Chapter 3

Describing Syntax and Semantics
Chapter 3 Topics

• Introduction
• The General Problem of Describing Syntax
• Formal Methods of Describing Syntax
• Attribute Grammars
• Describing the Meanings of Programs: Dynamic Semantics
Introduction

• **Syntax:** the form or structure of the expressions, statements, and program units

• **Semantics:** the meaning of the expressions, statements, and program units

• Syntax and semantics provide a language’s definition
  
  – Users of a language definition
    • Other language designers
    • Implementers
    • Programmers (the users of the language)
The General Problem of Describing Syntax: Terminology

• **Sentence**
  – a string of characters over some alphabet

• **Language**
  – a set of sentences

• **Lexeme**
  – the lowest level syntactic unit of a language (e.g., *, sum, begin)

• **Token**
  – category of lexemes (e.g., identifier)

\[
\text{Index} = 2 \times \text{count} + 17;
\]

<table>
<thead>
<tr>
<th><strong>Lexemes</strong></th>
<th><strong>Tokens</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>identifier</td>
</tr>
<tr>
<td>=</td>
<td>equal_sign</td>
</tr>
<tr>
<td>2</td>
<td>int_literal</td>
</tr>
<tr>
<td>*</td>
<td>mult_op</td>
</tr>
<tr>
<td>count</td>
<td>identifier</td>
</tr>
<tr>
<td>+</td>
<td>plus_op</td>
</tr>
<tr>
<td>17</td>
<td>int_literal</td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
</tr>
</tbody>
</table>
Formal Definition of Languages

- Languages can be formally defined in two ways: by recognition and by generation

- **Recognizers**
  - A recognition device reads input strings of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler
  - Detailed discussion in Chapter 4

- **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
Formal Methods of Describing Syntax

• Backus-Naur Form and Context-Free Grammars
  – Most widely known method for describing programming language syntax

• Extended BNF
  – Improves readability and writability of BNF

• Grammars and Recognizers
BNF and Context-Free Grammars

• Context-Free Grammars
  – Developed by Noam Chomsky in the mid-1950s
  – He described four classes of language generators, (or grammars) that define four classes of natural languages
  – Two of them, named context-free and regular, turned out to be useful for describing the syntax of programming languages
    • Regular grammar:
      – describe the tokens of programming languages
    • Context-free grammar:
      – describe the whole programming languages
Backus-Naur Form (BNF)

• Backus-Naur Form (1959)
  – Invented by John Backus to describe Algol 58
  – Peter Naur modified it to describe Algol60, and called Backus-Naur Form (BNF).
  – BNF is equivalent to context-free grammars
  – BNF is a *metalanguage* used to describe another language
  – In BNF, *abstractions* (also called *nonterminal symbols*) are used to represent classes of syntactic structures (act like variables)
    • E.g., a simple Java assignment statement might be represented by the abstraction `<assign>`
      with definition: `<assign> → <var> = <expression>`

Copyright © 2006 Addison-Wesley. All rights reserved.
BNF Fundamentals

- **Non-terminals**: BNF abstractions
- **Terminals**: lexemes and tokens
- **Grammar**: a collection of rules
- **Rule**: it has a left-hand side (LHS) and a right-hand side (RHS), and consists of *terminals* and *nonterminals*

  Ex. `<if_stmt> → if <logic_expr> then <stmt>`
  - LHS is the abstraction being defined
  - RHS consisting of token, lexeme or abstraction is the definition of LHS

- An abstraction can have more than one definitions (RHS)

  `<if_stmt> → if <logic_expr> then <stmt> else <stmt>`
  
  or
  `<if_stmt> → if <logic_expr> then <stmt>`
  
  or
  `<if_stmt> → if <logic_expr> then <stmt> else <stmt>`
Describing Lists

• Variable-length lists in mathematics are described using an ellipsis (...) which is NOT used in BNF.
• Syntactic lists are described using recursion (its LHS shows up in its RHS).

\[ <\text{ident\_list}> \rightarrow \text{ident} \]

\[ \text{\hspace{1cm} | \hspace{1cm}} \text{ident, <ident\_list>} \]

• A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminals)
• In a grammar for a programming language, the start symbol denotes a complete program and is usually named <program> .
An Example: Grammar & Derivation

**Grammar Example**

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

<expr> → <term> + <term> | <term> − <term>

<term> → <var> | const

**Derivation Example**

<program> => <stmts>

=> <stmt>

=> <var> = <expr>

=> a = <expr>

=> a = <term> + <term>

=> a = <var> + <term>

=> a = b + <term>

=> a = b + const
Derivation

• Every string of symbols in the derivation 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

• A derivation may be neither leftmost nor rightmost
Derivation

Leftmost Derivation

\[ \text{<program>} \Rightarrow \text{<stmts>} \]
\[ \Rightarrow \text{<stmt>} \]
\[ \Rightarrow \text{<var> = <expr>} \]
\[ \Rightarrow a = \text{<expr>} \]
\[ \Rightarrow a = \text{<term> + <term>} \]
\[ \Rightarrow a = \text{<var> + <term>} \]
\[ \Rightarrow a = b + \text{<term>} \]
\[ \Rightarrow a = b + \text{const} \]

Rightmost Derivation

\[ \text{<program>} \Rightarrow \text{<stmts>} \]
\[ \Rightarrow \text{<stmt>} \]
\[ \Rightarrow \text{<var> = <expr>} \]
\[ \Rightarrow \text{<var> = <term> + <term>} \]
\[ \Rightarrow \text{<var> = <term> + const} \]
\[ \Rightarrow \text{<var> = <var> + const} \]
\[ \Rightarrow \text{<var> = b + const} \]
\[ \Rightarrow a = b + \text{const} \]
Parse Tree

- A hierarchical representation of a derivation
  - Internal nodes: nonterminals
  - Leave nodes: terminals

```plaintext
<program>
  <stmts>
    <stmt>
      <var> = <expr>
        <var> <term> + <term>
          a <term> + <term>
            const
          b
    </stmt>
  </stmts>
</program>
```
A grammar is ambiguous if and only if it generates a sentential form that has two or more distinct parse trees:
- The grammar generates a sentence with more than one leftmost derivation.
- The grammar generates a sentence with more than one rightmost derivation.

If a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely.
An Ambiguous Grammar

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

- **Two left-most derivations for** A = B + C * A

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

<assign> → <id> = <expr>
→ A = <expr>
→ A = <expr> + <expr>
→ A = <id> + <expr>
→ A = B + <expr>
→ A = B + <expr> * <expr>
→ A = B + <id> * <expr>
→ A = B + C * <expr>
→ A = B + C * <id>
→ A = B + C * A
An Ambiguous Grammar

\[ \text{<assign>} \rightarrow \text{<id>} = \text{<expr>} \]

\[ \text{<id>} \rightarrow A \mid B \mid C \]

\[ \text{<expr>} \rightarrow \text{<expr>} + \text{<expr>} \mid \text{<expr>} * \text{<expr>} \mid (\text{<expr>}) \mid \text{id} \]

\[ A = B + C * A \]

has two distinct parse trees
Operator Precedence

• If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity
  – **Operator Precedence**: The evaluation order of the operators in the same expression.
  – Operators with higher precedence levels will be evaluated first, regardless of the order of the operators in the expression.
    • \( x + y \times z, \ x \times y + z \) => \( y \times z \) is always done first
  – The operator generated *lower* in the parse tree has *higher* precedence level and vice versa.
An Unambiguous Grammar

\[
\text{<assign> } \rightarrow \text{ <id> } = \text{ <expr> }
\]

\[
\text{<id> } \rightarrow \text{ A | B | C }
\]

\[
\text{<expr> } \rightarrow \text{ <expr> + <term> | <term> }
\]

\[
\text{<term> } \rightarrow \text{ <term> * <factor> | <factor> }
\]

\[
\text{<factor> } \rightarrow \text{ (<expr>) | <id> }
\]

The * operator has highest precedence level and is lowest in the tree.

\[A = B + C * A\] has an unique parse trees.
Quiz

• Which one is an ambiguous grammar? Prove your answer using two distinct parse trees.

Grammar 1

\[
\begin{align*}
<\text{expr}> & \rightarrow <\text{expr}> <\text{op}> <\text{expr}> \mid \text{const} \\
<\text{op}> & \rightarrow / \mid -
\end{align*}
\]

Grammar 2

\[
\begin{align*}
<\text{expr}> & \rightarrow <\text{expr}> - <\text{term}> \mid <\text{term}> \\
<\text{term}> & \rightarrow <\text{term}> / \text{const} \mid \text{const}
\end{align*}
\]
Associativity of Operators

• When an expression includes two operators that have the same precedence, the rule to specify which is done first is called **associativity**
  – Addition and multiplication are associative
    • \((A+B) + C = A + (B+C)\)
  – Subtraction and division are not associative
    • \((A-B) - C = A - (B-C)\)
  – *Left recursive* specifies left associativity
    • Left recursive: LHS appears at the beginning of its RHS
  – *Right recursive* specifies right associativity
    • Right recursive: LHS appears at the right end of RHS
Associativity of Operators

- Operator associativity can be indicated by a grammar
  - Left recursive specifies left associativity
  - Right recursive specifies right associativity

\[
\begin{align*}
<expr> & \rightarrow <expr> + <expr> \mid \text{const (ambiguous)} \\
<expr> & \rightarrow <expr> + \text{const} \mid \text{const (unambiguous)} \\
<factor> & \rightarrow \text{const} ** <factor> \mid \text{const (unambiguous)}
\end{align*}
\]
Extended BNF

- **Optional** parts are placed in brackets \([ \) \]
  \(<\text{proc\_call}> \rightarrow \text{ident} \ [(\langle\text{expr\_list}\rangle)]\)

- **Alternative** parts of RHSs are placed inside parentheses and separated via vertical bars
  \(<\text{term}> \rightarrow \langle\text{term}> (+|−) \text{const}\)

- **Repetitions** (0 or more) are placed inside braces \(\{ \) \}
  \(<\text{ident}> \rightarrow \text{letter} \ \{\text{letter}\mid\text{digit}\}\)
BNF and EBNF

**BNF**

<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>

**EBNF**

<expr> → <term> { + <term>}  
<term> → <factor> { * <factor>}  

**BNF**

<expr> → <expr> + <term>  
| <expr> - <term>  
| <term>

**EBNF ?**
Attribute Grammars

- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Additions to CFGs to carry some semantic info along parse trees
- Primary value of attribute grammars (AGs)
  - Static semantics specification
    - semantics checking can be done at compile time
    - E.g., a floating-point value cannot be assigned to an integer type variable. (hard to specify in BNF)
    - E.g., all variables must be declared before they are referenced. (cannot be specified in BNF)
Attribute Grammars: Definition

- An attribute grammar is a context-free grammar $G = (S, N, T, P)$ with the following additions:
  - For each grammar symbol $x$ there is a set $A(x)$ of attribute values
  - Each rule has a set of functions that define certain attributes of the nonterminals in the rule
  - Each rule has a (possibly empty) set of predicates to check for attribute consistency
Attribute Grammars: A simple example

• Ex. The name on the **end** of an procedure must match the procedure’s name.

```plaintext
Procedure myfunc
  a = b + c;
end myfunc;
```

Syntax rule: `<proc_def> → Procedure <proc_name> [1]
<proc_body> end <proc_name>[2];`

Predicate: `<proc_name>[1].string == <proc_name>[2].string`

– When there is more than one occurrence of a non-terminal in a syntax rule, the non-terminals are subscripted with brackets to distinguish them
Attribute Grammars: An Example

• A simple assignment statement
  – The right side of the assignments can be: a variable or an expression
  – The expression is in the form of a variable added to another variable
  – The variable names are A, B, and C.

  – The variable type can be: int or real
  – The type of expression
    • Real: if the operand types are not the same.
    • Type of the operands: otherwise.
  – The LHS and RHS of the assignment have the same type.

  e.g., A = B + C

  e.g.,
  (o) int = int + int    (o) real = real + int
  (o) real = real + real  (x) int = real + int
Attribute Grammars: Definition

• Let $X_0 \to X_1 \ldots X_n$ be a rule

• **Synthesized attributes:**
  – used to pass semantic information up a parse tree.
  – The values depends on that node’s children nodes.
  – Functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$

• **inherited attributes**
  – Used to pass semantic information down a parse tree.
  – The values depends on its parent and its sibling nodes.
  – Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, for $i \leq j \leq n$.

• Initially, there are **intrinsic attributes** on the leaves.
  – Intrinsic attribute values are decided outside a parse tree.
Attribute Grammars: An Example

• Syntax

\[
\begin{align*}
<\text{assign}> & \rightarrow <\text{var}> = <\text{expr}> \\
<\text{expr}> & \rightarrow <\text{var}> + <\text{var}> \mid <\text{var}> \\
<\text{var}> & \rightarrow A \mid B \mid C
\end{align*}
\]

• Attributes

- \texttt{actual\_type}: synthesized attributes for \texttt{<var>} and \texttt{<expr>}
- \texttt{expected\_type}: inherited attributes for \texttt{<expr>}

Copyright © 2006 Addison-Wesley. All rights reserved.
1. Syntax rule: \(<assign> \rightarrow \var = \expr\>
   Semantic rule: \(<\expr>.expected\_type \leftarrow \var.actual\_type\>

2. Syntax rule: \(<\expr> \rightarrow \var[2] + \var[3]\>
   Semantic rules: \(<\expr>.actual\_type \leftarrow \)
   \[
   \text{if} (\var[2].actual\_type = \text{int}) \text{ and } (
   \var[3].actual\_type = \text{int}) \text{ then int}
   \text{else real}
   \]
   Predicate: \(<\expr>.actual\_type == \expr.expected\_type\>

3. Syntax rule: \(<\expr> \rightarrow \var\>
   Semantic rule: \(<\expr>.actual\_type \leftarrow \var.actual\_type\>
   Predicate: \(<\expr>.actual\_type == \expr.expected\_type\>

4. Syntax rule: \(<\var> \rightarrow A|B|C\>
   Semantic rule: \(<\var>.actual\_type \leftarrow \text{lookup} (\var.string)\>
Attribute Grammars (continued)

• How are attribute values computed?
  – If all attributes were inherited, the tree could be decorated in top-down order.
  – If all attributes were synthesized, the tree could be decorated in bottom-up order.
  – In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.
Attribute Grammars (continued)

Figure 3.6
A parse tree for
\[ A = A + B \]
Attribute Grammars (continued)

- Evaluate the attributes of \( A = A + B \)
  1. `<var>.actual_type ← lookup(A)` (Rule 4)
  2. `<expr>.expected_type ← <var>.actual_type` (R1)
  3. `<var>[2].actual_type ← lookup(A)` (R4)
     `<var>[3].actual_type ← Lookup(B)` (R4)
  4. `<expr>.actual_type ← either int or real` (R2)
  5. `<expr>.expected_type == <expr>.actual_type is either TRUE or FALSE` (R2)
Attribute Grammars (continued)

Figure 3.7
The flow of attributes in the tree

Copyright © 2006 Addison-Wesley. All rights reserved.
Attribute Grammars (continued)

Figure 3.8
A fully attributed parse tree

```
<assign>
    <expr>  expected_type = real_type
        <var>[2]  actual_type = real_type
            <var>  A = A
        +
            <var>[3]  actual_type = int_type
                B
```

Copyright © 2006 Addison-Wesley. All rights reserved.
Semantics

• There is no single widely acceptable notation or formalism for describing semantics

• Operational Semantics
  – Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
Operational Semantics

• The idea of operational semantics
  – To describe the meaning of a program by translating it into a more easily understood intermediate language.

C statement
for(expr1; expr2; expr3) {
  ...
}

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

• The basic process
  – Build a translator (translates source code to the machine code of an idealized computer)
  – Build a simulator for the idealized computer
Axiomatic Semantics

• Original purpose: formal program verification
  – To prove the correctness of programs
• In a proof, each statement of a program is both preceded and followed by a logical expression (called *predicates*, or *assertions*) that specifies constraints on program variables.
Axiomatic Semantics (continued)

• **Precondition**
  – An assertion before a statement
  – Describes the constraints on program variables at that point in the program.

• **Postcondition**
  – An assertion following a statement
  – Describes the new constraints on those variables after execution of the statement

• The postcondition for the entire program is the desired result
  – Work back through the program to first statement. If its precondition meets the program specification, the program is correct.
Axiomatic Semantics Form

• Pre-, post form: \( \{P\} \) statement \( \{Q\} \)

• An example
  – \( a = b + 1 \) \( \{a > 1\} \)
  – One possible precondition: \( \{b > 10\} \)
  – Weakest precondition: \( \{b > 0\} \)

• A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition
Axiomatic Semantics: Axioms

• **Axiom**
  – a logical statement that is assumed to be true.
  – An axiom for assignment statements
    \[ \{P\} x = E \{Q\}, \quad P \text{ is defined by } P = Q_{x \rightarrow E} \]
    • Means \( P \) is computed as \( Q \) with all instances of \( x \) replaced by \( E \).

• **Inference rule**
  – a method of inferring the truth of one assertion on the basis of the values of other assertions.
  – General form: \[ S_1, S_2, \ldots, S_n \]
  
  
  \[ S \]
  
  
  If \( S_1, S_2, \ldots, S_n \) are true, the truth of \( S \) can be inferred
  – The Rule of Consequence: \[ \{P\} S \{Q\}, P' \Rightarrow P, Q \Rightarrow Q' \]
  
  
  \[ \{P'\} S \{Q'\} \]
Axiomatic Semantics: Axioms

• Examples - assignment
  – Compute the weakest pre-condition for each assignment.
    
    1. \( a = b/2 \) -1 \{ a<10 \} \quad b < 22
    2. \( x = 2*y - 3 \) \{ x > 25 \} \quad y > 14
    3. \( x = x + y - 3 \) \{ x > 10\} \quad y > 13 - x

  – Prove each of the assignment below
    
    1. \{x>3\} \( x = x - 3 \) \{ x>0 \}
    2. \{x>5\} \( x = x - 3 \) \{ x>0 \}
Axiomatic Semantics: Axioms

• An inference rule for sequences

\[
\begin{align*}
\{P1\} S1 \{P2\} \\
\{P2\} S2 \{P3\} \\
\{P1\} S1; S2 \{P3\}
\end{align*}
\]

• Example - sequence

• Find the weakest pre-condition for the sequence.

- \( y = 3*x + 1; \)
- \( x = y + 3; \)
- \( \{ x < 10 \} \)

\[
\begin{align*}
3 \cdot x + 1 &< 7 \quad \Rightarrow \quad x < 2 \\
Y + 3 &< 10 \quad \Rightarrow \quad y < 7
\end{align*}
\]
Axiomatic Semantics: Axioms

• An inference rule for logical pretest loop
  – Computing the weakest precondition for a pretest loop is difficult than for a sequence, because the number of iterations cannot always be predetermined.

\[
\{P\} \text{ while } B \text{ do } S \text{ end } \{Q\} \\
(I \text{ and } B) \text{ S } \{I\} \\
\{I\} \text{ while } B \text{ do } S \{I \text{ and } (\text{not } B)\} \\
\]

where I is the loop invariant (the inductive hypothesis)
Axiomatic Semantics: Axioms

• \{P\} while B do S end \{Q\}

• Characteristics of the loop invariant: I must meet the following conditions:
  
  – P \Rightarrow I \quad \text{-- the loop invariant must be true initially}
  
  – \{I\} B \{I\} \quad \text{-- evaluation of the Boolean must not change the validity of I}
  
  – \{I \text{ and } B\} S \{I\} \quad \text{-- I is not changed by executing the body of the loop}
  
  – (I \text{ and } \neg B) \Rightarrow Q \quad \text{-- if I is true and B is false, is implied}
  
  – The loop terminates
Axiomatic Semantics: Axioms

• **Example 1 – Find loop invariant**
  
  – while y <> x do y=y+1 end {y=x}
  
  – zero iterations: \{y=x\}
  – One iteration: \(wp(y=y+1, \{y=x\}) = \{y=x-1\}\)
  – Two iterations: \(wp(y=y+1, \{y=x-1\}) = \{y=x-2\}\)
  – Three iterations: \(wp(y=y+1, \{y=x-2\}) = \{y=x-3\}\)
  – ...
  – Loop invariant \{ y<= x \}, which can be used as P

• **Prove the loop invariant requirements**
Axiomatic Semantics: Axioms

• **Example 2** – Find loop invariant

  – **while** $s > 1$ **do** $s = s/2$ **end** \{ $s = 1$ \}

  – Zero iteration \( s = 1 \)
  – One iteration \( s/2 = 1 \Rightarrow s = 2 \)
  – Two iterations \( s/2 = 2 \Rightarrow s = 4 \)
  – Three iterations \( s/2 = 4 \Rightarrow s = 8 \)

  The invariant is \{ $s$ is a nonnegative **power of 2** \}
  The computed invariant I can serve as $P$. 
Loop Invariant

• 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.
Evaluation of Axiomatic Semantics

• Developing axioms or inference rules for all of the statements in a language is difficult

• It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers

• Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers
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, axiomatic