Evaluation of programming languages:

1. Zuse’s Plankalkül
2. Pseudocodes
3. The IBM 704 and Fortran
4. Functional Programming: LISP
5. The First Step Toward Sophistication: ALGOL 60
6. Computerizing Business Records: COBOL
7. The Beginnings of Timesharing: BASIC
8. Everything for Everybody: PL/I
9. Two Early Dynamic Languages: APL and SNOBOL
10. The Beginnings of Data Abstraction: SIMULA 67
11. Orthogonal Design: ALGOL 68
12. Some Early Descendants of the ALGOLs
13. Programming Based on Logic: Prolog
14. History’s Largest Design Effort: Ada
15. Object-Oriented Programming: Smalltalk
16. Combining Imperative and Object-Oriented Features: C++
17. An Imperative-Based Object-Oriented Language: Java
18. Scripting Languages
19. The Flagship .NET Language: C#
20. Markup/Programming Hybrid Languages

Zuse’s Plankalkül: 1945

- It is the first programming language, and is highly unusual in several respects.
- For one thing, it was never implemented.
- Furthermore, although developed in 1945, its description was not published until 1972.
- In those days only few people were familiar with the language, some of its capabilities did not appear in other languages until 15 years after its development.

Pseudocodes: 1949

The code written using any natural language like English is called "psuedocode". These codes help us to understand the logic.

What was wrong with using machine code?

a. Poor readability
b. Poor modifiability
c. Expression coding was tedious (boring)
d. Machine deficiencies--no indexing

-Short code: expressions were coded, left to right - Some operations

1n => (n+2)nd power
2n => (n+2)nd root
07 => addition

The IBM 704 AND FORTRAN I - 1957

(FORTRAN 0 - 1954 - not implemented)

- Designed for the new IBM 704, which had index registers and floating point hardware
- **Environment of development:**
  - ➢ Computers were small and unreliable
  - ➢ Applications were scientific
  - ➢ No programming methodology or tools
  - ➢ Machine efficiency was most important
- **Impact of environment on design**
  - ➢ No need for dynamic storage
  - ➢ Need good array handling and counting loops
  - ➢ No string handling, decimal arithmetic, or powerful input/output (commercial stuff)
- **First implemented version of FORTRAN has following things**
  - ➢ Names could have up to six characters
  - ➢ Post test counting loop (DO)
  - ➢ Formatted i/o
  - ➢ User-defined subprograms
  - ➢ Three-way selection statement (arithmetic IF)
  - ➢ No data typing statements
  - ➢ No separate compilation
  - ➢ Compiler released in April 1957, after 18 worker/ years of effort
  - ➢ Programs larger than 400 lines rarely compiled correctly, mainly due to poor reliability of the 704
  - ➢ Code was very fast
  - ➢ Quickly became widely used

FORTRAN II - 1958

- ➢ Independent compilation
- ➢ Fix the bugs

FORTRAN IV - 1960-62

- ➢ FORTRAN III was developed, but never widely distributed.
- ➢ Explicit type declarations
- ➢ Logical selection statement
- ➢ Subprogram names could be parameters
- ➢ ANSI standard in 1966

FORTRAN 77 - 1978

- ➢ Character string handling
- ➢ Logical loop control statement
- ➢ IF-THEN-ELSE statement

FORTRAN 90 - 1990

- ➢ Modules

- Dynamic arrays
- Pointers
- Recursion
- CASE statement
- Parameter type checking

FORTRAN Evaluation

- Dramatically changed forever the way computers are used

Functional Programming: LISP

- List Processing language (Designed at MIT by McCarthy)
- AI research needed a language that:
  - Process data in lists (rather than arrays)
  - Symbolic computation (rather than numeric)
- Only two data types: atoms and lists
- Syntax is based on lambda calculus
- Pioneered functional programming
  - No need for variables or assignment
  - Control via recursion and conditional expressions
- Still the dominant language for AI
- COMMON LISP and Scheme are contemporary dialects of LISP
- ML, Miranda, and Haskell are related languages

The First Step Toward Sophistication: ALGOL 60

- **Environment of development:**
- FORTRAN had (barely) arrived for IBM 70x
- Many other languages were being developed, all for specific machines
- No portable language; all were machine-dependent
- No universal language for communicating algorithms

ALGOL 58 (continued)

- ACM and GAMM met for four days for design
- **Goals of the language:**
- Close to mathematical notation
- Good for describing algorithms
- Must be translatable to machine code

- **Language Features:**
  - Concept of type was formalized
  - Names could have any length
  - Arrays could have any number of subscripts

- Parameters were separated by mode (in & out)
- Subscripts were placed in brackets
- Compound statements (begin ... end)
- Semicolon as a statement separator
- Assignment operator was :=
- if had an else-if clause

- ***Comments:***
  - Not meant to be implemented, but variations of it were (MAD, JOVIAL)
  - Although IBM was initially enthusiastic, all support was dropped by mid-1959

ALGOL 60 - 1960

- ***New Features*:**
  - Block structure (local scope)
  - Two parameter passing methods
  - Subprogram recursion
  - Stack-dynamic arrays
  - Still no i/o and no string handling

- ***Successes*:**
  - It was the standard way to publish algorithms for over 20 years
  - All subsequent imperative languages are based on it
  - First machine-independent language
  - First language whose syntax was formally defined (BNF)

- ***Failure*:**

  Never widely used, especially in U.S.

  Reasons:
  - No i/o and the character set made programs nonportable
  - Too flexible--hard to implement
  - Entrenchment of FORTRAN
  - Formal syntax description
  - Lack of support of IBM

<u>**Computerizing Business Records: COBOL 1960**</u>

- ***Environment of development:***
  - UNIVAC was beginning to use FLOW-MATIC
  - USAF was beginning to use AIMACO
  - IBM was developing COMTRAN
  - *Based on FLOW-MATIC*

- **FLOW-MATIC features:**
  - Names up to 12 characters, with embedded hyphens
  - English names for arithmetic operators
  - Data and code were completely separate
  - Verbs were first word in every statement
- **Design goals:**
  - Must look like simple English
  - Must be easy to use, even if that means it will be less powerful
  - Must broaden the base of computer users
  - Must not be biased by current compiler problems
  - Design committee were all from computer manufacturers and DoD branches
  - Design Problems: arithmetic expressions? subscripts? Fights among manufacturers

Contributions:
  - First macro facility in a high-level language
  - Hierarchical data structures (records)
  - Nested selection statements
  - Long names (up to 30 characters), with hyphens
  - Data Division

- ***Comments:***
  - First language required by DoD; would have failed without DoD
  - Still the most widely used business applications language

<u>The Beginnings of Timesharing: BASIC</u>

BASIC - 1964
- Designed by Kemeny & Kurtz at Dartmouth
- Design Goals:
  - Easy to learn and use for non-science students
  - Must be ”pleasant and friendly"
  - Fast turnaround for homework
  - Free and private access
  - User time is more important than computer time
- Current popular dialects: QuickBASIC and Visual BASIC

<u>Everything for Everybody: PL/I</u>

It is the first large-scale attempt to design a language that could be used for broad spectrum of application areas, such as science, Artificial Intelligence, and Business.

- **Designed by IBM and SHARE**
- ***Computing situation in 1964 (IBM's point of view)***
  1. Scientific computing
     - IBM 1620 and 7090 computers
     - FORTRAN
     - SHARE user group
  2. Business computing
     - IBM 1401, 7080 computers
     - COBOL
     - GUIDE user group
- **By 1963, however,**
  - Scientific users began to need more elaborate i/o, like COBOL had; Business users began to need fl. pt. and arrays (MIS)
  - It looked like many shops would begin to need two kinds of computers, languages, and support staff--too costly
- ***The obvious solution:***
  - Build a new computer to do both kinds of applications
  - Design a new language to do both kinds of applications
- ***PL/I contributions:***
  - First unit-level concurrency
  - First exception handling
  - Switch-selectable recursion
  - First pointer data type
  - First array cross sections
- ***Comments:***
  - Many new features were poorly designed
  - Too large and too complex
  - Was (and still is) actually used for both scientific and business applications

<u>Two Early Dynamic Languages: APL and SNOBOL</u>

- Characterized by dynamic typing and dynamic storage allocation
- APL (A Programming Language) 1962
  - Designed as a hardware description language (at IBM by Ken Iverson)
  - Highly expressive (many operators, for both scalars and arrays of various dimensions)
  - Programs are very difficult to read
- SNOBOL(1964)

- Designed as a string manipulation language (at Bell Labs by Farber, Griswold, and Polensky)
- Powerful operators for string pattern matching

The Beginnings of Data Abstraction: SIMULA 67

- Designed primarily for system simulation (in Norway by Nygaard and Dahl)
- Based on ALGOL 60 and SIMULA I
- *Primary Contribution:*
  - Coroutines - a kind of subprogram
    - Implemented in a structure called a class
      - Classes are the basis for data abstraction
        - Classes are structures that include both local data and functionality

Orthogonal Design: ALGOL 68

- From the continued development of ALGOL 60, but it is not a superset of that language
- Design is based on the concept of orthogonality
- ***Contributions:***
  1. User-defined data structures
  2. Reference types
  3. Dynamic arrays (called flex arrays)

- ***Comments:***
  - Had even less usage than ALGOL 60
  - Had strong influence on subsequent languages, especially Pascal, C, and Ada

Some Early Descendants of the ALGOLs

Pascal -1971

- Designed by Wirth, who quit the ALGOL 68 committee (didn't like the direction of that work)
- Designed for teaching structured programming - Small, simple, nothing really new
- Still the most widely used language for teaching programming in colleges (but use is shrinking)

C - 1972

- Designed for systems programming (at Bell Labs by Dennis Richie)
- Evolved primarily from B, but also ALGOL 68
- Powerful set of operators, but poor type checking - Initially spread through UNIX

Perl

- It is related to ALGOL through C language
- Variable names begin with $sign
- array names begin with @sign

Programming Based on Logic: Prolog

Developed at the University of Aix-Marseille,
by Comerauer and Roussel, with some help from Kowalski at the University of Edinburgh

- Based on formal logic
- Non-procedural

Can be summarized as being an intelligent database system that uses an inferencing process to infer the truth of given queries.

History's Largest Design Effort: Ada

- Huge design effort, involving hundreds of people, much money, and about eight years
- It was developed for DoD (Department of Defence, USA)
- ***Contributions:***
  1. Packages - support for data abstraction
  2. Exception handling - elaborate
  3. Generic program units
  4. Concurrency - through the tasking model
- ***Comments:***
  - Competitive design
  - Included all that was then known about software engineering and language design
  - First compilers were very difficult; the first really usable compiler came nearly five years after the language design was completed
- **Ada 95 (began in 1988)**
  - Support for OOP through type derivation
  - Better control mechanisms for shared data (new concurrency features)
    More flexible libraries

Object-Oriented Programming: Smalltalk

- Developed at Xerox PARC, initially by Alan Kay, later by Adele Goldberg
- First full implementation of an object-oriented language (data abstraction, inheritance, and dynamic type binding)
- Pioneered the graphical user interface everyone now uses

Combining Imperative and Object-Oriented Features: C++

- Developed at Bell Labs by Bjarne Stroustrup in the year 1979.
- Evolved from C and SIMULA 67
- Facilities for object-oriented programming, taken partially from SIMULA 67, were added to C
- Also has exception handling
- A large and complex language, in part because it supports both procedural and OO programming
- Rapidly grew in popularity, along with OOP
- ANSI standard approved in November, 1997
- Eiffel - a related language that supports OOP
  - (Designed by Bertrand Meyer - 1992)
  - Not directly derived from any other language

Smaller and simpler than C++, but still has most of the power

An Imperative-Based Object-Oriented Language: Java

- Developed at Sun in the early 1990s
- Based on C++
  - Significantly simplified
  - Supports *only* OOP
  - Has references, but not pointers
  - Includes support for applets and a form of concurrency

Scripting Languages: Java Script, PHP, Python and Ruby

- These are used in web applications
- Java Script is a HTML Resident client side scripting language
- PHP is an HTML Resident server side scripting language
- Python and Ruby are used for Common Gateway Interface programming

The Flagship .NET Language: C#

- C#, along with new development platform .NET was , developed by Microsoft in the year 2000
- It is based on C++ and Java
- The purpose of C# is to provide a language for component based software development
- Components from different languages such as Visual Basics .NET, Managed C++, J# .NET, and Jscript can be easily combined to form systems.

Markup/Programming Hybrid Languages

-XSLT –eXtensible Style Sheet language used for for transforming the markup languages
- JSP –Java Server pages are used for Server Side Programming

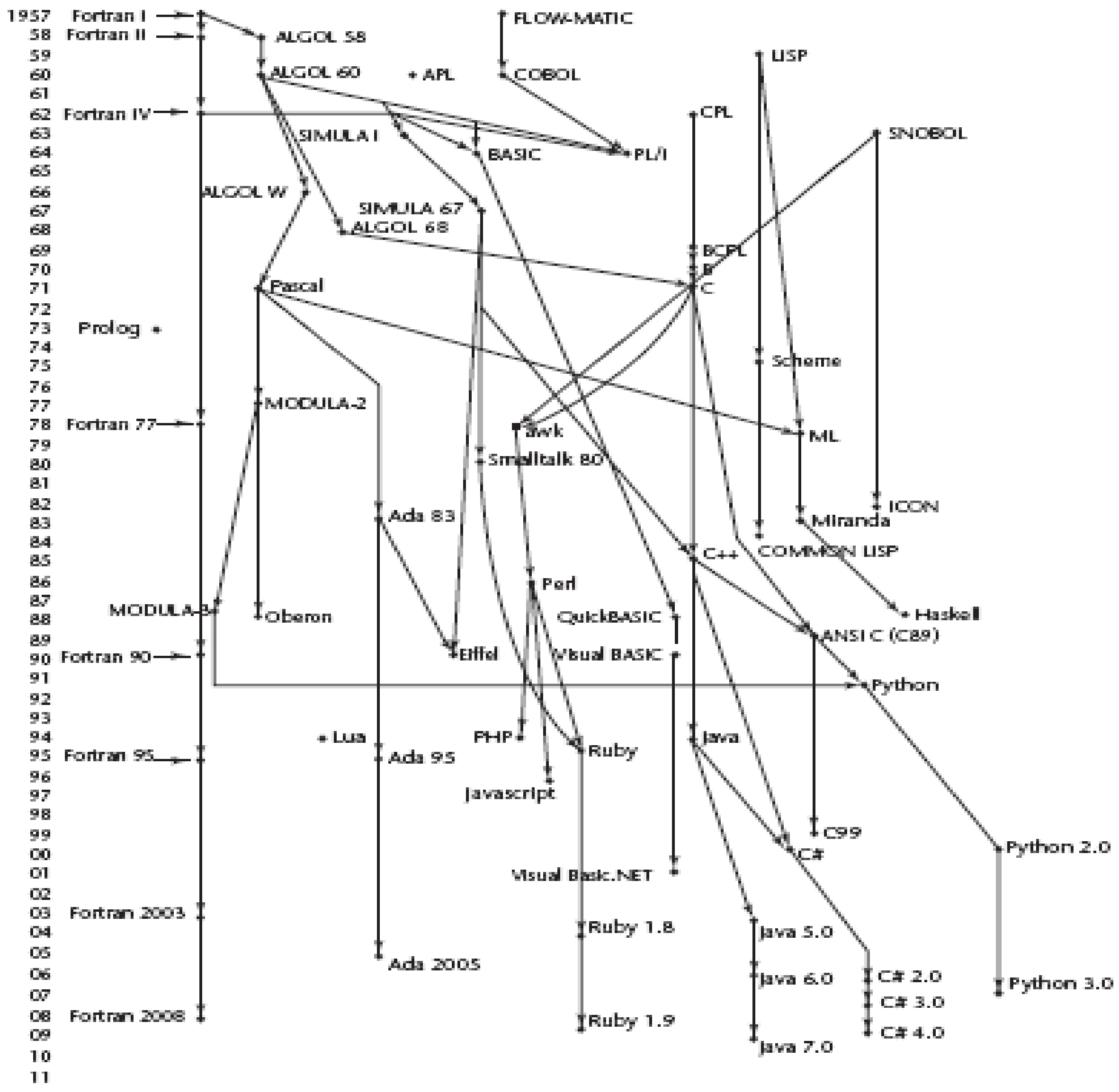


Fig: Genealogy of High Level Programming Languages

The General Problem of Describing Syntax

A **language,** whether natural (such as English) or artificial (such as Java), is a set of strings of characters from some alphabet. The strings of a language are called **sentences** or statements. The syntax rules of a language specify which strings of characters from the language's alphabet are in the language. A ***lexeme*** is the lowest level syntactic unit of a language (e.g., *, sum, begin). A ***token*** is a category of lexemes (e.g., identifier). The lexemes of a programming language include its numeric literals, operators, and special words, among others. Program is strings of lexemes rather than of characters. Lexemes are partitioned into groups—for example, the names of variables, methods, classes, and so forth in a programming language form a group called *identifiers*. Each lexeme group is represented by a name, or token. For example, an identifier is a token that can have lexemes, or instances, such as sum and total. In some cases, a token has only a single possible lexeme. For example, the token for the arithmetic operator symbol + has just one possible lexeme. Consider the following Java statement:

index = 2 * count + 17;

The lexemes and tokens of this statement are

Lexemes	*Tokens*
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

Language Recognizers

In general, languages can be formally defined in two distinct ways: by **recognition** and by **generation**. Suppose we have a language L that uses an alphabet of characters. To define L formally using the recognition method, we would need to construct a mechanism R, called a recognition device, capable of reading strings of characters from the alphabet. The syntax analysis part of a compiler is a recognizer for the language the compiler translates. In this role, it determine whether given programs are in the language. In effect then, the syntax analyzer determines whether the given programs are syntactically correct. The structure of syntax analyzers, also known as parsers,

Language Generators

A language generator is a device that can be used to generate the sentences of a language. It is often possible to determine whether the syntax of a particular statement is correct by comparing it with the structure of the generator. There is a close connection between formal generation and recognition devices

Formal Methods of Describing Syntax

1. Backus-Naur Form

2. Context-Free Grammars

Backus-Naur Form

In the mid 1950s, two men, Noam Chomsky and John Backus, in unrelated research efforts, developed the same syntax description formalism, which became the most widely used method for describing programming language syntax. The new notation of Backus was later modified slightly by Peter Naur for the description of ALGOL 60 (Naur, 1960). This revised method of syntax description became known as **Backus-Naur Form**, or simply **BNF**. BNF is a natural notation for describing syntax. BNF is nearly identical to Chomsky's generative devices for context-free languages, called **context-free grammars**. we refer context-free grammars simply as grammars. Furthermore, the terms BNF and grammar are used interchangeably.

Context-Free Grammars

In the mid-1950s, Chomsky described four classes of generative devices or grammars that define four classes of language. Two of these grammar classes, named *context-free* and *regular,* useful for describing the syntax of programming languages. The forms of the tokens of programming languages can be described by regular grammars. The syntax of whole programming languages, can be described by context-free grammars.

Fundamentals

A **metalanguage** is a language that is used to describe another language. BNF is a meta language for programming languages. BNF uses abstractions for syntactic structures. A simple Java assignment statement, for example, might be represented by the abstraction <assign> (pointed brackets are often used to delimit names of abstractions). The actual definition of <assign> can be given by
<assign> -> <var> = <expression>
The text on the left side of the arrow, which is called the **left-hand side** (LHS), The text to the right of the arrow is the definition of the LHS. It is called the **right-hand side** (RHS) and consists of some mixture of tokens, lexemes. Altogether, the definition is called a **rule**, or **production**.This particular rule specifies that the abstraction <assign> is defined as an instance of the abstraction <var>, followed by the lexeme =, followed by an instance of the abstraction <expression>. One example sentence whose syntactic structure is described by the rule is
total = subtotal1 + subtotal2
The abstractions in a BNF description, or grammar, are often called **nonterminal symbols**, or simply **nonterminals**, and the lexemes and tokens of the rules are called **terminal symbols**, or simply **terminals**. A BNF description,

or **grammar**, is a collection of rules.
Nonterminal symbols can have two or more distinct definitions, representing two or more possible syntactic forms in the language. Multiple definitions can be written as a single rule, with the different definitions separated by the symbol |, meaning logical OR. For example, a Java **if** statement can be described with the rules
<if_stmt> →**if** (<logic_expr>) <stmt>
<if_stmt> →**if** (<logic_expr>) <stmt> **else** <stmt>
or with the rule
<if_stmt> →**if** (<logic_expr>) <stmt>
| **if** (<logic_expr>) <stmt> **else** <stmt>
In these rules, <stmt> represents either a single statement or a compound statement.

Describing Lists

Variable-length lists in mathematics are often written using an ellipsis (. . .); BNF does not include the ellipsis, so an alternative method is required for describing lists of syntactic elements in programming languages (for example, a list of identifiers appearing on a data declaration statement). For BNF, the alternative is recursion. A rule is **recursive** if its
LHS appears in its RHS. The following rules illustrate how recursion is used to describe lists:

```
<ident_list> –> identifier
                | identifier, <ident_list>
```

This defines <ident_list> as either a single token (identifier) or an identifier
followed by a comma and another instance of <ident_list>. Recursion is used to
describe lists in many of the example grammas.

Grammars and Derivations

A grammar is a generative device for defining languages. The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the **start symbol**. This sequence of rule applications is called a **derivation**. In a grammar for a complete programming language, the start symbol represents a complete program and is often named <program>. The simple grammar is following.

EXAMPLE : **A Grammar for a Small Language**

```
<program> -> begin <stmt_list> end
<stmt_list> -> <stmt>
                | <stmt> ; <stmt_list>
<stmt> –> <var> = <expression>
<var> –> a | b | c
<expression> –> <var> + <var>
                  | <var> - <var>
                  | <var>
```

The language described by the grammar above has only one statement form: assignment. A program consists of the special word **begin**, followed by a list of statements separated by semicolons, followed by the special word **end**. An expression is either a single variable or two variables separated by either a + or – operator. The only variable names in this language are a, b, and c.

Derivation: Generating sentences of grammar

A derivation of a program in this language follows:

```
<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
```

This derivation, like all derivations, begins with the start symbol, in this case <program>. The symbol => is read "derives." Each successive string in the sequence is derived from the previous string by replacing one of the nonterminals with one of that nonterminal's definitions. Each of the strings in the derivation, including <program>, is called a **sentential form**.

In this derivation, the leftmost nonterminal is replaced at each step are called **leftmost derivations**. If the rightmost nonterminal is replaced at each step, it is called **right most derivation**

EXAMPLE 3.2: **A Grammar for Simple Assignment Statements**

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

The above 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 leftmost 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 )
```

Parse Trees

One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the languages they define. These hierarchical structures are called **parse**

trees. For example, the parse tree in Figure 3.1 shows the structure of the assignment statement derived previously.

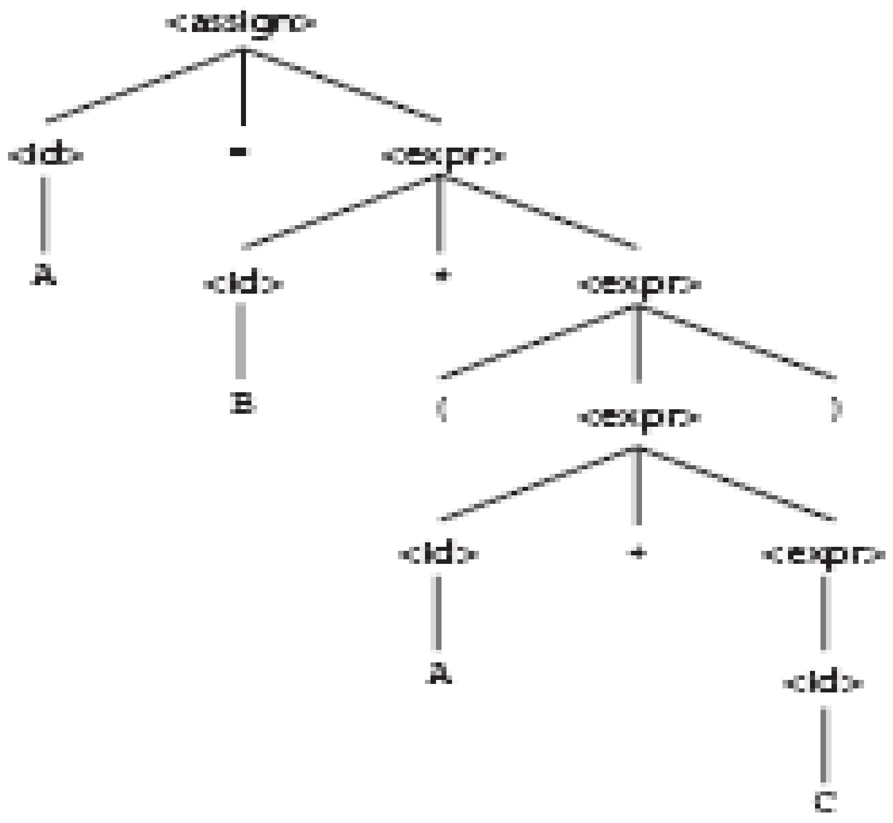


figure 3.1 : parse tree for `a = b * ( a + c )`

Every internal node of a parse tree is labeled with a nonterminal symbol; every leaf is labeled with a terminal symbol. Every subtree of a parse tree describes one instance of an abstraction in the sentence.

Ambiguity

A grammar that generates a sentential form for which there are two or more distinct parse trees is said to be **ambiguous**. Consider the grammar shown in Example 3.3, which is a minor variation of the grammar shown in Example 3.2.

EXAMPLE 3.3 **An Ambiguous Grammar for Simple Assignment Statements**

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

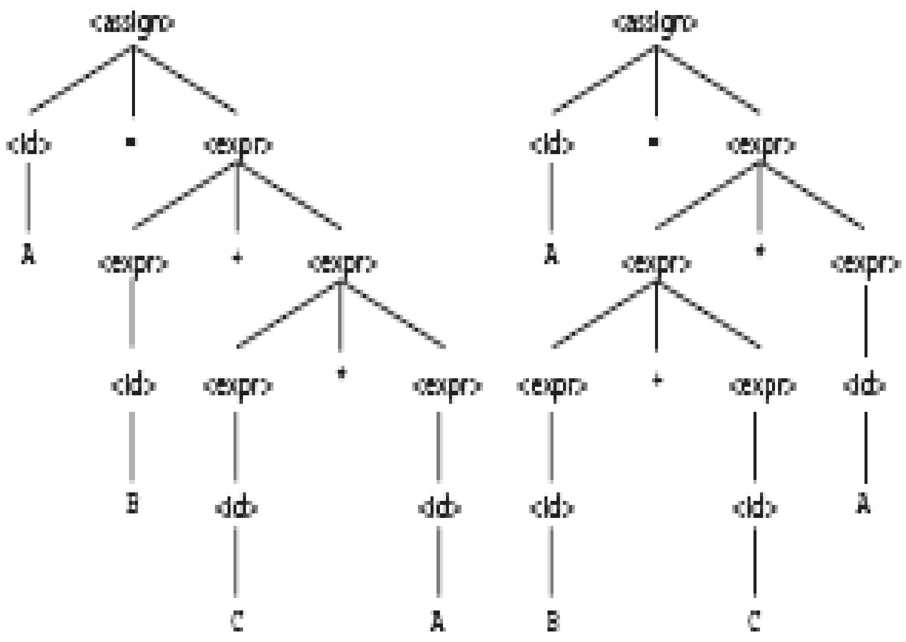
The grammar of Example 3.3 is ambiguous because the sentence

a = b + c * a

has two distinct parse trees, as shown in Figure 3.2.

Figure 3.2

Two distinct parse trees for the same sentence, A = B + C * A



If a language structure has more than one parse tree, then the meaning in two specific examples in the following subsections.
There are several other characteristics of a grammar that useful in determining whether a grammar is ambiguous.They include the following:
(1) if the grammar generates a sentence with more than one leftmost derivation and (2) if the grammar generates a sentence with more than one rightmost derivation. . In many cases, an ambiguous grammar can be rewritten to be unambiguous to generate the desired language.

Operator Precedence

When an expression includes two different operators, for example, x + y * z, For example, if * has been assigned higher precedence than + (by the language designer), multiplication will be done first later addition. If we use precedence levels of operators , we cannot have ambiguity.

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>
```

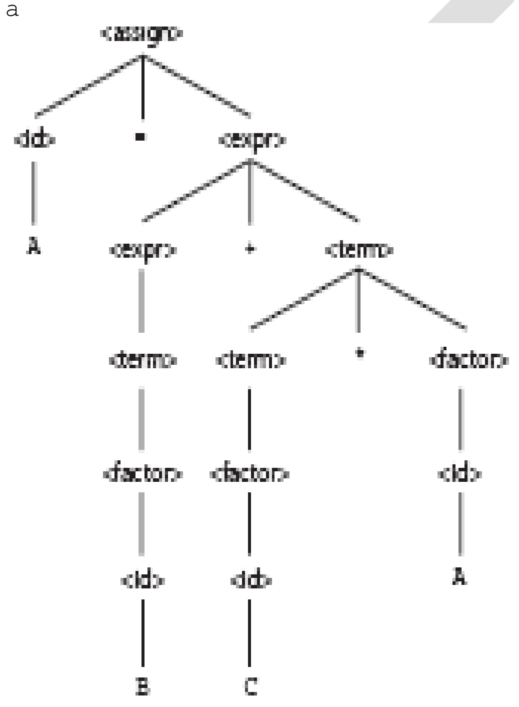
The following derivation of the sentence `a = b + c * a` uses the above grammar

<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
```

Figure 3.3

The unique parse tree for A = B + C * A using an unambiguous grammar

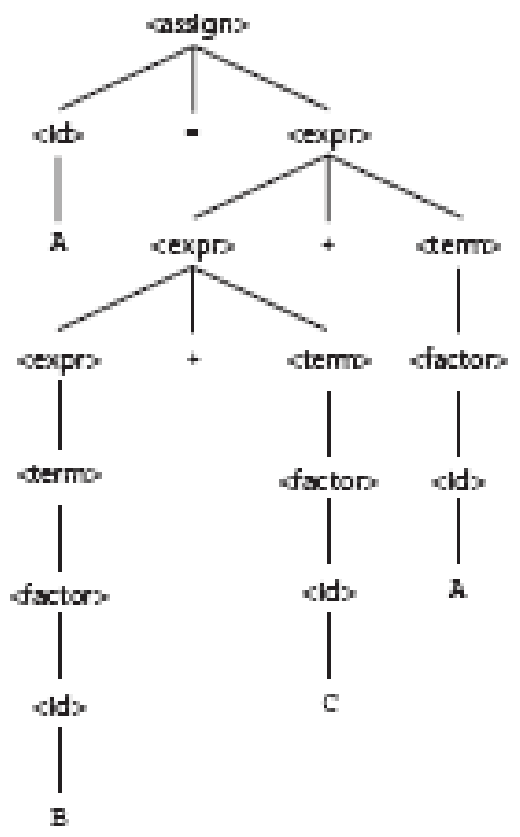


Associativity of Operators

Operator associativity can also be indicated by a grammar

Figure 3.4

A parse tree for `A = B + C + A` illustrating the associativity of addition



An Unambiguous Grammar for if-then-else

The BNF rules for an Ada **if-then-else** statement are as follows:

<if_stmt> → **if** <logic_expr> **then** <stmt>
 if <logic_expr> **then** <stmt> **else** <stmt>

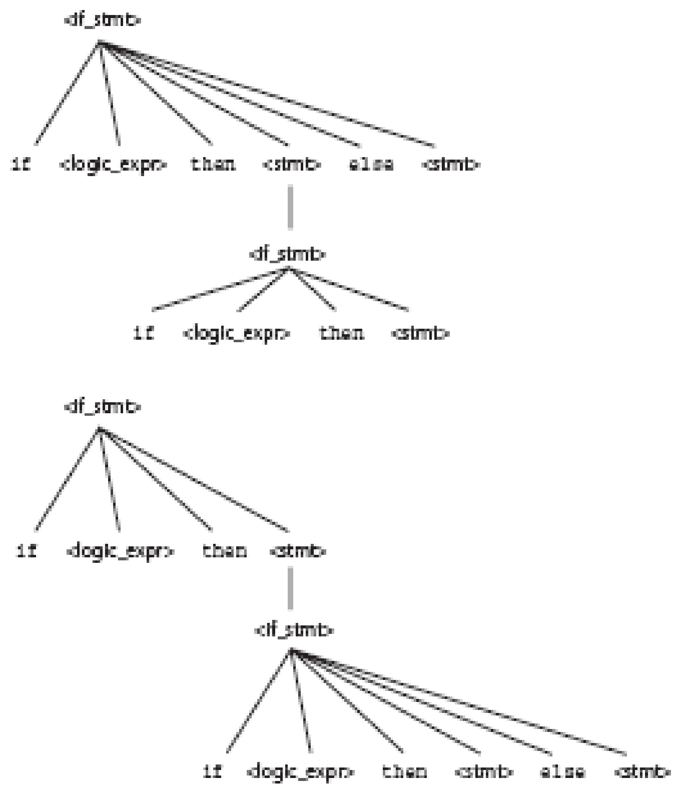
If we also have <stmt> → <if_stmt>, this grammar is ambiguous. The simplest sentential form that illustrates this ambiguity is

if <logic_expr> **then if** <logic_expr> **then** <stmt> **else** <stmt>

The two parse trees in Figure 3.5 show the ambiguity of this sentential form.

Figure 3.5

Two distinct parse trees for the same sentential form



Therefore, there cannot be an **if** statement without an **else** between a **then** and its matching **else**. So, for this situation, statements must be distinguished between those that are matched and those that are unmatched, where unmatched statements are **else**-less **if**s and all other statements are matched. The problem with the earlier grammar is that it treats all statements as if they had equal syntactic significance—that is, as if they were all matched.

To reflect the different categories of statements, different abstractions, or nonterminals, must be used. The unambiguous grammar based on these ideas follows:

```
<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>
```

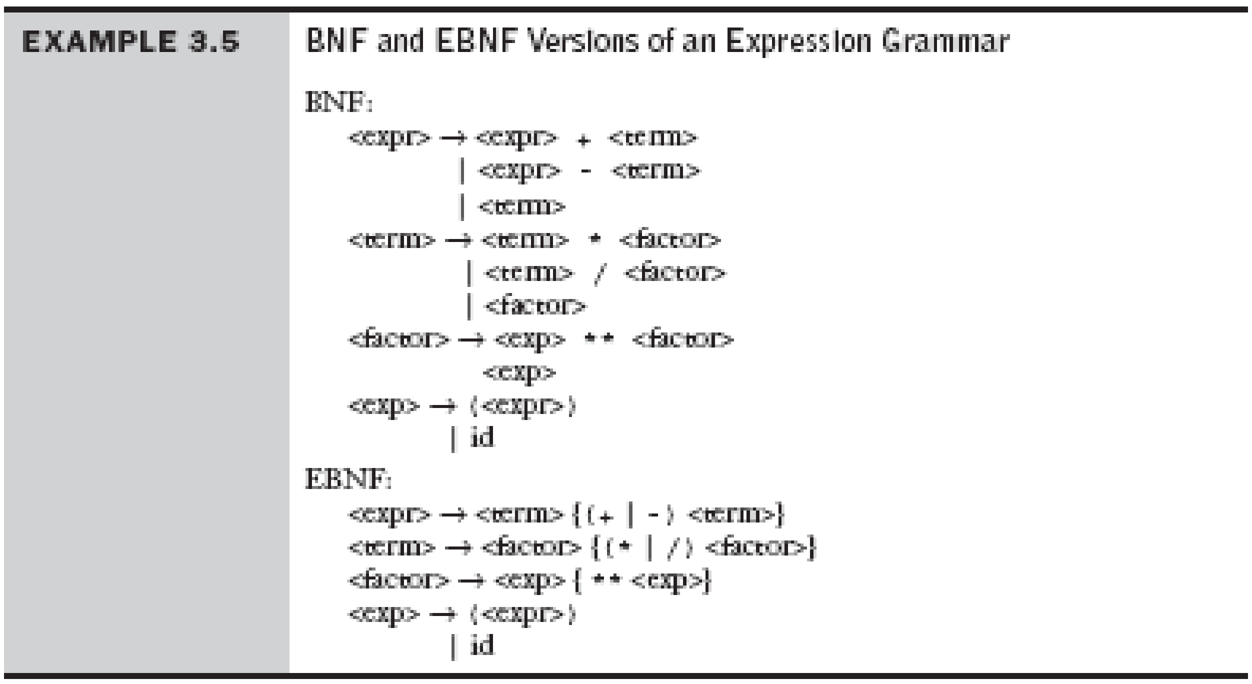
There is just one possible parse tree, using this grammar, for the following sentential form:

```
if <logic_expr> then if <logic_expr> then <stmt> else <stmt>
```

Extended BNF (EBNF)

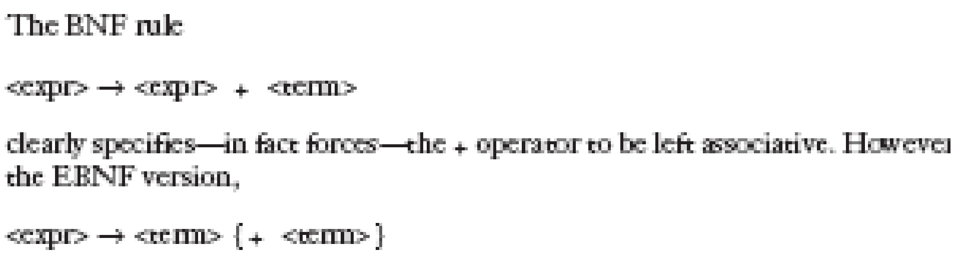
For example, a C **if-else** statement can be described in EBNF as
<if_stmt> –> **if** (<expression>) <statement> {**else** <statement>}
Without the use of the brackets, the syntactic description of this statement would require the following two rules:
<if_stmt> –> **if** (<expression>) <statement>
| **if** (<expression>) <statement> **else** <statement>

<ident_list> –> <identifier> {, <identifier>}



EXAMPLE 3.5 BNF and EBNF Versions of an Expression Grammar

```
BNF:
    <expr> → <expr> + <term>
           | <expr> - <term>
           | <term>
    <term> → <term> * <factor>
           | <term> / <factor>
           | <factor>
    <factor> → <exp> ** <factor>
             <exp>
    <exp> → (<expr>)
          | id
EBNF:
    <expr> → <term> {(+ | -) <term>}
    <term> → <factor> {(* | /) <factor>}
    <factor> → <exp> { ** <exp>}
    <exp> → (<expr>)
          | id
```



The BNF rule

<expr> → <expr> + <term>

clearly specifies—in fact forces—the + operator to be left associative. However the EBNF version,

<expr> → <term> {+ <term>}

Attribute grammar

An **attribute grammar** is a device used to describe more of the structure of a programming language than can be described with a context-free grammar. An attribute grammar is an extension to a context-free grammar. Attribute Grammars allows certain language rules to be conveniently described, such as type compatibility. Before we formally define the form of attribute grammars, we must clarify the concept of static semantics.

Static Semantics

There are some characteristics of the structure of programming languages that are difficult to describe with BNF, As an example of a syntax rule that is difficult to specify with BNF, consider type compatibility rules. As an example of a syntax rule that cannot be specified in BNF, consider the common rule that all variables must be declared before they are referenced. It has been proven that this rule cannot be specified in BNF. These problems exemplify the categories of language rules called static semantics rules. The **static semantics** of a language is only indirectly related to the meaning of programs during execution; Many static semantic rules of a language state its type constraints. **Static semantics is so named because the analysis required to check these specifications can be done at compile time**. Because of the problems of describing static semantics with BNF, a variety of more powerful mechanisms has been devised for that task. One such mechanism, attribute grammars, was designed by Knuth (1968) to describe both the syntax and the static semantics of programs. Attribute grammars are a formal approach both to describing and checking the correctness of the static semantics rules of a program.
Dynamic semantics, which is the meaning of expressions, statements, and program units

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 (the terminal and nonterminal symbols), 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. They are used to specify how attribute values are computed. **Predicate functions**, which state the static semantic rules of the language, are ssociated with grammar rules.

Attribute Grammars Defined

An attribute grammar is a grammar with the following additional features: • 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, respectively. **Synthesized attributes** are used to pass semantic information up a parse tree, while **inherited attributes** pass semantic information down and across a tree.

Let X_0 -> X_1 ... X_n be a rule.

Functions of the form $S(X_0) = f(A(X_1), \ldots A(X_n))$ define *synthesized attributes*

Functions of the form $I(X_j) = f(A(X_0), \ldots , A(X_n))$, for $i \le j \le n$, define *inherited attributes*

. If all the attribute values in a parse tree have been computed, the tree is said to be **fully attributed**.

Intrinsic attributes are synthesized attributes of leaf nodes whose values are determined outside the parse tree. For example, the type of an instance of a variable in a program could come from the symbol table, which is used to store variable names . The attribute grammar specifies these static semantic rules.

The syntax portion of our example(3.6) attribute grammar is

```
<assign> → <var> = <expr>
<expr>   → <var> + <var>
           | <var>
<var> → a | b | c
```

The attributes for the nonterminals in the example attribute grammar are described in the following paragraphs:

• *actual_type*—A synthesized attribute associated with the nonterminals <var> and <expr>. It is used to store the actual type, int or real, of a variable or expression. In the case of a variable, the actual type is intrinsic. In the case of an expression, it is determined from the actual types of the child node or children nodes of the <expr> nonterminal.

• *expected_type*—An inherited attribute associated with the nonterminal <expr>. It is used to store the type, either int or real, that is expected for the expression, as determined by the type of the variable on the left side of the assignment statement.

The complete attribute grammar follows in Example 3.6.

EXAMPLE 3.6 **An Attribute Grammar for Simple Assignment Statements**

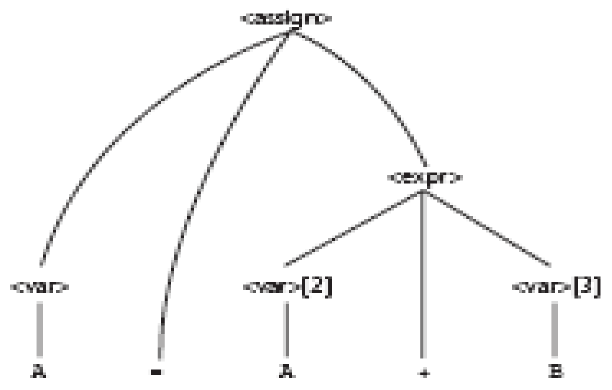
```
1. Syntax rule: <assign> → <var> = <expr>
Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
Semantic rule: <expr>.actual_type ←
                        if (<var>[2].actual_type = int) and
                            (<var>[3].actual_type = int)
                            then int
                            else real
                            end if
Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
Semantic rule: <expr>.actual_type ← <var>.actual_type
Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → a | b | c
Semantic rule: <var>.actual_type ← look-up(<var>.string)
```

The look-up function looks up a given variable name in the symbol table and

returns the variable's type.
A parse tree of the sentence a = a + b generated by the grammar in Example 3.6 is shown in Figure 3.6.

Figure 3.6

A parse tree for
A = A + B



As in the grammar, bracketed numbers are added after the repeated node labels in the tree so they can be referenced unambiguously.Consider the process of computing the attribute values of a parse tree, which is sometimes called **decorating** the parse tree. If all attributes were inherited, this could proceed in a completely top-down order, from the
root to the leaves. Alternatively, it could proceed in a completely bottomup order, from the leaves to the root, if all the attributes were synthesized. Because our grammar has both synthesized and inherited attributes, the evaluation process cannot be in any single direction. The following is an evaluation of the attributes, in an order in which it is possible to compute them:
1. <var>.actual_type ← look-up(A) (Rule 4)
2. <expr>.expected_type ← <var>.actual_type (Rule 1)
3. <var>[2].actual_type ← look-up(A) (Rule 4)
<var>[3].actual_type ← look-up(B) (Rule 4)
4. <expr>.actual_type ← either int or real (Rule 2)
5. <expr>.expected_type == <expr>.actual_type is either TRUE or FALSE (Rule 2)
The tree in Figure 3.7 shows the flow of attribute values in the example of Figure 3.6. Solid lines are used for the parse tree; dashed lines show attribute flow in the tree.
The tree in Figure 3.8 shows the final attribute values on the nodes. In this example, a is defined as a real and b is defined as an int.

Figure 3.7

The flow of attributes in the tree

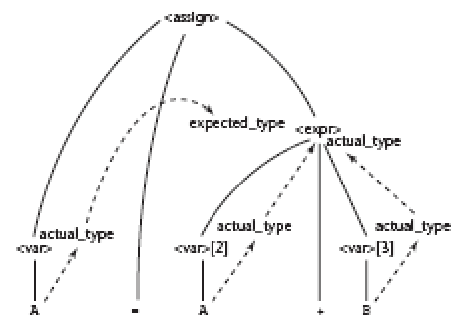
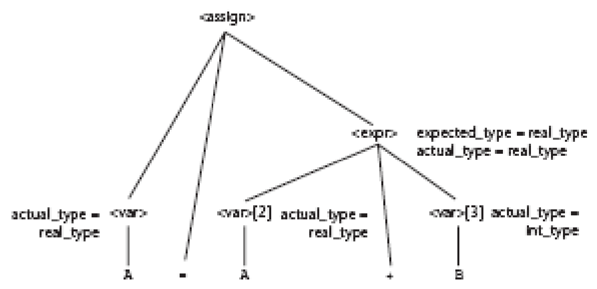


Figure 3.8

A fully attributed parse tree



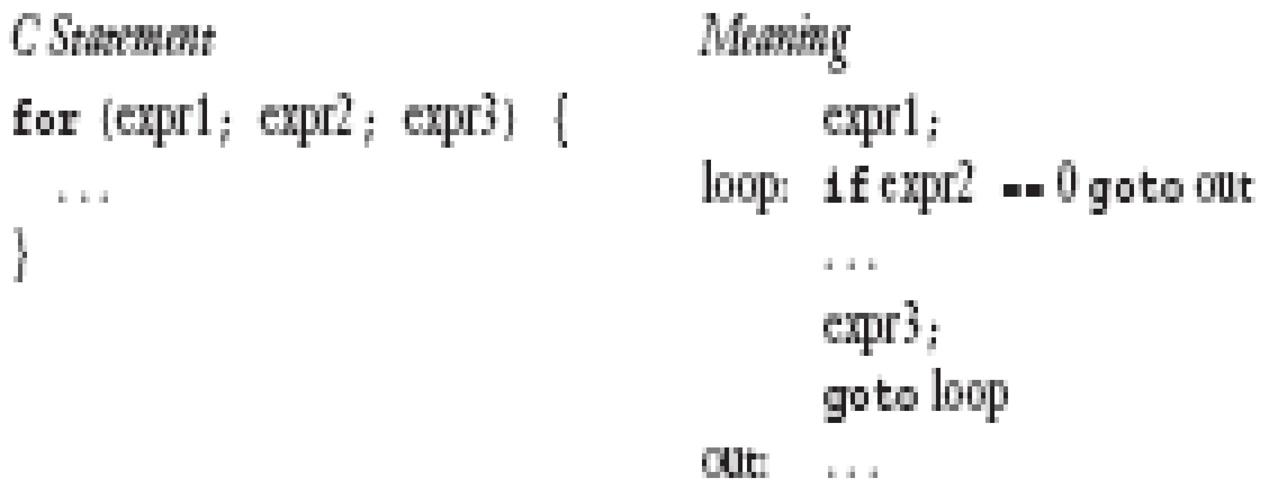
Dynamic Semantics

describing meaning of the expressions, statements, and program units of a programming language.
Dynamic semantics are of 3 types

1.Operational Semantics
2. Axiomatic semantics
3. Denotational Semantics

Operational semantics describing the meaning of program by translating it into more easily understandable form
-The idea behind **operational semantics** is to describe the meaning of a statement or program by specifying the effects of running it on a machine.The effects on the machine are viewed as the sequence of changes in its state, where the machine's state is the collection of the values in its storage.

For example, the semantics of the C **for** construct can be described in terms of simpler statements, as in



C Statement	*Meaning*
for (expr1; expr2; expr3) {	expr1;
...	loop: if expr2 == 0 goto out
}	...
	expr3;
	goto loop
	out: ...

- *Evaluation of operational semantics:*
  - Good if used informally
  - Extremely complex if used formally (e.g., VDL)

Axiomatic semantics was defined in conjunction with the development of an approach to roving the correctness of programs. In a proof, each statement of a program is both preceded and followed by a logical expression that specifies constraints on program variables. These, rather than the entire state of an abstract machine (as with operational semantics), are used to specify the meaning of the statement. Axiomatic semantics are based on mathematical logic.

Assertions

The logical expressions used in axiomatic semantics are called predicates, or **assertions**. An assertion immediately preceding a program statement describes the constraints on the program variables at that point in the program. An assertion immediately following a statement describes the new constraints on those variables (and possibly others) after execution of the statement. These assertions are called the **precondition** and **postcondition**, respectively, of the statement. For two adjacent statements, the postcondition of the first serves as the precondition of the second. Developing an axiomatic description or proof of a given program requires that every statement in the program has both a precondition and a postcondition. We assume all variables are integer type.
As a simple example, consider the following assignment statement and postcondition:
sum = 2 * x + 1 {sum > 1}

One possible precondition for this statement is {x > 10}.
In axiomatic semantics, the meaning of a specific statement is defined by its precondition and its post condition.

Weakest Preconditions

The **weakest precondition** is the least restrictive precondition that will guarantee the validity of the associated postcondition. For example, in the statement and postcondition given in Section 3.5.3.1, {x > 10}, {x > 50}, and {x > 1000} are all valid preconditions. The weakest of all preconditions in this case 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. The general form of an inference rule is as follows:
S1, S2, ………. S*n*

S

This rule states that if S1, S2, . . . , and S*n* are true, then the truth of S can be inferred. The top part of an inference rule is called its **antecedent**; the bottom part is called its **consequent**.
An **axiom** is a logical statement that is assumed to be true. Therefore, an axiom is an inference rule without an antecedent.

Assignment Statements

The precondition and postcondition of an assignment statement together define precisely its meaning. To define the meaning of an assignment statement, given a postcondition, there must be a way to compute its precondition from that postcondition.
Let x = E be a general assignment statement and Q be its postcondition.
Then, its precondition, P, is defined by the axiom
$P = Q_{x->E}$
which means that P is computed as Q with all instances of x replaced by E. For example, if we have the assignment statement and postcondition
a = b / 2 - 1 {a < 10}
the weakest precondition is computed by substituting b / 2 - 1 for a in the postcondition
{a < 10}, as follows:
b / 2 - 1 < 10
b < 22
Thus, the weakest precondition for the given assignment statement and postcondition
is {b < 22}.
The usual notation for specifying the axiomatic semantics of a given statement
form is
{P}S{Q}
As another example of computing a precondition for an assignment statement,
consider the following:
x = 2 * y - 3 {x > 25}
The precondition is computed as follows:
2 * y - 3 > 25
y > 14
So {y > 14} is the weakest precondition for this assignment statement and
postcondition.

Sequences

The weakest precondition for a sequence of statements cannot be described by an axiom, because the precondition depends on the particular kinds of statements in the sequence. In this case, the precondition can only be described with an inference rule. Let S1 and S2 be adjacent program statements. If S1 and S2 have the following pre- and postconditions
{P1} S1 {P2}
{P2} S2 {P3}
the inference rule for such a two-statement sequence is

{P1} S1 {P2}, {P2} S2 {P3}

{P1} S1, S2 {P3}

So, for our example, {P1} S1; S2 {P3} describes the axiomatic semantics of the sequence S1; S2. The inference rule states that to get the sequence precondition, the precondition of the second statement is computed. This new assertion is then used as the postcondition of the first statement, which can then be used to compute the precondition of the first statement, which is also the precondition of the whole sequence.

Selection

We consider the inference rule for selection statements, the general form of which is
if B **then** S1 **else** S2
We consider only selections that include **else** clauses. The inference rule is

{B and P} S1 {Q}, {(not B) and P} S2{Q}

{P} **if** B **then** S1 **else** S2 {Q}

This rule indicates that selection statements must be proven both when the Boolean control expression is true and when it is false. The first logical statement above the line represents the **then** clause; the second represents the **else** clause. According to the inference rule, we need a precondition P that can be used in the precondition of both the **then** and **else** clauses.

Consider the following example of the computation of the precondition using the selection inference rule. The example selection statement is
if x > 0 **then**
y = y - 1
else
y = y + 1
Suppose the postcondition, Q, for this selection statement is {y > 0}. We can use the axiom for assignment on the **then** clause
y = y - 1 {y > 0}
This produces {y - 1 > 0} or {y > 1}. It can be used as the P part of the precondition for the **then** clause. Now we apply the same axiom to the **else** clause
y = y + 1 {y > 0}
which produces the precondition {y + 1 > 0} or {y > -1}.
Because {y > 1} => {y > -1}, the rule of consequence allows us to
use {y > 1} for the precondition of the whole selection statement.

Logical Pretest Loops

Another essential construct of imperative programming languages is the logical pretest, or **while** loop. Computing the weakest precondition for a **while** loop is inherently more difficult than for a sequence, because the number of iterations cannot always be predetermined. In a case where the number of iterations is known, the loop can be unrolled and treated as a sequence. The corresponding step in the axiomatic semantics of a **while** loop is finding an assertion called a **loop invariant**, which is crucial to finding the weakest precondition.

The inference rule for computing the precondition for a **while** loop is

{I and B} S {I}

{I} **while** B **do** S **end** {I and (not B)}

where I is the loop invariant. This seems simple, but it is not. The complexity lies in finding an appropriate loop invariant.

The axiomatic description of a **while** loop is written as

{P} **while** B **do** S **end** {Q}

The loop invariant must satisfy a number of requirements to be useful. First, the weakest precondition for the **while** loop must guarantee the truth of the loop invariant. In turn, the loop invariant must guarantee the truth of the post condition upon loop termination. These constraints move us from the inference rule to the axiomatic description. During execution of the loop, the truth of the loop invariant must be unaffected by the evaluation of the loop controlling. Boolean expression and the loop body statements. Hence, the name *invariant.*
Another complicating factor for **while** loops is the question of loop termination. A loop that does not terminate cannot be correct, and in fact computes nothing. If Q is the post condition that holds immediately after loop exit, then a precondition P for the loop is one that guarantees Q at loop exit and also guarantees that the loop terminates. The complete axiomatic description of a **while** construct requires all of the following to be true, in which I is the loop invariant:

P => I
{I and B} S {I}
(I and (not B)) => Q
the loop terminates

As example of finding a loop invariant using the approach used in mathematical induction, consider the following loop statement:
while s > 1 **do** s = s / 2 **end** {s = 1}
As before, we use the assignment axiom to try to find a loop invariant and a precondition for the loop. For zero iterations, the weakest precondition is {s = 1}. For one iteration, it is
wp(s = s / 2, {s = 1}) = {s / 2 = 1}, or {s = 2}
For two iterations, it is
wp(s = s / 2, {s = 2}) = {s / 2 = 2}, or {s = 4}
For three iterations, it is
wp(s = s / 2, {s = 4}) = {s / 2 = 4}, or {s = 8}
From these cases, we can see clearly that the invariant is {s is a nonnegative power of 2}

Denotational semantics

Denotational semantics most widely known formal method for describing the meaning of programs. In D.S, the syntactic entities are mapped into mathematical objects with concrete meaning

The process of constructing a denotational semantics specification for a programming language requires one to define for each language entity both a mathematical object and a function that maps instances of that language entity onto instances of the mathematical object.

The method is named *denotational* because the mathematical objects denote the meaning of their corresponding syntactic entities.

Two Simple Examples

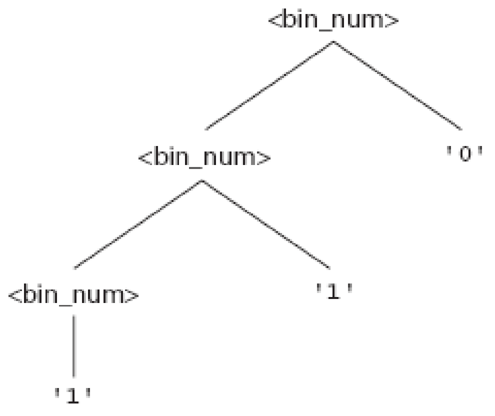
We use a very simple language construct, character string representations of binary numbers, to introduce the denotational method. The syntax of such binary numbers can be described by the following grammar rules:

```
<bin_num> -> '0'
        | '1'
        | <bin_num> '0'
        | <bin_num> '1'
```

A parse tree for the example binary number, 110, is shown in Figure 3.9. Notice that we put apostrophes around the syntactic digits to show they are not mathematical digits. This is similar to the relationship between ASCII coded digits and mathematical digits. When a program reads a number as a string, it must be converted to a mathematical number before it can be used as a value in the program.

Figure 3.9

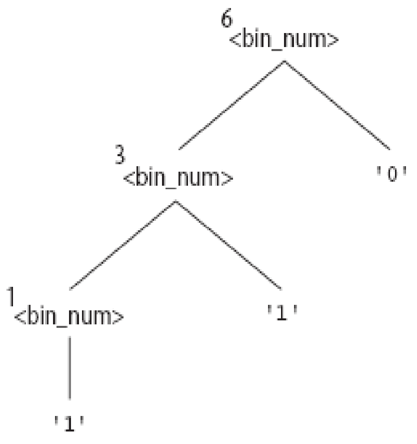
A parse tree of the binary number 110



The semantic function, named Mbin, maps the syntactic objects, as described in the previous grammar rules, to the objects in N, the set of nonnegative decimal numbers. The function Mbin is defined as follows:
Mbin('0') = 0
Mbin('1') = 1
Mbin(<bin_num> '0') = 2 * Mbin(<bin_num>)
Mbin(<bin_num> '1') = 2 * Mbin(<bin_num>) + 1
The meanings, or denoted objects (which in this case are decimal numbers), can be attached to the nodes of the parse tree shown on the previous page, yielding the tree in Figure 3.10. This is syntax-directed semantics. Syntactic entities are mapped to mathematical objects with concrete meaning.

Figure 3.10

A parse tree with denoted objects for 110



Grammar for describing Decimal numbers as follows

<dec_num> -> '0' | '1' | '2' |'3' | '4' | '5' | '6' | '7' | '8' |'9'
|<dec_num> ('0' | '1' | '2' | '3' |'4' | '5' | '6' | '7' | '8' | '9')
The denotational mappings for these syntax rules are
Mdec('0') = 0, Mdec('1') = 1, Mdec('2') = 2, . . ., Mdec('9') = 9
Mdec(<dec_num> '0') = 10 * Mdec(<dec_num>)
Mdec(<dec_num> '1') = 10 * Mdec(<dec_num>) + 1
. . .
Mdec(<dec_num> '9') = 10 * Mdec(<dec_num>) + 9

The State of a Program

Let the state s of a program be represented as a set of ordered pairs, as follows:
s = {<i1, v1>, <i2, v2>, . . . , <i*n*, v*n*>}
Each i is the name of a variable, and the associated v's are the current values of those variables

Lexical analysis

A lexical analyzer is essentially a pattern matcher. A pattern matcher attempts to find a substring of a given string of characters that matches a given character pattern. Pattern matching is a traditional part of computing. Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens. In the early days of compilers, lexical analyzers often processed an entire ource program file and produced a file of tokens and lexemes. Now, however, most lexical analyzers are subprograms that locate the next lexeme in the input, determine its associated token code, and return them to the caller, which is the syntax analyzer. So, each call to the lexical analyzer returns a single lexeme and its token. The only view of the input program seen by the syntax analyzer is the output of the lexical analyzer, one token at a time. The lexical-analysis process includes skipping comments and white space outside lexemes, as they are not relevant to the meaning of the program. Also,the lexical analyzer inserts lexemes for user-defined names into the symbol table, which is used by later phases of the compiler. Finally, lexical analyzers detect syntactic errors in tokens, such as ill-formed floating-point literals, and report such errors to the user.
Consider the following example of an assignment statement:

```
result = oldsum – value / 100;
```

Following are the tokens and lexemes of this statement:

Token	*Lexeme*
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

There are three approaches to building a lexical analyzer:

1. Write a formal description of the token patterns of the language using a descriptive language related to regular expressions.1 These descriptions are used as input to a software tool that automatically generates a lexical analyzer. There are many such tools available for this. The oldest of these, named lex, is commonly included as part of UNIX systems.
2. Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.
3. Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.

The following is a C implementation of a lexical analyzer specified in the state diagram of Figure 4.1, including a main driver function for testing purposes:

```
/* front.c - a lexical analyzer system for simple
arithmetic expressions */
#include <stdio.h>
```

```
#include <ctype.h>
/* Global declarations */
/* Variables */
int charClass;
char lexeme [100];
char nextChar;
int lexLen;
int token;
int nextToken;
FILE *in_fp, *fopen();
4.2 Lexical Analysis 173
/* Function declarations */
void addChar();
void getChar();
void getNonBlank();
int lex();
/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99
/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
/******************************************************/
/* main driver */
main() {
/* Open the input data file and process its contents */
if ((in_fp = fopen("front.in", "r")) == NULL)
printf("ERROR - cannot open front.in \n");
else {
getChar();
do {
lex();
} while (nextToken != EOF);
}
}
/*****************************************************/
/* lookup - a function to lookup operators and parentheses
and return the token */
int lookup(char ch) {
switch (ch) {
case '(':
addChar();
nextToken = LEFT_PAREN;
break;
case ')':
addChar();
nextToken = RIGHT_PAREN;
break;
```

```
case '+':
addChar();
nextToken = ADD_OP;
break;
case '-':
addChar();
nextToken = SUB_OP;
break;
case '*':
addChar();
nextToken = MULT_OP;
break;
```

4.2 Lexical Analysis **175**

```
case '/':
addChar();
nextToken = DIV_OP;
break;
default:
addChar();
nextToken = EOF;
break;
}
return nextToken;
}
/*****************************************************/
/* addChar - a function to add nextChar to lexeme */
void addChar() {
if (lexLen <= 98) {
lexeme[lexLen++] = nextChar;
lexeme[lexLen] = 0;
}
else
printf("Error - lexeme is too long \n");
}
/*****************************************************/
/* getChar - a function to get the next character of
input and determine its character class */
void getChar() {
if ((nextChar = getc(in_fp)) != EOF) {
if (isalpha(nextChar))
charClass = LETTER;
else if (isdigit(nextChar))
charClass = DIGIT;
else charClass = UNKNOWN;
}
else
charClass = EOF;
}
/*****************************************************/
/* getNonBlank - a function to call getChar until it
returns a non-whitespace character */
void getNonBlank() {
while (isspace(nextChar))
getChar();
}
```

176 Chapter 4 Lexical and Syntax Analysis

```
/
*****************************************************/
/* lex - a simple lexical analyzer for arithmetic
expressions */
int lex() {
lexLen = 0;
getNonBlank();
switch (charClass) {
/* Parse identifiers */
case LETTER:
addChar();
getChar();
while (charClass == LETTER || charClass == DIGIT) {
addChar();
getChar();
}
nextToken = IDENT;
break;
/* Parse integer literals */
case DIGIT:
addChar();
getChar();
while (charClass == DIGIT) {
addChar();
getChar();
}
nextToken = INT_LIT;
break;
/* Parentheses and operators */
case UNKNOWN:
lookup(nextChar);
getChar();
break;
/* EOF */
case EOF:
nextToken = EOF;
lexeme[0] = 'E';
lexeme[1] = 'O';
lexeme[2] = 'F';
lexeme[3] = 0;
break;
} /* End of switch */
```

4.3 The Parsing Problem **177**

```
printf("Next token is: %d, Next lexeme is %s\n",
nextToken, lexeme);
return nextToken;
} /* End of function lex */
```

This code illustrates the relative simplicity of lexical analyzers.

Consider the following expression:

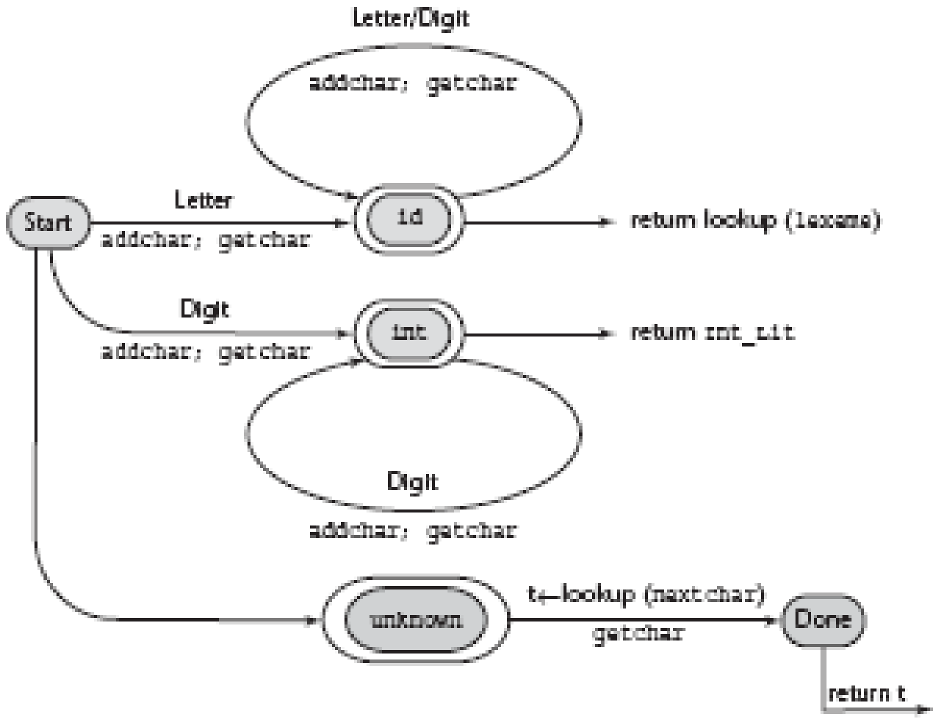
```
(sum + 47) / total
```

Following is the output of the lexical analyzer of `front.c` when used on this expression:

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF
```

Figure 4.1

A state diagram to recognize names, parentheses, and arithmetic operators



Recursive Descent Parsing

Consider the following EBNF description of simple arithmetic expressions:
<expr> –> <term> {(+ | –) <term>}
<term> □ <factor> {(* | /) <factor>}
<factor> □ id | int_constant | (<expr>)
The recursive-descent subprogram for the first rule in the previous example grammar, written in C, is

```
/* expr
Parses strings in the language generated by the rule:
<expr> -> <term> {(+ | -) <term>}
*/
void expr() {
printf("Enter <expr>\n");
/* Parse the first term */
term();
/* As long as the next token is + or -, get
the next token and parse the next term */
while (nextToken == ADD_OP || nextToken == SUB_OP) {
```

```
lex();
term();
}
printf("Exit <expr>\n");
} /* End of function expr */

/* term
Parses strings in the language generated by the rule:
<term> -> <factor> {(* | /) <factor>)
*/
void term() {
printf("Enter <term>\n");
/* Parse the first factor */
factor();
/* As long as the next token is * or /, get the
next token and parse the next factor */
while (nextToken == MULT_OP || nextToken == DIV_OP) {
lex();
factor();
}
printf("Exit <term>\n");
} /* End of function term */
/* factor
Parses strings in the language generated by the rule:
<factor> -> id | int_constant | ( <expr )
*/
void factor() {
printf("Enter <factor>\n");
/* Determine which RHS */
if (nextToken == IDENT || nextToken == INT_LIT)
/* Get the next token */
lex();
/* If the RHS is ( <expr>), call lex to pass over the
left parenthesis, call expr, and check for the right
parenthesis */
else {
if (nextToken == LEFT_PAREN) {
lex();
expr();
if (nextToken == RIGHT_PAREN)
lex();
else
error();
} /* End of if (nextToken == ... */
/* It was not an id, an integer literal, or a left
parenthesis */
else
error();
} /* End of else */
printf("Exit <factor>\n");;
} /* End of function factor */
```

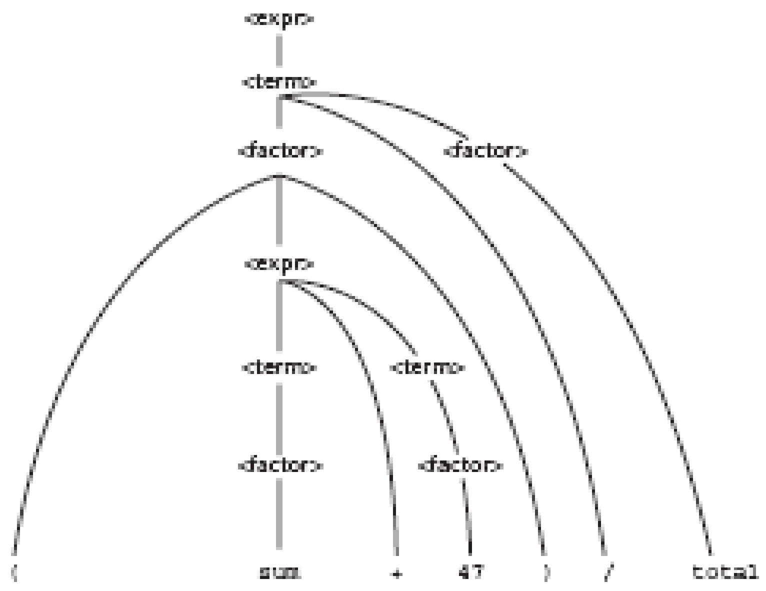
Following is the trace of the parse of the example expression `(sum + 47) / total`, using the parsing functions `expr`, `term`, and `factor`, and the function `lex` from Section 4.2. Note that the parse begins by calling `lex` and the start symbol routine, in this case, `expr`.

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
Next token is: 11 Next lexeme is total
Enter <factor>
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

The parse tree traced by the parser for the preceding expression is shown in Figure 4.2.

Figure 4.2

Parse tree for (sum + 47) / total



Following is a grammatical description of the Java **if** statement:
<ifstmt> ->**if** (<boolexpr>) <statement> [**else** <statement>]
The recursive-descent subprogram for this rule follows:

```
/* Function ifstmt
Parses strings in the language generated by the rule:
<ifstmt> -> if (<boolexpr>) <statement>
[else <statement>]
*/
void ifstmt() {
/* Be sure the first token is 'if' */
if (nextToken != IF_CODE)
error();
else {
/* Call lex to get to the next token */
lex();
/* Check for the left parenthesis */
if (nextToken != LEFT_PAREN)
error();
else {
/* Call boolexpr to parse the Boolean expression */
boolexpr();
/* Check for the right parenthesis */
if (nextToken != RIGHT_PAREN)
error();
else {
/* Call statement to parse the then clause */
statement();
/* If an else is next, parse the else clause */
if (nextToken == ELSE_CODE) {
/* Call lex to get over the else */
lex();
statement();
} /* end of if (nextToken == ELSE_CODE ... */
} /* end of else of if (nextToken != RIGHT ... */
} /* end of else of if (nextToken != LEFT ... */
} /* end of else of if (nextToken != IF_CODE ... */
} /* end of ifstmt */
```