

Chapter 3

Describing Syntax and Semantics

3.1 Introduction 110

3.2 The General Problem of Describing Syntax 111

3.3 Formal Methods of Describing Syntax 113

3.4 Attribute Grammars 128

3.5 Describing the Meanings of Programs: Dynamic Semantics 134

Summary • Bibliographic Notes • Review Questions • Problem Set 155

Chapter 3

Describing Syntax and Semantics

3.1 Introduction 110

- **Syntax** – the **form** of the expressions, statements, and program units
- **Semantics** - the **meaning** of the expressions, statements, and program units.
- Ex: the syntax of a Java while statement is

```
while (boolean_expr) statement
```

- The semantics of this statement form is that when the current value of the Boolean expression is true, the embedded statement is executed.
- The form of a statement should strongly suggest what the statement is meant to accomplish.

3.2 The General Problem of Describing Syntax 111

- A **sentence** or “**statement**” is a string of characters over some alphabet. The syntax rules of a language specify which strings of characters from the language’s alphabet are in the language.
- A **language** is a set of sentences.
- A **lexeme** is the lowest level syntactic unit of a language. It includes identifiers, literals, operators, and special word (e.g. *, sum, begin). A **program** is strings of lexemes.
- A **token** is a category of lexemes (e.g., identifier). An identifier is a token that have lexemes, or instances, such as sum and total.
- Ex:

```
index = 2 * count + 17;
```

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

Language Recognizers and Generators

- In general, language can be formally defined in two distinct ways: by recognition and by generation.
- **Language Recognizers:**
 - A recognition device reads input strings of the language and decides whether the input strings belong to the language.
 - It only determines whether given programs are in the language.
 - Example: syntax analyzer part of a compiler. The syntax analyzer, also known as **parsers**, determines whether the given programs are syntactically correct.
- **Language Generators:**
 - A device that generates **sentences** of a language
 - One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator

3.3 Formal Methods of Describing Syntax 113

- The formal language generation mechanisms are usually called **grammars**
- Grammars are commonly used to describe the **syntax** of programming languages.

3.3.1 Backus-Naur Form and Context-Free Grammars

- It is a syntax description formalism that became the **most widely** used method for programming language syntax.

3.3.1.1 Context-free Grammars

- Developed by Noam Chomsky in the mid-1950s who described four classes of generative devices or grammars that define four classes of languages.
- Context-free and regular grammars are useful for describing the syntax of programming languages.
- Tokens of programming languages can be described by regular grammars.
- Whole programming languages can be described by context-free grammars.

3.3.1.2 Origins of Backus-Naur Form (1959)

- Invented by John Backus to describe ALGOL 58 syntax.
- **BNF** (Backus-Naur Form) is equivalent to context-free grammars used for describing syntax.

3.3.1.3 Fundamentals

- A metalanguage is a language used to describe another language. **BNF** is a metalanguage for programming language.
- In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called nonterminal symbols)

`<assign> → <var> = <expression>`

- This is a **rule**; it describes the structure of an assignment statement
- A rule has a left-hand side (**LHS**) “The abstraction being defined” and a right-hand side (**RHS**) “consists of some mixture of tokens, lexemes and references to other abstractions”, and consists of terminal and nonterminal symbols.
- 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>`.
- Example:

`total = subtotal1 + subtotal2`

- A grammar is a finite nonempty set of rules and the abstractions are called nonterminal symbols, or simply **nonterminals**.
- The lexemes and tokens of the rules are called terminal symbols or **terminals**.

- A BNF description, or grammar, is simply **a collection of rules**.
- An abstraction (or nonterminal symbol) can have more than one RHS
- For Example, a Java if statement can be described with the rule

```
<if_stmt> → if (<logic_expr>) <stmt>
          | if (<logic_expr>) <stmt> else <stmt>
```

- Multiple definitions can be written as a single rule, with the different definitions separated by the symbol |, meaning logical **OR**.

3.3.1.4 Describing Lists

- Syntactic lists are described using recursion.

```
<ident_list> → identifier
             | identifier, <ident_list>
```

- A rule is **recursive** if its LHS appears in its RHS.

3.3.1.5 Grammars and Derivations

- 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**.
- A **derivation** is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)
- Example 3.1 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>
```

- An example derivation for a 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
```

- Every string of symbols in the derivation, including <program>, is a sentential form.
- A sentence is a sentential form that has **only** terminal symbols.

- A **leftmost derivation** is one in which the leftmost nonterminal in each sentential form is the one that is expanded. The derivation continues until the sentential form contains no nonterminals.
- A derivation may be neither leftmost nor rightmost.
- Example 3.2 A Grammar for a Small Assignment Statements:

```

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

```

- An example derivation for the assignment $A = B * (A + C)$

```

<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)

```

3.3.1.6 Parse Trees

- Hierarchical structures of the language are called **parse trees**.
- A parse tree for the simple statement $A = B * (A + C)$

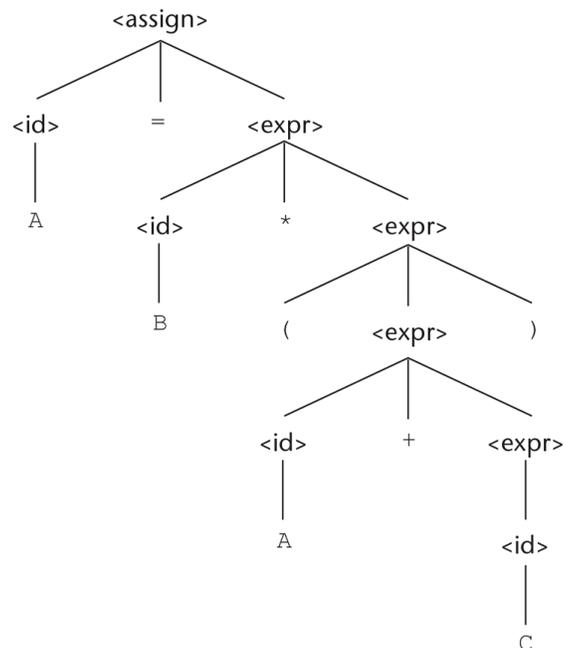


Figure 3.1 A parse tree for the simple statement $A = B * (A + C)$

3.3.1.7 Ambiguity

- A grammar is ambiguous if it generates a sentential form that has **two or more** distinct parse trees.
- Example 3.3 An Ambiguous Grammar for Small Assignment Statements
- Two distinct parse trees for the same sentence, **A = B + C * A**

```

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

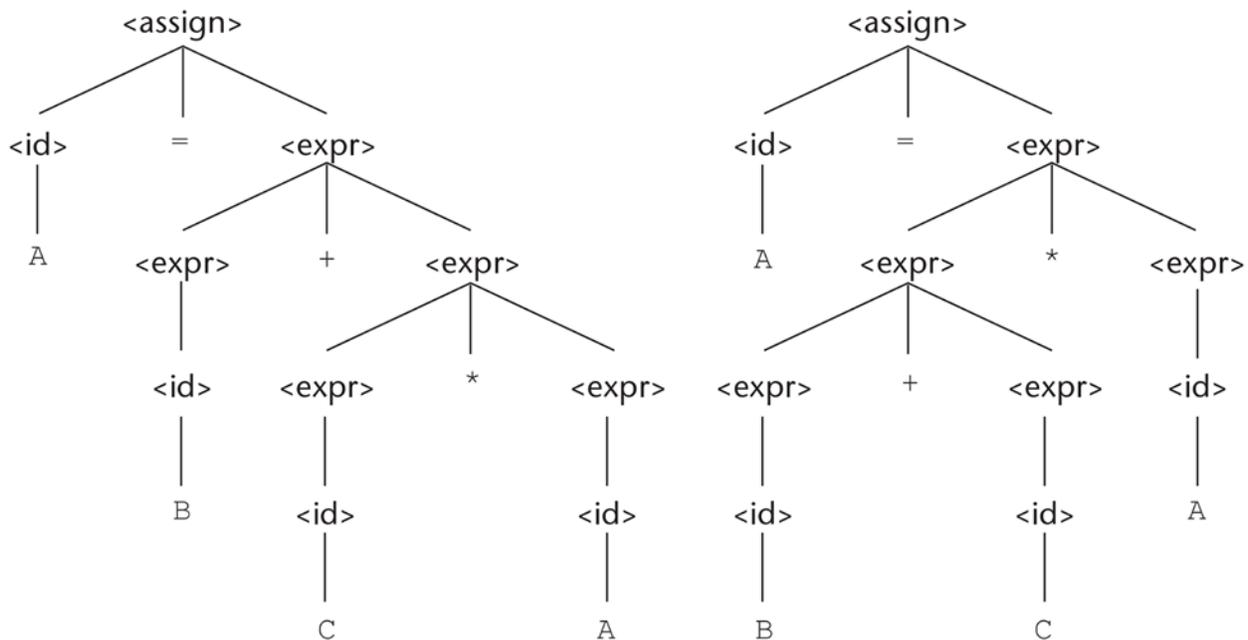


Figure 3.2 Two distinct parse trees for the same sentence, $A = B + C * A$

3.3.1.8 Operator Precedence

- The fact that an operator in an arithmetic expression is generated **lower** in the parse tree can be used to indicate that it has **higher precedence** over an operator produced higher up in the tree.
- In the left parsed tree above, one can conclude that the $*$ operator has precedence over the $+$ operator. How about the tree on the right hand side?

- 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>

```

- **Leftmost derivation** of the sentence **A = B + C * A**

```

<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

```

A parse tree for the simple statement, $A = B + C * A$

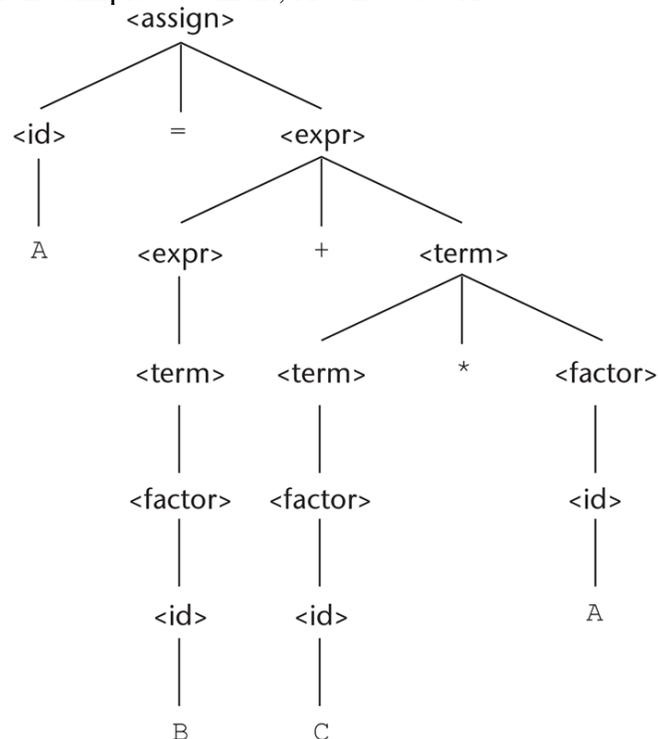


Figure 3.3 The unique parse tree for $A = B + C * A$ using an unambiguous grammar

- **Rightmost derivation** of the sentence $A = B + C * A$

```
<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
```

- **Both** of these derivations, however, are represented by the **same** parse tree.

3.3.1.9 Associativity of Operators

- Do parse trees for expressions with two or more adjacent occurrences of operators with **equal precedence** have those occurrences in proper hierarchical order?
- An example of an assignment using the previous grammar is: $A = B + C + A$

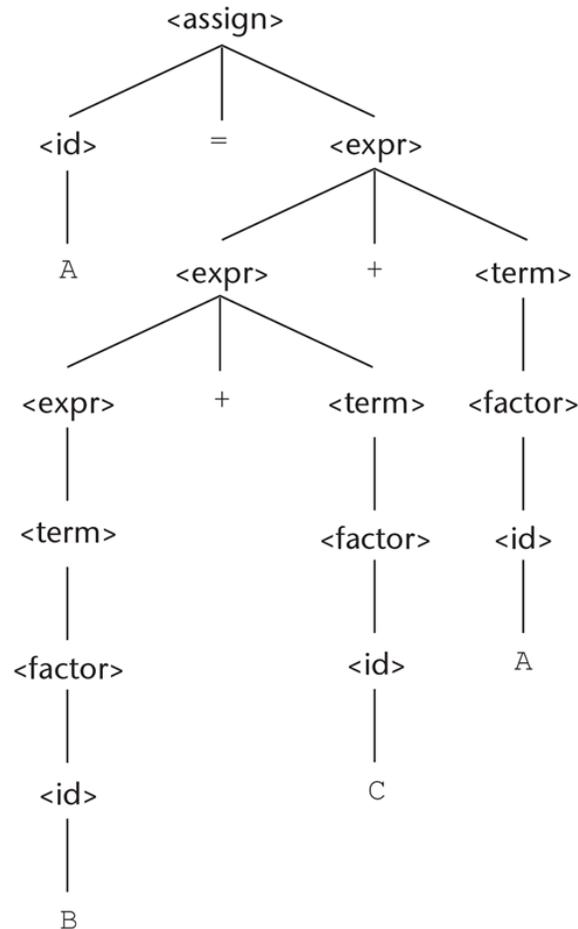


Figure 3.4 A parse tree for $A = B + C + A$ illustrating the associativity of addition

- Figure above shows the left + operator lower than the right + operator. This is the correct order if + operator meant to be **left associative**, which is typical.
- When a grammar rule has LHS also appearing at beginning of its RHS, the rule is said to be left recursive. The left recursion specifies left associativity.
- In most languages that provide it, the **exponentiation operator** is **right associative**. To indicate right associativity, right recursion can be used. A grammar rule is **right recursive** if the LHS appears at the right **end** of the RHS. Rules such as

```

<factor>  → <exp> ** <factor>
          | <exp>
<exp>    → (<exp>)
          | id
  
```

3.3.2 Extended BNF

- Because of minor inconveniences in BNF, it has been extended in several ways. EBNF extensions do not enhance the descriptive powers of BNF; they **only** increase its readability and writability.
- Three extension are commonly included in various versions of EBNF
 - **Optional** parts are placed in brackets ([])

```
<if_stmt > → if (expression) <statement> [else <statement>]
```

- Without the use the brackets, the syntactic description of this statement would require the following two rules:

```
<if_stmt > → if (expression) <statement>
           | if (expression) <statement> else <statement>
```

- Put **repetitions** (0 or more) in braces ({ })

```
<ident_list> → <identifier> {, <identifier>}
```

- Put **multiple-choice** options of RHSs in parentheses and separate them with vertical bars (|, OR operator)

```
<term> → <term> (* | / | %) <factor>
```

- In BNF, a description of this <term> would require the following three rules:

```
<term> → <term> * <factor>
       | <term> / <factor>
       | <term> % <factor>
```

- 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
```

3.4 Attribute Grammars 128

- 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.

3.4.1 Static Semantics

- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages.
- In Java, for example, a floating-point value **cannot** be assigned to an integer type variable, although the opposite is legal.
- The **static semantics** of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (syntax rather than semantics).
- Many static semantic rules of a language state its type constraints. Static semantics is so named because the analysis required to these specifications can be done at **compile time**.
- **Attribute grammars** was designed by Knuth (1968) to describe both the **syntax** and the **static semantics** of programs.

3.4.2 Basic Concepts

- Attribute grammars have additions to are context-free grammars to carry some **semantic** information on parse tree nodes.
- Attribute grammars are context-free grammars to which have been added attributes, attribute computation functions, and predicate function.

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 set $S(X)$ and $I(X)$, call synthesized and inherited attributes.
 - Synthesized attributes are used to pass semantic information **up** a parse tree, while inherited attributes pass semantic information **down** and across tree.
- Let $X_0 \rightarrow X_1 \dots X_n$ be a rule
 - Functions of the form $S(X_0) = f(A(X_1), \dots, A(X_n))$ define synthesized attributes
 - Functions of the form $I(X_j) = f(A(X_0), \dots, A(X_n))$, for $i \leq j \leq n$, define inherited attributes
 - Initially, there are **intrinsic attributes** on the leaves

3.4.4 Intrinsic Attributes

- 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 and their types.

3.4.5 Examples Attribute Grammars

- Example 3.6 An Attribute Grammar for Simple Assignment Statements

1. Syntax rule: $\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$
Semantic rule: $\langle \text{expr} \rangle.\text{expected_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$
2. Syntax rule: $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle[2] + \langle \text{var} \rangle[3]$
Semantic rule: $\langle \text{expr} \rangle.\text{actual_type} \leftarrow$
if $(\langle \text{var} \rangle[2].\text{actual_type} = \text{int} \text{ and } (\langle \text{var} \rangle[3].\text{actual_type} = \text{int}))$
then int
else real
end if
Predicate: $\langle \text{expr} \rangle.\text{actual_type} == \langle \text{expr} \rangle.\text{expected_type}$
3. Syntax rule: $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle$
Semantic rule: $\langle \text{expr} \rangle.\text{actual_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$
Predicate: $\langle \text{expr} \rangle.\text{actual_type} == \langle \text{expr} \rangle.\text{expected_type}$
4. Syntax rule: $\langle \text{var} \rangle \rightarrow A \mid B \mid C$
Semantic rule: $\langle \text{var} \rangle.\text{actual_type} \leftarrow \text{look-up}(\langle \text{var} \rangle.\text{string})$

The look-up function looks up a given variable name in the symbol table and returns the variable's type

- The syntax portion of our example attribute grammar is

```
 $\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$   
 $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle[2] + \langle \text{var} \rangle[3]$   
                  |  $\langle \text{var} \rangle$   
 $\langle \text{var} \rangle \rightarrow A \mid B \mid C$ 
```

- actual_type - A **synthesized** attribute associated with nonterminal $\langle \text{var} \rangle$ and $\langle \text{expr} \rangle$.
 - It is used to store the actual type, int or real, or a variable or expression.
 - In the case of a variable, the actual type is intrinsic.
 - In the of an expression, it is determined from the actual types of child node or children nodes of the $\langle \text{expr} \rangle$ nonterminal.
- expected_type - A **inherited** attribute associated with nonterminal $\langle \text{expr} \rangle$.
 - 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.

3.4.6 Computing Attribute Values

- Now, consider the process of computing the attribute values of a parse tree, which is sometimes called decorating the parse tree.
- The tree in Figure 3.7 show the flow of attribute values in the example of Figure 3.6.

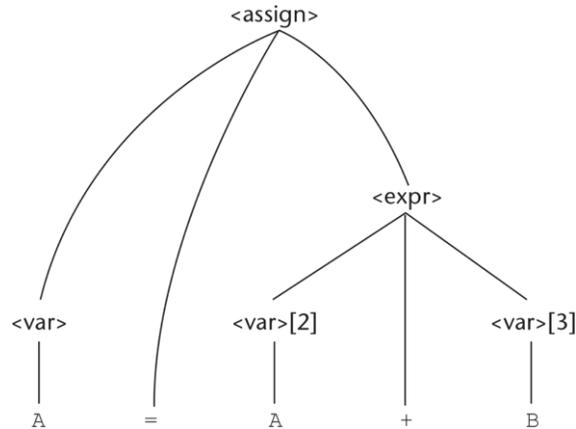


Figure 3.6 A parse tree for $A = A + B$

- The following is an evaluation of the attributes, in an order in which it is possible to computer them:

1. $\langle \text{var} \rangle . \text{actual_type}$ ← look-up (A) (Rule 4)
2. $\langle \text{expr} \rangle . \text{expected_type}$ ← $\langle \text{var} \rangle . \text{actual_type}$ (Rule 1)
3. $\langle \text{var} \rangle [2] . \text{actual_type}$ ← look-up (A) (Rule 4)
4. $\langle \text{var} \rangle [3] . \text{actual_type}$ ← look-up (B) (Rule 4)
5. $\langle \text{expr} \rangle . \text{actual_type}$ ← either int or real (Rule 2)
5. $\langle \text{expr} \rangle . \text{expected_type}$ == $\langle \text{expr} \rangle . \text{actual_type}$ is either TRUE or FALSE (Rule 2)

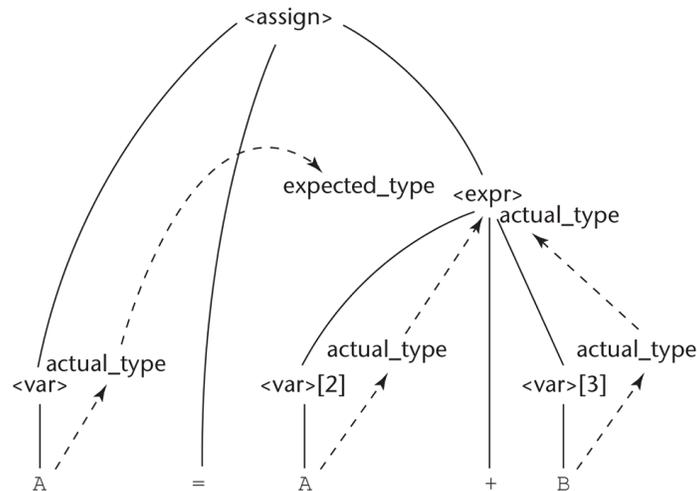
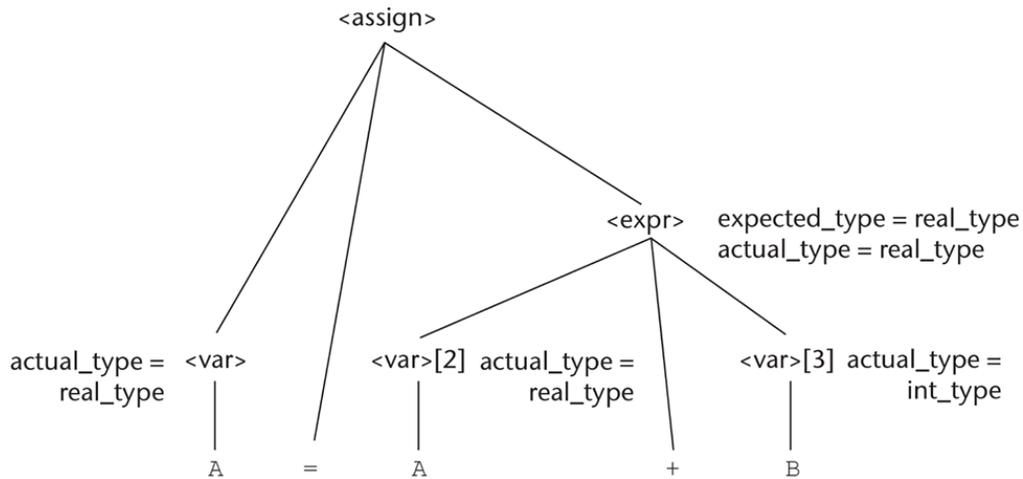


Figure 3.7 The flow of attributes in the tree

- 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**.



3.4.7 Evaluation

- Checking the static semantic rules of a language is an essential part of all **compiler**.
 - One of the main difficulties in using an attribute grammar to describe all of the syntax and static semantics of a real contemporary programming language is the **size and complexity** of the attribute grammar.
 - Furthermore, the attribute values on a large parse tree are **costly** to evaluate.

3.5 Describing the Meanings of Programs: Dynamic Semantics 134

- Three methods of semantic description:
 - **Operational semantics:** It is a method of **describing** the meaning of language constructs in terms of their effects on an ideal machine.
 - **Denotation semantics:** **Mathematical** objects are used to represent the meanings of language constructs. Language entities are converted to these mathematical objects with recursive functions.
 - **Axiomatic semantics:** It is based on formal **logic** and devised as a tool for proving the correctness of programs.

3.5.1 Operational Semantics

- 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 states**, where the machine's state is the collection of the collection of the values in its storage.
- Most programmers have written a small **test** program to determine the meaning of some programming language construct.
- The basic process of operational semantics is not unusual. In fact, the concept is frequently used in programming textbooks and programming in reference manuals.
- For example, the semantics of the C **for** construct can be described in terms of simpler statements, as in

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

- Operational semantics **depends** on programming languages of lower level, **not** mathematics and logic.

3.5.2 Denotational Semantics

- **Denotational semantics** is the most rigorous and most widely known formal method for describing the meaning of programs.
- It is solidly based on **recursive** function theory.

Two Simple Examples

- Example 1:
 - 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 show in Figure 3.9.

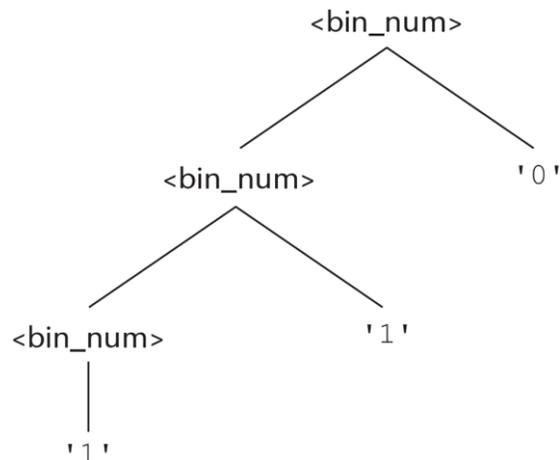


Figure 3.9 A parse tree of the binary number 110

- The semantic function, named 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. The function M_{bin} is defined as follows:

$$\begin{aligned}
 M_{bin} ('0') &= 0 \\
 M_{bin} ('1') &= 1 \\
 M_{bin} (<bin_num> '0') &= 2 * M_{bin} (<bin_num>) \\
 M_{bin} (<bin_num> '1') &= 2 * M_{bin} (<bin_num>) + 1
 \end{aligned}$$

- The meanings, or **denoted** objects (which in this case are decimal numbers), can be attached to the nodes of the parse tree, yielding the tree in Figure 3.10)

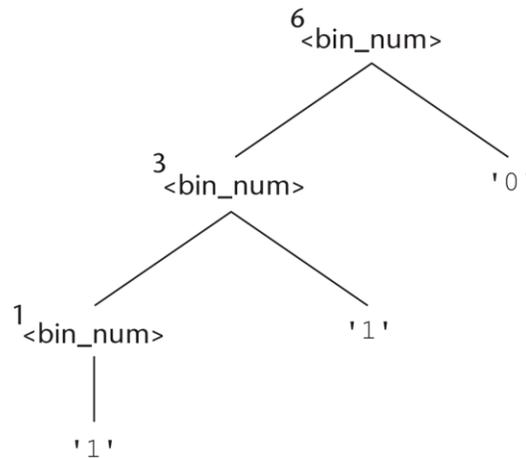


Figure 3.10 A parse tree with denoted objects for 110

- Example 2:
 - The syntactic domain is the set of character string representations of **decimal numbers**. The semantic domain is once again the set N .

$$\begin{aligned}
 <dec_num> &\rightarrow '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' \\
 &| <dec_num> ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9')
 \end{aligned}$$

- The denotational mapping for these syntax rules are

$$\begin{aligned}
 M_{dec} ('0') &= 0, M_{dec} ('1') = 1, M_{dec} ('2') = 2, \dots, M_{dec} ('9') = 9 \\
 M_{dec} (<dec_num> '0') &= 10 * M_{dec} (<dec_num>) \\
 M_{dec} (<dec_num> '1') &= 10 * M_{dec} (<dec_num>) + 1 \\
 \dots & \\
 M_{dec} (<dec_num> '9') &= 10 * M_{dec} (<dec_num>) + 9
 \end{aligned}$$

3.5.3 Axiomatic Semantics

- Axiomatic Semantics is based on **mathematical logic**.
- It is defined in conjunction with the development of a method to **prove** the correctness of programs.
 - Such correctness proofs, when they can be constructed, show that a program performs the computation described by its specification.
 - In a proof, each statement of a program is both preceded and followed by a logical expression that specifies constraints on program variables.
- Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions.)
 - The expressions are called **assertions**.

3.5.3.1 Assertions

- The logical expressions are called predicates, or **assertions**.
- An assertion before a statement (a **precondition**) states the relationships and constraints among variables that are true at that point in execution.
- An assertion following a statement is a **postcondition**.

3.5.3.2 Weakest Preconditions

- A **weakest precondition** is the least restrictive precondition that will guarantee the validity of the associated postcondition.
- The usual notation for specifying the axiomatic semantics of a given statement form is

$\{P\}$ statement $\{Q\}$

where P is the precondition, Q is the postcondition, and S is the statement form

- An example: $a = b + 1$ $\{a > 1\}$

One possible precondition: $\{b > 10\}$

Weakest precondition: **$\{b > 0\}$**

- If the weakest precondition can be computed from the given postcondition for each statement of a language, then correctness proofs can be constructed from programs in that language.
- **Program proof process:** The postcondition for the whole program is the desired result. Work back through the program to the first statement. If the precondition on the first statement is the same as the program spec, the program is correct.
- An **Axiom** is a logical statement that is assumed to be true.

- An **Inference Rule** is a method of inferring the truth of one assertion on the basis of the values of other assertions.

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

- The rule states that if S1, S2, ..., and Sn are true, then the truth of S can be inferred. The top part of an inference rule is call its antecedent; the bottom part is called it consequent.

3.5.3.3 Assignment Statements

- Ex:

$$a = b / 2 - 1 \{a < 10\}$$

The weakest precondition is computed by substituting $b / 2 - 1$ in the assertion $\{a < 10\}$ as follows:

$$\begin{array}{l} b / 2 - 1 < 10 \\ b / 2 < 11 \\ b < 22 \end{array}$$

∴ the weakest precondition for the given assignment and the postcondition is **$\{b < 22\}$**

- An assignment statement has a side effect if it changes some variable other than its left side.

- Ex:

$$x = 2 * y - 3 \{x > 25\}$$

The weakest precondition is computed as follows:

$$\begin{array}{l} 2 * y - 3 > 25 \\ 2 * y > 28 \\ y > 14 \end{array}$$

∴ the weakest precondition for the given assignment and the postcondition is **$\{y > 14\}$**

- Ex:

$$x = x + y - 3 \{x > 10\}$$

The weakest precondition is computed as follows:

$$\begin{array}{l} x + y - 3 > 10 \\ y > 13 - x \end{array}$$

∴ the weakest precondition for the given assignment and the postcondition is **$\{y > 13 - x\}$**

3.5.3.4 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 preconditions and postconditions.

$$\begin{array}{l} \{P1\} S1 \{P2\} \\ \{P2\} S2 \{P3\} \end{array}$$

The inference rule for such a two-statement sequence is

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

- Ex:

$$\begin{array}{l} y = 3 * x + 1; \\ x = y + 3; \quad \{x < 10\} \end{array}$$

The weakest precondition is computed as follows:

$$\begin{array}{l} y + 3 < 10 \\ y < 7 \\ \\ 3 * x + 1 < 7 \\ 3 * x < 6 \\ x < 2 \end{array}$$

∴ the weakest precondition for the first assignment statement is $\{x < 2\}$

3.5.3.5 Selection

- We next 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 clause. The inference rule is

$$\frac{\{B \text{ and } P\} S1 \{Q\}, \{\text{not } B \text{ and } P\} S2 \{Q\}}{\{P\} \text{ if } B \text{ then } S1 \text{ else } S2 \{Q\}}$$

- Example of selection statement is

```
If (x > 0) then
    y = y - 1;
else
    y = y + 1;
{y > 0}
```

We can use the axiom for assignment on the then clause

$y = y - 1 \{y > 0\}$

This produce precondition $\{y - 1 > 0\}$ or $\{y > 1\}$

Now we apply the same axiom to the else clause

$y = y + 1 \{y > 0\}$

This produce precondition $\{y + 1 > 0\}$ or $\{y > -1\}$

$y > 1 \text{ AND } y > -1$

$\{y > 1\}$

Because $\{y > 1\} \Rightarrow \{y > -1\}$, the rule of consequence allows us to use $\{y > 1\}$ for the precondition of selection statement.

3.5.3.6 Logical Pretest Loops

- Computing the weakest precondition (wp) for a while loop is inherently more difficult than for a sequence b/c the number of iterations cannot always be predetermined.
- 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 as follows:

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

In this rule, I is the loop invariant

- The axiomatic description of a while loop written as

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

- It is helpful to treat the process of producing the wp as a function, **wp**.

`wp(statement, postcondition) = precondition`

- To find **I**, we use the loop postcondition to compute preconditions for several different numbers of iterations of the loop body, starting with none. If the loop body contains a single assignment statement, the axiom for assignment statements can be used to compute these cases.
- Characteristics of the loop invariant: **I** must meet the following conditions:
 - `P => I` -- the loop invariant must be true initially
 - `{I} B {I}` -- evaluation of the Boolean must not change the validity of I
 - `{I and B} S {I}` -- I is not changed by executing the body of the loop
 - `(I and (not B)) => Q` -- if I is true and B is false, Q is implied
 - The loop terminates -- can be difficult to prove

- Ex:

```
while y <> x do y = y + 1 end {y = x}
```

For 0 iterations, the wp is \rightarrow {y = x}

For 1 iteration,

wp(y = y + 1, {y = x}) = {y + 1 = x}, or {y = x - 1}

For 2 iterations,

wp(y = y + 1, {y = x - 1}) = {y + 1 = x - 1}, or {y = x - 2}

For 3 iterations,

wp(y = y + 1, {y = x - 2}) = {y + 1 = x - 2}, or {y = x - 3}

- It is now obvious that {y < x} will suffice for cases of one or more iterations. Combining this with {y = x} for the 0 iterations case, we get **{y <= x}** which can be used for the loop invariant.

- Ex:

```
while s > 1 do s = s / 2 end {s = 1}
```

For 0 iterations, the wp is \rightarrow {s = 1}

For 1 iteration,

wp(s > 1, {s = s / 2}) = {s / 2 = 1}, or {s = 2}

For 2 iterations,

wp(s > 1, {s = s / 2}) = {s / 2 = 2}, or {s = 4}

For 3 iterations,

wp(s > 1, {s = s / 2}) = {s / 2 = 4}, or {s = 8}

- Loop Invariant **I** is {s is a **nonnegative power of 2**}
- The loop invariant **I** is a weakened version of the loop postcondition, and it is also a precondition.
- **I** must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition.

Summary

- **BNF** and context-free grammars are equivalent meta-languages
 - Well-suited for describing the **syntax** of programming languages
- An **attribute grammar** is a descriptive formalism that can describe both the syntax **and** the semantics of a language
- Three primary methods of semantics description
 - **Operation, denotational, axiomatic**