-associative. If an operator is left-associative then operations are performed left to right. If an operator is right-associative, operations are performed right to left.

Consider the excerpt from the C++ grammar for additive expressions.

Id	**LHS**		**RHS**
	additive-expression	→	*multiplicative-expression*
	additive-expression	→	*additive-expression* **+** *multiplicative-expression*
	additive-expression	→	*additive-expression* **-** *multiplicative-expression*

Addition operators, **+** and **-**, associative to the left because the recursive nonterminal symbol, *additive-expression*, appears leftmost on the right hand side of the production.

Consider the following grammar and a sentence in the grammar **A=B+C+A**

Id	LHS		RHS
	assignment	→	*id* **=** *expression*
	id	→	**A**
	id	→	**B**
	id	→	**C**
	expression	→	*expression* **+** *primary*
	expression	→	*expression* **-** *primary*
	primary	→	*id*
	primary	→	**(** *expression* **)**

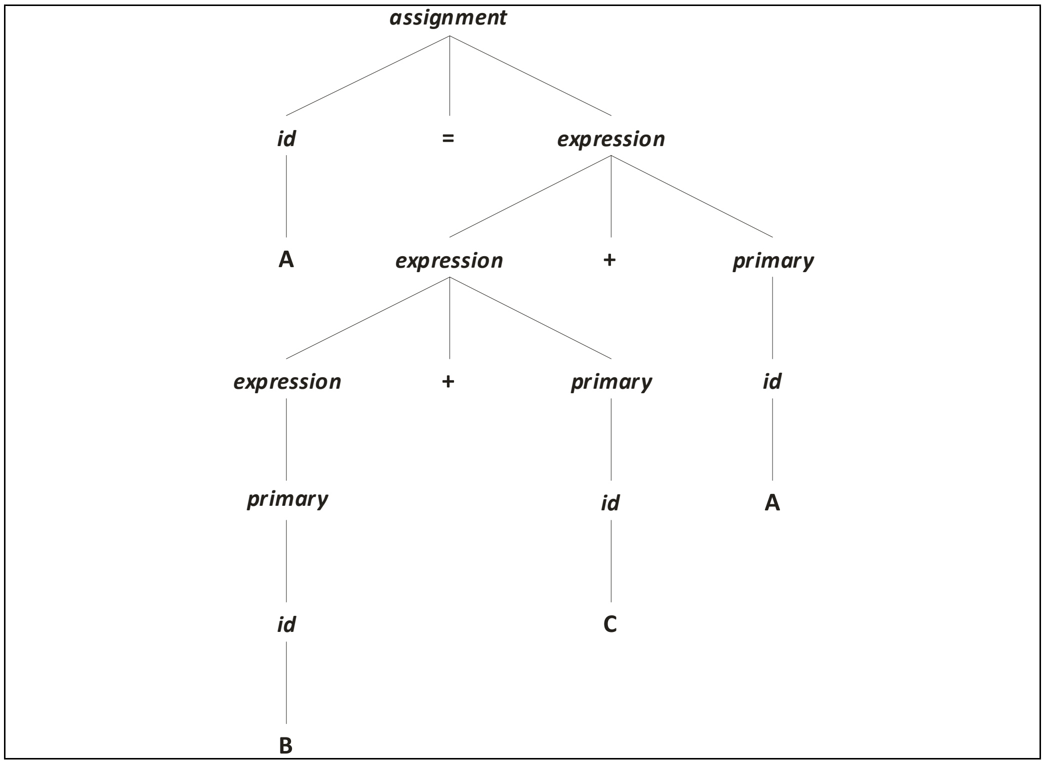


Figure 3.4 A Parse Tree for A=B+C+A that illustrate a left-associative operator

In a similar way, assignment-expressions are right-associative.

Id	LHS		RHS
	assignment-expression	→	*conditional-expression*
	assignment-expression	→	*logical-or-expression assignment-operator assignment-expression*
	assignment-expression	→	*throw-expression*
	assignment-operator	→	**=**
	assignment-operator	→	***=**
	assignment-operator	→	**/=**
	assignment-operator	→	**=**
	assignment-operator	→	**%=**
	assignment-operator	→	**+=**
	assignment-operator	→	**-=**
	assignment-operator	→	**>>=**
	assignment-operator	→	**<<=**
	assignment-operator	→	**&=**
	assignment-operator	→	**^=**
	assignment-operator	→	**|=**

Note that the recursive nonterminal assignment-expression appears rightmost on the right hand side of the highlighted production.

Consider the statement **A=B=C=A+3**
and the grammar

Id	LHS		RHS
	assignment	→	*expression* **=** *assignment*
	assignment	→	*expression*
	expression	→	*expression* **+** *primary*
	expression	→	*primary*
	primary	→	**(** *expression* **)**
	primary		**INTLIT**
	primary	→	**ID**

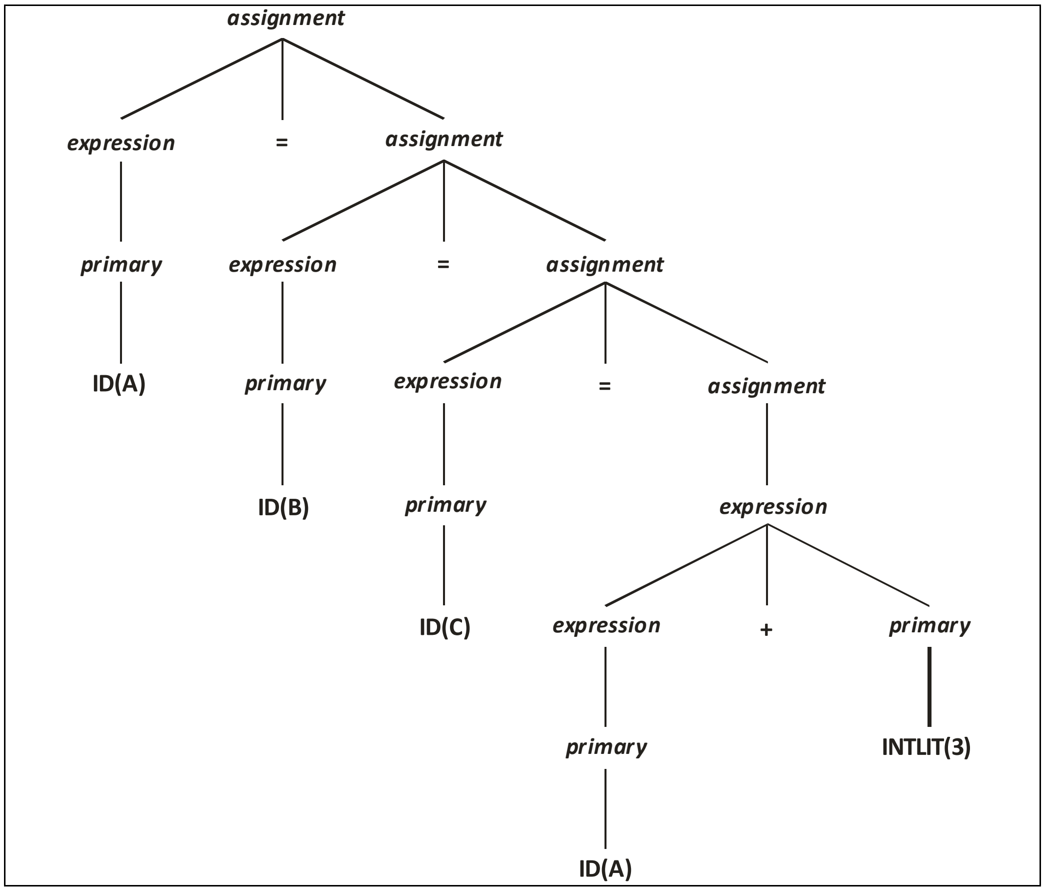


Figure 3.4 A Parse Tree for A=B=C=A+3 that illustrate a right-associative operator

3.3.1.10 An Unambiguous Grammar of if-then-else

Id	LHS		RHS
	if-statement	→	**if** *expression* **then** *statement*
	if-statement	→	**if** *expression* **then** *statement* **else** *statement*

if *done* **= true**
    then if *denom* **= 0**
        then *quotient* **:= 0;**
        else *quotient* **:=** *num* **div** *denom***;**

Id	LHS		RHS
	statement		*matched-statement*
	statement		*unmatched-statement*
	matched-statement		**if** *expression* **then** *matched-statement* **else** *unmatched-statement*
	matched-statement		*any-non-if-statement*
	unmatched-statement		**if** *expression* **then** *statement*
	unmatched-statement		**if** *expression* **then** *matched-statement* **else** *unmatched-statement*

3.3.2 Extended BNF

Extended BNF or EBNF has three additions not included in BNF.

1. Braces [] designate optional suffixes
   Example BNF:
   if-statement → **if (** *expression* **)** *statement*
   if-statement → **if (** *expression* **)** *statement* **else** *statement*

   Equivalent EBNF:
   if-statement → **if (** *expression* **)** *statement* [**else** *statement*]

   Note that the braces are not printed in bold to distinguish them from terminal symbols which are printed in bold.

2. Brackets {} specify that a suffix is repeated zero or more times
   Example BNF:
   identifier-list → *identifier*
   identifier-list → *identifier-list* **,** *identifier*

   Equivalent EBNF:
   identifier-list → *identifier* { **,** *identifier*}

   Note that the brackets are not printed in bold to distinguish them from terminal symbols which are printed in bold.

3. Parentheses () indicate that one of the items enclosed in the parentheses must be selected.
   Example BNF:
   term → *term* * *factor*
   term → *term* **/** *factor*
   term → *term* **%** *factor*

   Equivalent EBNF:
   term → *term* {(*|**/**|**%**) *term*}

   Note that the parentheses are not printed in bold to distinguish them from terminal symbols which are printed in bold.

BNF:

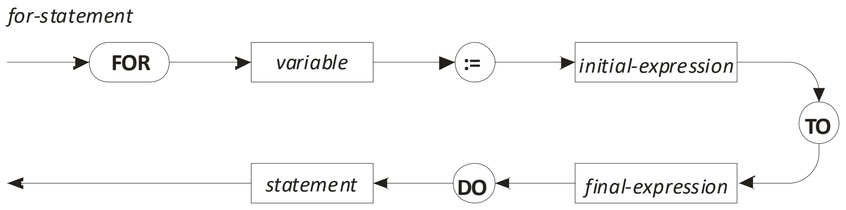
expression	→	*term*
expression	→	*expression* **+** *term*
expression	→	*expression* **-** *term*
term	→	*factor*
term	→	*term* ***** *factor*
term	→	*term* **/** *factor*
factor	→	**id**
factor	→	**(** *expression* **)**

EBNF:

expression	→	*term* {(**+**|**-**) *term*}
term	→	*factor* {(*****|**/**|**-**) *factor*}
factor	→	**id** | **(** *expression* **)**

3.3.2.1 Syntax Graphs

for-statement → **for** *variable* **:=** *initial-expression* **to** *final-expression* **do** *statement*



Syntax Graph for a *for-statement*

3.3.3 Grammars and Recognizers

- Given a context free grammar a recognizer can be implemented.
- A recognizer is a parser or a syntax analyzer, the second phase of a compiler.

3.4 Attribute Grammars

- An attribute grammar can be used to amplify a context free grammar to include semantic information.

3.4.1 Static Semantics

- Any form of static analysis is analysis performed at **compile-time** – during compilation.
- Dynamic analysis, in contrast to static analysis, is performed during execution.
- Static semantics refers to those semantics that can be performed during compilation.

3.4.2 Basic Concepts

- Attribute grammars are context-free grammars to which have been added attributes, attribute computation functions, and predicate functions.
- **Attributes**, which are associated with grammar symbols, terminal and nonterminal, are similar to variables in the sense that they can have values assigned to them.
- **Attribute computation functions**, sometimes called semantic functions, are associated with grammar rules.
- **Predicate functions**, which state the static semantic rules of the language, are associated with grammar rules.

3.4.3 Attribute Grammars Defined

- Associated with each grammar symbol X is a set of attributes $A(X)$. The set $A(X)$ consists of two disjoint sets $S(X)$ and $I(X)$, called **synthesized** and **inherited** attributes.

3.4.4 Synthesized Attributes

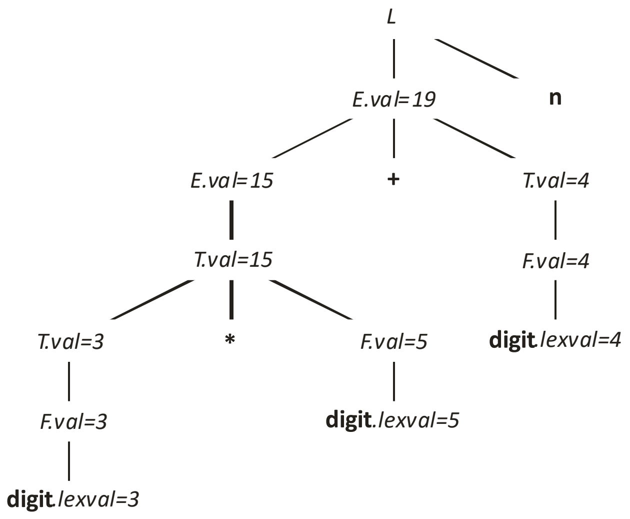
Synthesized Attributes. Given a production rule $X_0 \to X_1 \cdots X_n$, synthesized attributes of X_0 can be computed using a function $S(X_0) = f\big(A(X_1) \cdots A(X_n)\big)$.

Example:

Production Rule	Semantic Rule
$L \to E\ \mathbf{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val \coloneqq E_1.val + T.val$
$E \to T$	$E.val \coloneqq T.val$
$T \to T_1 * F$	$T.val \coloneqq T_1.val \times F.val$
$T \to F$	$T.val \coloneqq F.val$
$F \to (\ E\ )$	$F.val\!:= E.val$
$F \to \mathbf{digit}$	$F.val \coloneqq \mathbf{digit}.lexval$

Notes:

1. In the production, $L \to E\ \mathbf{n}$**,** the $\mathbf{n}$ represents a newline character.
2. Members of the grammar symbols are the synthesized attributes. For example, member val is a synthesized attribute of grammar symbol E.



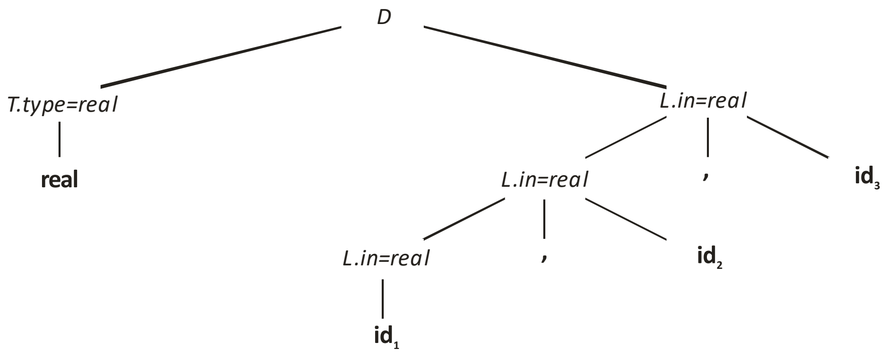
Annotated parse tree for 3*5+4 n

3.4.5 Inherited Attributes

Inherited Attributes. Given a production rule $X_0 \rightarrow X_1 \cdots X_j \cdots X_n$, inherited attributes of X_j can be computed using a function $I(X_j) = f(A(X_1) \cdots A(X_n))$ where $1 \leq j \leq n$

Example:

Production Rule	Semantic Rule
$D \rightarrow T\ L$	$L.in \coloneqq T.type$
$T \rightarrow \mathbf{int}$	$T.type \coloneqq integer$
$T \rightarrow \mathbf{real}$	$T.type \coloneqq real$
$L \rightarrow L_1\ ,\ \mathbf{id}$	$L_1.in \coloneqq L.in$
	$addtype(\mathbf{id}.entry, L.in)$
$L \rightarrow \mathbf{id}$	$addtype(\mathbf{id}.entry, L.in)$



Parse tree with inherited attribute in at each node labeled L.

3.5 Describing the Meanings of Programs: Dynamic Semantics

- Dynamic semantics are used to specify the meaning of expressions, statements, and program units of a programming language.
- Describing syntax is relatively easy – describing semantics is more difficult because
  - There is no universally accepted notation for dynamic semantics.
  - Imprecise English descriptions are often used.
  - Semantics are implemented in the compiler translating the language and different compilers may implement a language differently even if the same machine is targeted.

3.5.1 Operational Semantics

- Operational semantics are used to describe the meaning of a statement or program by specifying the effects of running it on a machine.

3.5.1.1 The Basic Process

- Example

C Statement	*Meaning*

for (*expr1***;** *expr2***;** *expr3***)** *statement***;**	expr1;
	loop: **if** expr2 == 0 **goto** out
	statement;
	expr3;
	goto loop;
	out:

- Programming language constructs are defined in terms of virtual machine primitives.
- The virtual machine is a precise definition of a machine that has no real existence.

3.5.1.2 Evaluation

- PL/I was the first language to employ formal operational semantics. IBM named the semantic definition language the Vienna Definition Language (VDL).

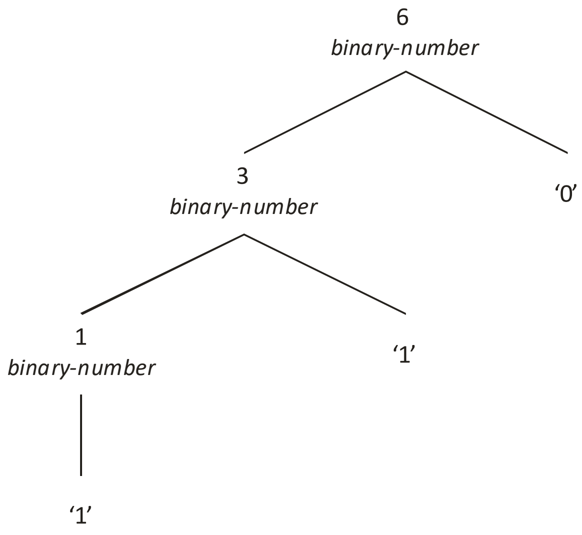
3.5.2 Denotational Semantics

- Denotational semantics is the most rigorous and most widely known method for describing the meaning of programs.
- Define a mathematical object for each programming language entity.
- Define a mathematical function that maps instances of each language entity to its mathematical object.
- The domain of the mathematical functions for a denotational semantics programming language specification is called the **syntactic domain**.
- The range of the mathematical functions for a denotational semantics programming language specification is called the **semantic domain**.
- In operational semantics, programming language constructs are translated into simpler programming constructs, which become the basis of the meaning of the construct.
- In denotational semantics, programming language constructs are mapped to mathematical objects, either sets or, more often, functions. However, unlike

operational semantics, denotational semantics does not model the step-by-step computational processing of programs.

3.5.2.1 Two Simple Examples

binary-number	→	**‘0’**
binary-number	→	**‘1’**
binary-number	→	*binary-number* **‘0’**
binary-number	→	*binary-number* **‘1’**



Parse Tree of the Binary Number 110

- The semantic function M_{bin}, maps the syntactic objects, as described in the previous grammar rules, to the objects in N, the set of non-negative decimal numbers.

$$M_{bin}('0') = 0$$
$$M_{bin}('1') = 1$$
$$M_{bin}(binary - number\ '0') = 2 \times M_{bin}(binary - number)$$
$$M_{bin}(binary - number\ '1') = 2 \times M_{bin}(binary - number) + 1$$

3.5.2.2 The State of a Program

- Denotational semantics employs the state of the program to describe meaning.
- The state of the program is a set of ordered pairs,
$$s = \{< i_1.v_1 >, < i_2.v_2 >, \cdots, < i_n.v_n >\}$$
- Each i is the name of a variable.
- Each corresponding v is the current value of the variable.
- In contrast operational semantics are defined in terms of the state changes on an ideal computer.
- The value of **VARMAP**(i_j, s) is v_j, the value paired with i_j in state s.

3.5.2.3 Expressions

- Consider the expression grammar below.

expression	→	*decimal-number*
expression	→	*variable*
expression	→	*binary-expression*
binary-expression	→	*left-expression operator right-expression*
left-expression	→	*decimal-number*
left-expression	→	*variable*
right-expression	→	*decimal-number*
right-expression	→	*variable*
operator	→	**+**
operator	→	**-**

- The mapping function for a given expression E and a state s.
    - The symbol $\Delta=$ is used to define mathematical functions.
    - The symbol $=>$ is used in the following definitions to connect the form of an operand with its associated case (or switch) construct.
    - Dot notation is used to refer to the child nodes of a node. For example, <binary_expr>.<left_expr> refers to the left child node of the <binary_expr>.

```
$M_e(expression, s)\Delta=$ case $expression$ of
    decimal-number    => $M_e$(decimal-number,s)
    variable          => if VARMAP(variable,s)==undef
                             then error
                             else VARMAP(variable,s)
    decimal-number    =>
                         if  ($M_e$(binary-expression.left-expression,s)==undef
                         OR  $M_e$(binary-expression.right-expression,s) ==undef)
                             then error
                             else if (binary-expression.operator=='+')
                                 then  $M_e$(binary-expression.left-expression,s)
                                    +  $M_e$(binary-expression.right-expression,s)
                                 else  $M_e$(binary-expression.left-expression,s)
                                    -  $M_e$(binary-expression.right-expression,s)
```

3.5.2.4 Assignment Statements

```
$M_a\ (x = E, s)\Delta=$if $M_e(E,s)$== error
    then error
    else $s' = \{< i_1, v_1' >, < i_2, v_2' >, \cdots, < i_n, v_n' >\}$, where
        for $j = 1,2,\cdots,n$
            if $i_j == x$
                then $v_j' = M_e(E,s)$
                else $v_j'$ =VARMAP($i_j, s$)
```

3.5.2.5 Logical Pretest Loops

```
$M_l$ (while $B$ do $L, s)\Delta=$ if $M_b(B,s)$==undef
                                then error
```

```
else if $M_b(B,s)$==false
        then $s$
        else if $M_{sl}(L,s)$==error
                then error
                else $M_l$ (while $B$ do $L, M_{sl}(L,s)$)
```

3.5.2.6 Evaluation

- When the description of a language construct employing denotational semantics proves to be difficult, it can be a sign that the construct is ill-conceived.
- The complexity of denotational descriptions makes them of ***little use*** to language users.
- Other than PL/I our text mentions no programming languages where denotational semantics were employed.

3.5.3 Axiomatic Semantics

- Axiomatic semantics are based on mathematical logic.
- The primary use of axiomatic semantics is to prove that program fragments function according to specification – program verification. This is called program proof of correctness.

3.5.3.1 Assertions

- Logical expressions used in axiomatic semantics are called **predicates** or **assertions**.
- Assertions are used to define the **precondition** and the **postcondition** of a statement.
  Example postcondition (enclosed in braces}
  $\{x \geq 0\}$*sum*=**2***x*+**1;** {*sum*≥**1**}

3.5.3.2 Weakest Preconditions

- The weakest precondition is the least restrictive precondition that will guarantee the validity of the associated postcondition.
  Example preconditions for the postcondition {*sum*>**1**}
  - {*x*>**0**} *sum*=**2***x*+**1;** {*sum*>**1**}
  - {*x*>**10**} *sum*=**2***x*+**1;** {*sum*>**1**}
  - {*x*>**50**} *sum*=**2***x*+**1;** {*sum*>**1**}
  - {*x*>**1000**} *sum*=**2***x*+**1;** {*sum*>**1**}

  The weakest precondition that will guarantee the postcondition is {*x*>**0**}.

  An inference rule is a method of inferring the truth of one assertion on the basis of the values of other assertions.

$$\frac{S1, S2, \cdots, Sn}{S}$$

  This rule states that if $S1, S2, \cdots,$ and Sn are true, then the truth of S can be inferred.

3.5.3.3 Assignment Statements

- Let $x = E$ be a general assignment statement and Q be its postcondition. The precondition, P, is defined by the axiom.
  $$P = Q_{x \to E}$$
  which means that P is computed as Q with all instances of x replaced by E.
  Example: Find the weakest precondition given the following statement and postcondition.
  a=*b*/**2-1;** {*a*<**10**}
  1. *b*/**2-1 < 10**
  2. *b*/**2 < 11**
  3. *b* **< 22**

3.5.3.4 Sequences

- Let $S1$ and $S2$ be sequential statements and $P1$, $P2$, and $P3$ be assertions used either as preconditions or postconditions. Consider the following
  $\{P1\}\, S1\, \{P2\}$
  $\{P2\}\, S2\, \{P3\}$

  The inference rule for such a two-statement sequence is
  $$\frac{\{P1\}\, S1\, \{P2\}, \{P2\}\, S2\, \{P3\}}{\{P1\} S1; S2 \{P3\}}$$

3.5.3.5 Selection

- Consider
  if B **then** *S1* **else** *S2*

  The inference rule is:
  $$\frac{\{B \wedge P\} S1 \{Q\}, \{(\neg B) \wedge P\} S2 \{Q\}}{\{P\} \textbf{if } B \textbf{ then } S1 \textbf{ else } S2\ \{Q\}}$$

  where P is the precondition and Q is the postcondition.

  Example
  if *x*>**0 then** *y*=*y*-**1 else** *y*=*y*+**1 {***y*>**0}**

  Applying the postcondition to the then-clause.
  y=*y*-**1 {***y*>**0}** implies **{***y*>**1}**

  Applying the postcondition to the else-clause
  y=*y*+**1 {***y*>**0}** implies **{***y*>**0}**

  Because **{***y*>**1}** implies **{***y*>**0}** we use the **{***y*>**1}** for the precondition**.**

3.5.3.6 Logical Pretest Loops

- Loops are more difficult because the number of iterations is known, in many cases, only during execution.

- Axiomatic semantics seeks to find a **loop invariant** that is equivalent to the inductive hypothesis.
- The weakest precondition can be found by examining the loop invariant.
- The inference rule for computing the precondition for a **while**-loop is

  $$\frac{(I \wedge B) S \{I\}}{\{I\}\textbf{while } B \textbf{ do } S \textbf{ end } \{I \wedge (\neg B)\}}$$

  where I is the loop-invariant.

- The axiomatic description of a **while**-loop is:
  {P} while B do S end {Q}

3.5.3.7 Program Proofs

The first example of a correctness proof is for a very short program, consisting of a sequence of three assignment statements that interchange the value of two variables.

```
{x=A AND y=B}
t:=x;
x:=y;
y:=t;
{x=B AND y=A}
```

```
{x=A AND y=B}
t:=x;
{x=A AND t=A AND y=B}
x:=y;
{x=B AND t=A AND y=B}
y:=t;
{x=B AND t=A AND y=A}
{x=B AND y=A}
```

3.5.3.8 Evaluation

- Defining axioms or inference rules for all statements of programming languages has proven to be difficult.
- Axiomatic semantics is a tool for research into program correctness proofs.
- Axiomatic semantics has ***limited usefulness*** for compiler writers and for language users.