3. DESCRIBING SYNTAX AND SEMANTICS

Introduction

- The task of providing a concise yet understandable description of a programming language is difficult but essential to the language's success.

- A language description needs to be useful to both programmers and language implementors.

- A language description must cover two aspects of the language:

  Syntax: the form of its expressions, statements, and program units
  Semantics: the meaning of those expressions, statements, and program units

- Syntax of Java's **`while`** statement:

  `while` (*boolean_expr*) *statement*

  Semantics: If the Boolean expression is true, the embedded statement is executed. Control then returns to the expression to repeat the process. If the expression is false, control is transferred to the statement following the **`while`**.

- In a well-designed programming language, semantics should follow directly from syntax.

- Describing syntax is easier than describing semantics.

The General Problem of Describing Syntax

- A language, whether natural or artificial, 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.

- Formal descriptions of programming language syntax do not always include the lowest-level syntactic units (**lexemes**). Lexemes include identifiers, literals, operators, and special words, among others.

- A **token** of a language is a category of its lexemes.

- An example Java statement:

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

  Lexemes and tokens of this statement:

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

Language Recognizers and Generators

- In general, languages can be formally defined in two distinct ways: by **recognition** and by **generation.**

- A recognizer R for a language L would be given a string of characters. R would then indicate whether or not the string belonged to L. Such a recognizer would serve as a description of L.

- The syntax analysis part of a compiler is a recognizer for the language the compiler translates.

- A language generator is a device that can be used to generate the sentences of a language. Like a recognizer, a generator is also a description of a language.

- Generators are usually easier to understand than recognizers. It is often possible to determine whether a particular sentence is correct by comparing it with the structure of the generator.

- There is a close connection between formal generation and recognition devices for the same language.

Backus-Naur Form and Context-Free Grammars

- John Backus and Noam Chomsky separately invented a notation that has become the most widely used method for formally describing programming language syntax.

- In the mid-1950s, Chomsky, a noted linguist, described four classes of generative devices or **grammars** that define four classes of languages. Two of these classes are useful for describing the syntax of programming languages:

  Regular grammars—Can describe the appearance of tokens of programming languages.
  Context-free grammars—Can describe the syntax of whole programming languages, with minor exceptions.

- In 1959, John Backus published a similar notation for specifying programming language syntax.

- Backus's notation was later modified slightly by Peter Naur for the description of ALGOL 60. This revised notation became known as **Backus-Naur form,** or simply **BNF.**

- BNF is nearly identical to Chomsky's **context-free grammars,** so the terms are often used interchangeably.

Fundamentals of BNF

- A **metalanguage** is a language that is used to describe another language. BNF is a metalanguage for programming languages.

- BNF uses abstractions for syntactic structures. A simple Java assignment, for example, might be represented by the abstraction <assign>. The definition of <assign> is given by a **rule** or **production:**

  <assign> → <var> = <expression>

- Each rule has a **left-hand side** (LHS) and a **right-hand side** (RHS). The LHS is the abstraction being defined. The RHS consists of tokens, lexemes, and references to other abstractions.

- Abstractions are called **nonterminal symbols,** or simply **nonterminals.**

- Lexemes and tokens are called **terminal symbols,** or simply **terminals.**

- A BNF description, or **grammar,** is a collection of rules.

Fundamentals of BNF (Continued)

- Nonterminal symbols can have more than one definition. Multiple definitions can be written as a single rule, with the definitions separated by the symbol |.

- Java's **`if`** statement can be described with two rules:

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

  or with one rule:

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

- A rule is **recursive** if its LHS appears in its RHS. Recursion is often used when a nonterminal represents a variable-length list of items.

- The following rule describes a list of identifiers separated by commas:

  <ident_list> → identifier
  | identifier , <ident_list>

Grammars and Derivations

- A grammar is a generative device for defining languages.

- The sentences of a language are generated through a sequence of applications of the rules, beginning with a special nonterminal called the **start symbol.** In a programming language grammar, the start symbol is often named <program>.

- This sequence of rule applications is called a **derivation.**

- 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>

- A derivation of a program in this language:

<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**

- In a derivation, each successive string is derived from the previous string by replacing one of the nonterminals with one of that nonterminal's definitions.

Grammars and Derivations (Continued)

- Each string in a derivation, including the start symbol, is called a **sentential form.**

- A derivation continues until the sentential form contains no nonterminals.

- A **leftmost derivation** is one in which the replaced nonterminal is always the leftmost nonterminal. In addition to leftmost, a derivation may be rightmost or in an order that is neither leftmost nor rightmost.

- Derivation order has no effect on the language generated by a grammar.

- By choosing alternative rules with which to replace nonterminals in the derivation, different sentences in the language can be generated. By exhaustively choosing all combinations of choices, the entire language can be generated.

- A grammar for simple assignment statements:

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

- A leftmost derivation of the statement `A = B * ( A + C )`:

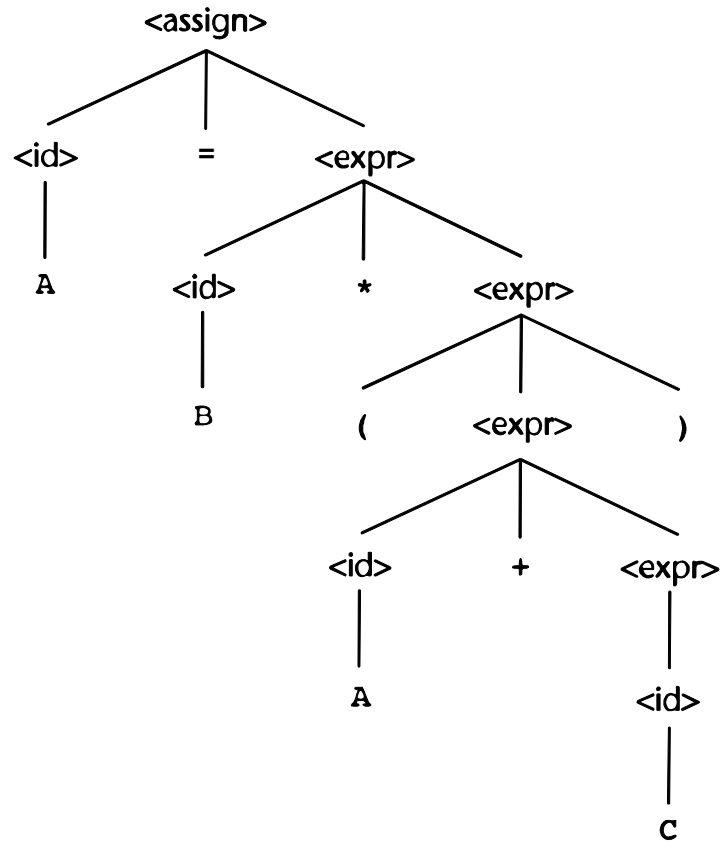
  <assign> => <id> = <expr>
  => A = <expr>
  => A = <id> * <expr>
  => A = B * <expr>
  => A = B * (<expr>)
  => A = B * (<id> + <expr>)
  => A = B * (A+ <expr>)
  => A = B * (A + <id>)
  => A = B * (A + C)

Parse Trees

- A derivation can be represented graphically in the form of a **parse tree.**

- The following parse tree shows the structure of the statement

```
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.

Ambiguity

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

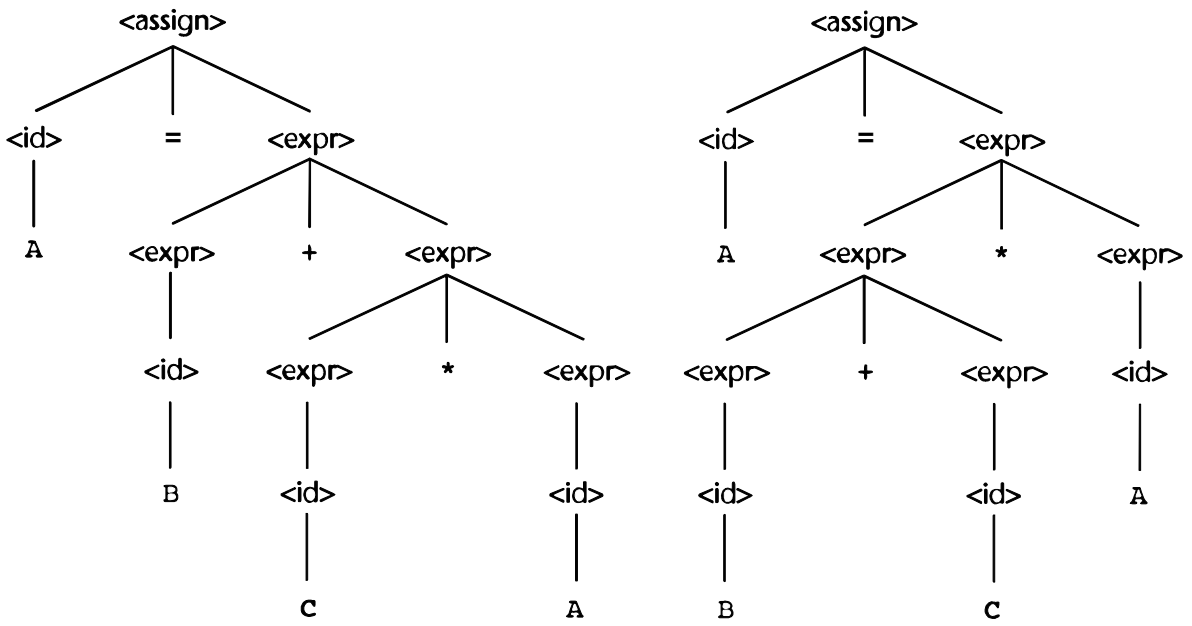
- An ambiguous grammar for simple assignment statements:

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

- This grammar is ambiguous because the sentence

  `A = B + C * A`

  has two distinct parse trees:



- Syntactic ambiguity is a problem because compilers often base the semantics of those structures on their syntactic form. If a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely.

Operator Precedence

- Ambiguity in an expression grammar can often be resolved by rewriting the grammar to reflect operator precedence. This process involves additional nonterminals and some new rules.

- An unambiguous grammar for simple assignment statements:

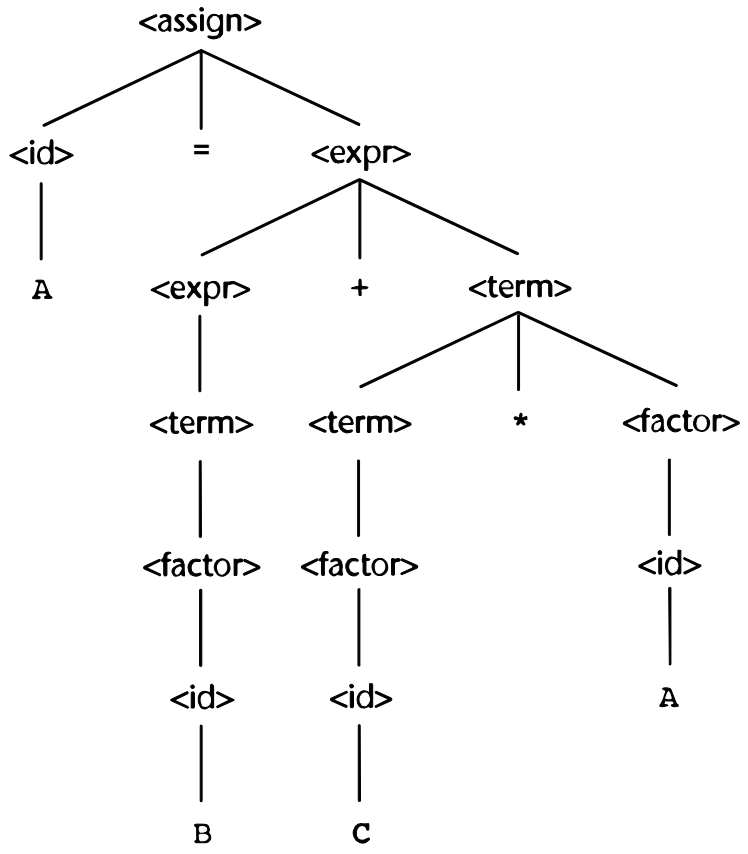
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| < term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

- The following derivation of the sentence A = B + C * A uses the unambiguous grammar:

<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

Operator Precedence (Continued)

- The sentence `A = B + C * A` now has a unique parse tree:



Operator Precedence (Continued)

- The connection between parse trees and derivations is very close; either can easily be constructed from the other.

- Every derivation with an unambiguous grammar has a unique parse tree, although that tree can be represented by different derivations.

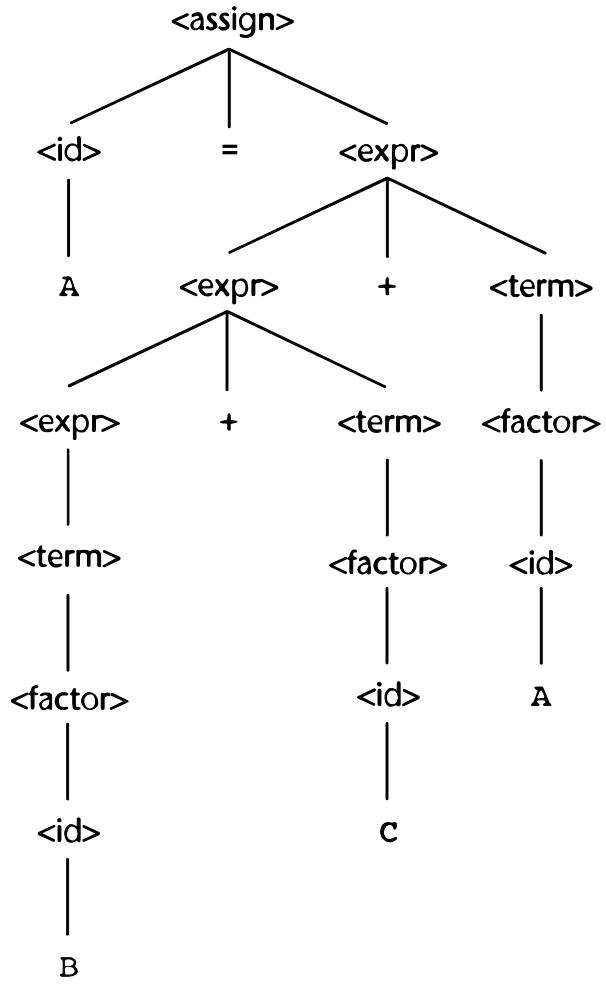
- For example, the following derivation of the sentence `A = B + C * A` is different from the derivation of the same sentence given previously.

  <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

  This is a rightmost derivation, whereas the previous one is leftmost. Both derivations, however, are represented by the same parse tree.

Associativity of Operators

- A grammar that describes expressions needs to handle associativity correctly.
- The parse tree for `A = B + C + A` illustrates this issue:



- The parse tree shows the left addition operator lower than the right addition operator. This is the correct order if addition is meant to be left associative.

Associativity of Operators (Continued)

- When a BNF rule has its LHS also appearing at the beginning of its RHS, the rule is said to be **left recursive.** Left recursion corresponds to left associativity.

- 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 to describe exponentiation as a right-associative operator:

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

An Unambiguous Grammar for `if-else`

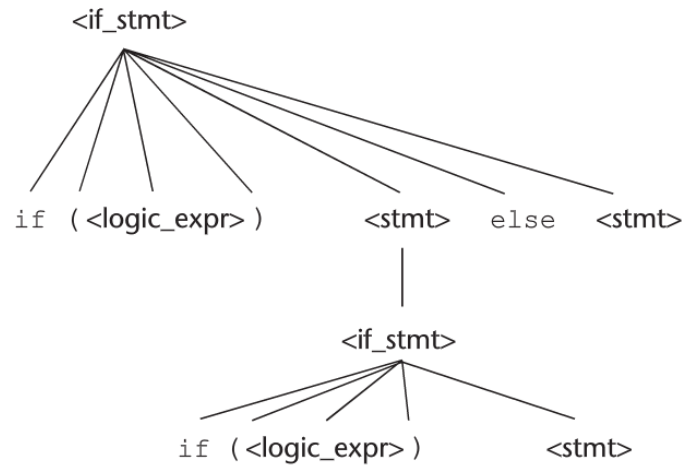
- BNF rules for the Java **`if`** statement:

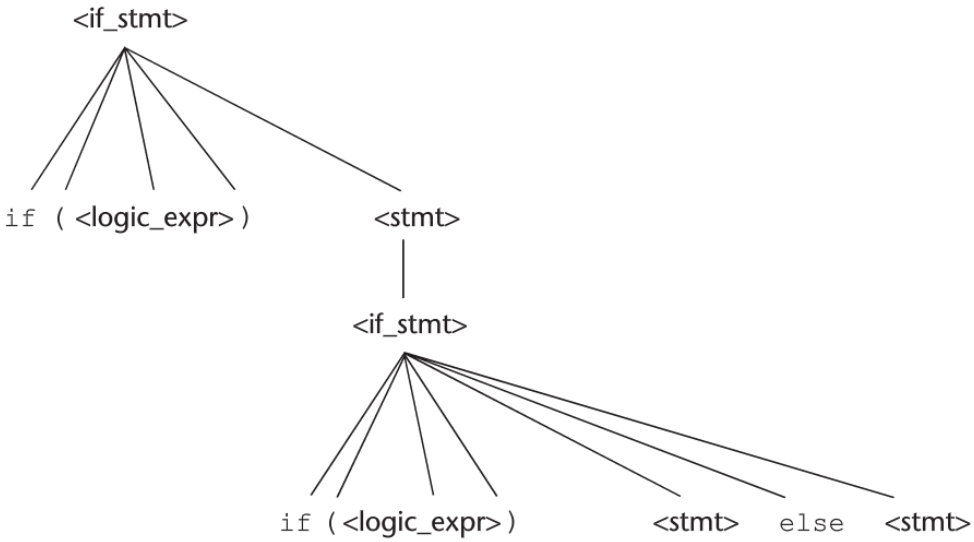
  <if_stmt> → **`if`** (<logic_expr>) <stmt>
  | **`if`** (<logic_expr>) <stmt> **`else`** <stmt>

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

  `if` (<logic_expr>) **`if`** (<logic_expr>) <stmt> **`else`** <stmt>

  The following parse trees show the ambiguity of this sentential form:





An Unambiguous Grammar for `if-else` (Continued)

- The rule for **`if`** constructs in most languages is that an **`else`** clause is matched with the nearest previous unmatched **`if`**.

- Therefore, between an **`if`** and its matching **`else`**, there cannot be an **`if`** statement without an **`else`** (an "unmatched" statement).

- To make the grammar unambiguous, two new nonterminals are added, representing matched statements and unmatched statements:

<stmt> → <matched> | <unmatched>
<matched> → **`if`** (<logic_expr>) <matched> **`else`** <matched>
| any non-if statement
<unmatched> → **`if`** (<logic_expr>) <stmt>
| **`if`** (<logic_expr>) <matched> **`else`** <unmatched>

Extended BNF

- Because of minor inconveniences in BNF, it has been extended in several ways. These extensions do not enhance the descriptive power of BNF.

- Most extended versions are called Extended BNF, or simply EBNF, even though they are not all exactly the same.

- Three extensions are commonly included in EBNF.

  Square brackets:

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

  Curly braces:

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

  Parentheses:

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

- The brackets, braces, and parentheses are **metasymbols.** In cases where these metasymbols are also terminal symbols in the language being described, the instances that are terminal symbols can be underlined or quoted.

Extended BNF (Continued)

- 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
```

- Although EBNF is more concise than BNF, it does not convey as much information. For example, the BNF rule

  <expr> → <expr> + <term>

  forces the + operator to be left associative, whereas the EBNF rule

  <expr> → <term> {+ <term>}

  does not.

- Some versions of EBNF allow a numeric superscript to be attached to the right brace to indicate an upper limit to the number of times the enclosed part can be repeated.

- Some versions use a plus (+) superscript to indicate one or more repetitions:

  <compound> → **begin** {<stmt>}$^{+}$ **end**

Extended BNF (Continued)

- A different version of EBNF is often used for describing the syntax of C-like languages.
- Nonterminals are not enclosed in brackets.
- A colon is used instead of an arrow, and the RHS of a rule is placed on the next line.
- Alternative RHSs are placed on separate lines (vertical bars are not used).
- The subscript $_{opt}$ is used to indicate an optional part of an RHS:

  ConstructorDeclarator:
  SimpleName (*FormalParameterList*$_{opt}$)

- Alternative RHSs can also be indicated by the using the words “one of”:

  AssignmentOperator: one of
  `=  *=  /=  %=  +=  -=  <<=  >>=  >>>=  &=  ^=  |=`

Grammars and Recognizers

- Given a context-free grammar, a recognizer for the language generated by the grammar can be algorithmically constructed.

- A number of software systems have been developed that perform this construction. `yacc` (*y*et *a*nother *c*ompiler-*c*ompiler) was one of the first.

Attribute Grammars

- An **attribute grammar** can be used to describe more of the structure of a programming language than is possible with a context-free grammar.

- Attribute grammars are useful because some language rules (such as type compatibility) are difficult to specify with BNF.

- Other language rules cannot be specified in BNF at all, such as the rule that all variables must be declared before they are referenced.

- Rules such as these are considered to be part of the **static semantics** of a language, not part of the language's syntax. The term "static" indicates that these rules can be checked at compile time.

- Attribute grammars, designed by Donald Knuth, can describe both syntax and static semantics.

- Attribute grammars are context-free grammars to which the following have been added:

  Attributes are properties that can have values assigned to them.
  Attribute computation functions (semantic functions) specify how attribute values are computed.
  Predicate functions state the static semantic rules of the language.

Attribute Grammars Defined

- An attribute grammar is a grammar with the following additional features:

  A set of attributes A(X) for each grammar symbol X.
  A set of semantic functions and a possibly empty set of predicate functions for each grammar rule.

- 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.
  Inherited attributes pass semantic information down and across a tree.

- For a rule $X_0 \rightarrow X_1 \ldots X_n$, the synthesized attributes of X_0 are computed with semantic functions of the form $S(X_0) = f(A(X_1),\ldots,A(X_n))$.

- Inherited attributes of symbols X_j, $1 \leq j \leq n$ (in the rule $X_0 \rightarrow X_1 \ldots X_n$), are computed with a semantic function of the form $I(X_j) = f(A(X_0),\ldots,A(X_n))$.

  To avoid circularity, inherited attributes are often restricted to functions of the form $I(X_j) = f(A(X_0),\ldots,A(X_{j-1}))$.

- A predicate function is a Boolean expression on the union of the attribute set $\{A(X_0),\ldots,A(X_n)\}$ and a set of literal attribute values. A derivation is allowed to continue only if every predicate associated with every nonterminal is true.

- A parse tree of an attribute grammar is the parse tree based on its underlying BNF grammar, with a possibly empty set of attribute values attached to each node.

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

Intrinsic Attributes

- **Intrinsic attributes** are synthesized attributes of leaf nodes whose values are determined outside the parse tree (perhaps coming from the compiler's symbol table).

- Initially, the only attributes with values are the intrinsic attributes of the leaf nodes. The semantic functions can then be used to compute the remaining attribute values.

Examples of Attribute Grammars

- The following fragment of an attribute grammar describes the rule that the name on the **`end`** of an Ada procedure must match the procedure's name:

  Syntax rule: <proc_def> → **`procedure`** <proc_name>[1]
  <proc_body> **`end`** <proc_name>[2] `;`
  Predicate: <proc_name>[1].string == <proc_name>[2].string

  Nonterminals that appear more than once in a rule are subscripted to distinguish them.

- An attribute grammar can be used to check the type rules of a simple assignment statement with the following syntax and semantics:

  The only variable names are `A`, `B`, and `C`.
  The right side of an assignment can be a variable or the sum of two variables.
  Variables are either int or real.
  Variables that are added need not have the same type. If the types are different, the result has type real.
  The variable on the left side of an assignment must have the same type as the expression on the right side.

- Syntax portion of attribute grammar for simple assignment statements:

  <assign> → <var> = <expr>
  <expr> → <var> + <var>
  | <var>
  <var> → `A` | `B` | `C`

- Attributes for nonterminals in the grammar:

  actual_type—A synthesized attribute associated with the nonterminals <var> and <expr>. Stores the actual type (int or real) of a variable or expression. In the case of a variable, the actual type is intrinsic.
  expected_type—An inherited attribute associated with the nonterminal <expr>. Stores the type expected for the expression.

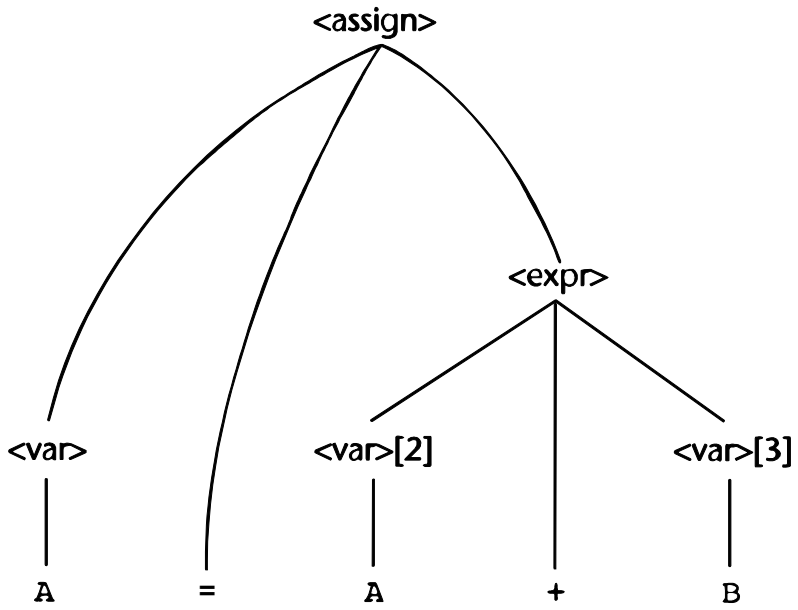
Examples of Attribute Grammars (Continued)

- Complete 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 variable name in the symbol table and returns the variable's type.

- A parse tree for the sentence `A = A + B`:



Computing Attribute Values

- The process of **decorating** the parse tree with attributes could proceed in a completely top-down order if all attributes were inherited.

- Alternatively, it could proceed in a completely bottom-up order if all the attributes were synthesized.

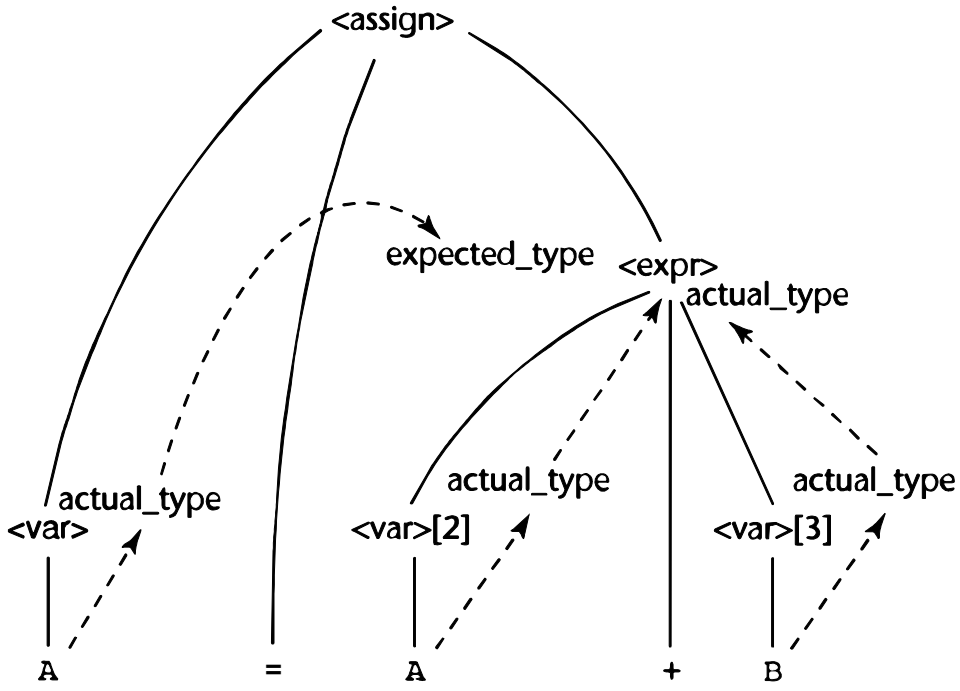
- Because our grammar has both synthesized and inherited attributes, the evaluation process cannot be in any single direction. One possible order for attribute evaluation:

  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)

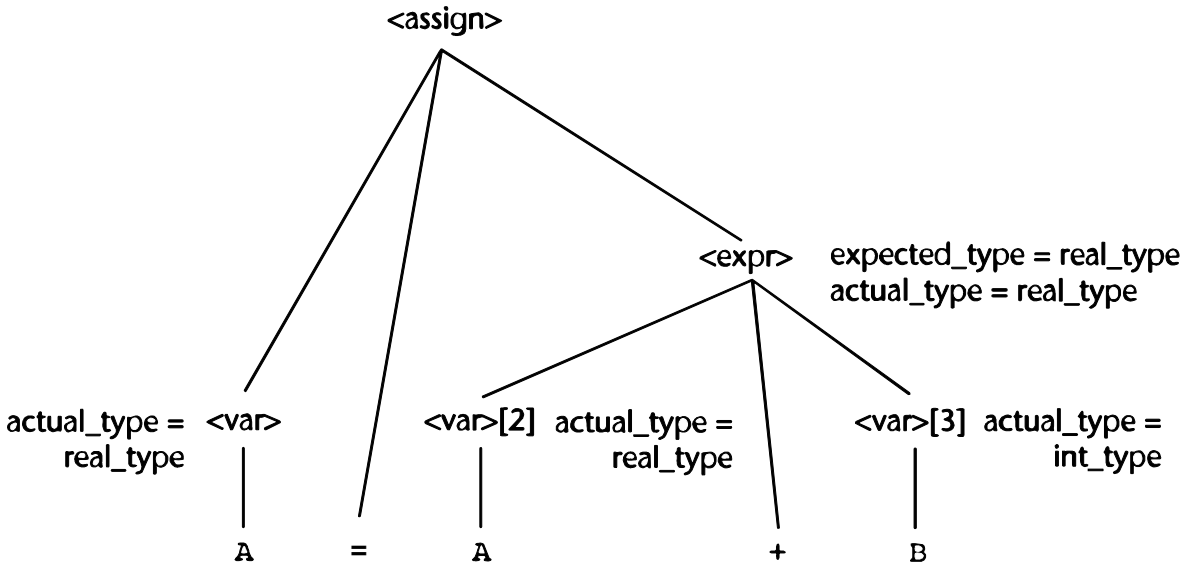
- Determining attribute evaluation order is a complex problem, requiring the construction of a graph that shows all attribute dependencies.

Computing Attribute Values (Continued)

- The following figure shows the flow of attribute values. Solid lines are used for the parse tree; dashed lines show attribute flow.



- The following tree shows the final attribute values on the nodes.



Evaluation of Attribute Grammars

- Attribute grammars have been used in a variety of applications, not all of which involve describing the syntax and static semantics of programming languages.

- An attribute grammar for a complete programming language can be difficult to write and read. Furthermore, the attribute values on a large parse tree are costly to evaluate.

- Although not every compiler writer uses attribute grammars, the underlying concepts are vital for constructing compilers.

Describing the Meanings of Programs: Dynamic Semantics

- No universally accepted notation or approach has been devised for describing **dynamic semantics,** the run-time meaning of the constructs in a programming language.

- Reasons for creating a formal semantic definition of a language:

  Programmers need to understand the meaning of language constructs in order to use them effectively.
  Compiler writers need to know what language constructs mean to correctly implement them.
  Programs could potentially be proven correct without testing.
  The correctness of compilers could be verified.
  Could be used to automatically generate a compiler.
  Would help language designers discover ambiguities and inconsistencies.

- Semantics are typically described in English. Such descriptions are often imprecise and incomplete.

Operational Semantics

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

- Using an actual machine language for this purpose is not feasible:

  The individual steps and the resulting state changes are too small and too numerous.
  The storage of a real computer is too large and complex, with several levels of memory devices plus connections to networks.

- Intermediate-level language and interpreters for idealized computers are used instead.

- Each construct in the intermediate language must have an obvious and unambiguous meaning.

- The concept of operational semantics is frequently used in programming books and language reference manuals. Example:

C Statement	*Meaning*

`for (expr1; expr2; expr3) {`
`  …`
`}`
	`  expr1;`
	`loop: if expr2 == 0 goto out`
	`  …`
	`  expr3;`
	`  goto loop`
	`out:`

  The virtual computer is the human reader, who is assumed to be able to correctly "execute" the instructions in the definition.

Operational Semantics (Continued)

- The following statements would be adequate for describing the semantics of the simple control statements of a typical programming language:

  ident = var
  ident = ident + 1
  ident = ident - 1
  `goto` label
  `if` var relop var **`goto`** label

  relop is a relational operator, ident is an identifier, and var is either an identifier or a constant.

- A slight generalization of the three assignments allows more general arithmetic expressions and assignment statements to be described:

  ident = var bin_op var
  ident = un_op var

  Multiple arithmetic data types and automatic type conversions complicate this generalization.

- Adding a few more instructions would allow the semantics of arrays, records, pointers, and subprograms to be described.

Evaluation of Operational Semantics

- The first and most significant use of formal operational semantics was to describe the semantics of PL/I. The abstract machine and the translation rules for PL/I were together named the Vienna Definition Language (VDL).

- Operational semantics can be effective as long as the descriptions are simple and informal. The VDL description of PL/I, unfortunately, is so complex that it serves no practical purpose.

- Operational semantics lacks precision because it is not based on mathematics. Other methods for describing semantics are much more formal.

Denotational Semantics

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

- A denotational description of a language entity is a function that maps instances of that entity onto mathematical objects.

- The term *denotational* comes from the fact that mathematical objects "denote" the meaning of syntactic entities.

- Each mapping function has a domain and a range:

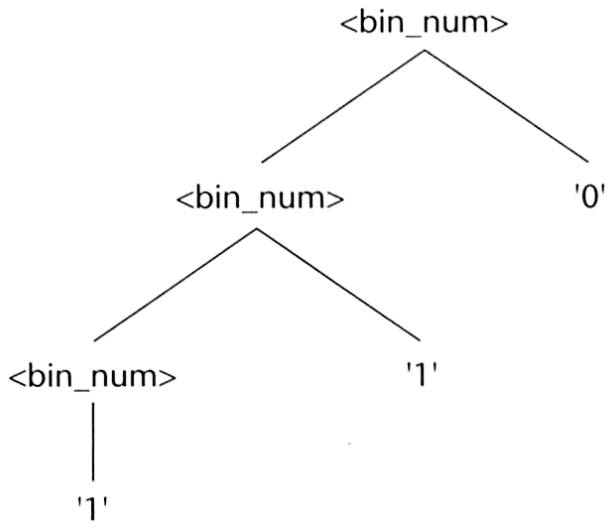
  The **syntactic domain** specifies which syntactic structures are to be mapped. The range (a set of mathematical objects) is called the **semantic domain.**

Two Simple Examples

- Grammar rules for character string representations of binary numbers:

  <bin_num> → '0'
  | '1'
  | <bin_num> '0'
  | <bin_num> '1'

- A parse tree for the binary number 110:

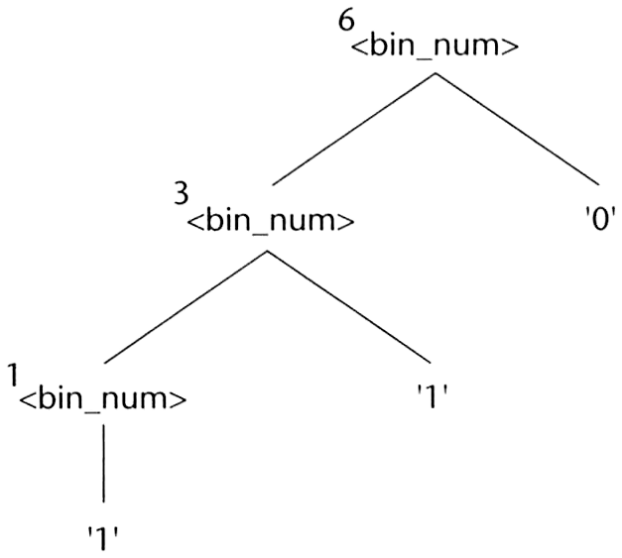


- The semantic function M_{bin} maps syntactic objects to nonnegative integers:

  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

Two Simple Examples (Continued)

- The meanings, or denoted objects (integers in this case), can be attached to the nodes of the parse tree:



- Syntax rules for decimal literals:

  <dec_num> → `'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'`
  |<dec_num>(`'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'`)

- Denotational mappings for these syntax rules:

  M_{dec}(`'0'`) = 0, M_{dec}(`'1'`) = 1, M_{dec}(`'2'`) = 2, …, 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
  …
  M_{dec}(<dec_num> `'9'`) = 10 * M_{dec}(<dec_num>) + 9

The State of a Program

- The state of a program in denotational semantics consists of the values of the program's variables.

- Formally, the state s of a program can be represented as a set of ordered pairs:

  $s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$

  Each i is the name of a variable, and the associated v's are the current values of those variables.

- Any of the v's can have the special value **undef,** which indicates that its associated variable is currently undefined.

- Let VARMAP be a function of two parameters, a variable name and the program state. The value of VARMAP(i_j, s) is v_j.

- Most semantics mapping functions for language constructs map states to states. These state changes are used to define the meanings of the constructs.

- Some constructs, such as expressions, are mapped to values, not states.

Expressions

- In order to develop a concise denotational definition of the semantics of expressions, the following simplifications will be assumed:

  No side effects.
  Operators are + and *.
  At most one operator.
  Operands are scalar integer variables and integer literals.
  No parentheses.
  Value of an expression is an integer.
  Errors never occur during evaluation; however, the value of a variable may be undefined.

  BNF description of these expressions:

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

- Mapping function for an expression:

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

  The symbol Δ= indicates the definition of a mathematical function.

Assignment Statements and Logical Pretest Loops

- Mapping function for an assignment statement:

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

The $i_j == x$ comparison is of names, not values.

- Mapping function for a logical pretest loop:

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

The functions M_{sl} and M_b are assumed to map statement lists to states and Boolean expressions to Boolean values (or **error**), respectively.

Evaluation of Denotational Semantics

- Denotational semantics have been used in programming language standards. The international standard for Modula-2, for example, uses denotational semantics.

- It is possible to use denotational language descriptions to generate compilers automatically, but the work has never progressed to the point where it can be used to generate useful compilers.

- Denotational descriptions are of little use to language users. On the other hand, they provide an excellent way to describe the semantics of a language concisely.

Axiomatic Semantics

- **Axiomatic semantics**, which is based on mathematical logic, is the most abstract technique for specifying semantics.

- Axiomatic semantics originated with the development of an approach to proving the correctness of programs.

- This approach is based on the use of assertions. An **assertion** is a logical expression that specifies constraints on program variables.

- A **precondition** is an assertion that describes any necessary constraints on program variables *before* the execution of a statement.

- A **postcondition** is an assertion that describes new constraints on program variables *after* the execution of a statement.

- Preconditions and postconditions are enclosed within braces to distinguish them from program statements.

- Example of a postcondition:

  ```
  sum = 2 * x + 1 {sum > 1}
  ```

  In this and later examples, all variables are assumed to have integer type.

- In axiomatic semantics, the precondition and postcondition for a statement specify the effect of executing the statement.

Weakest Preconditions

- The **weakest precondition** is the least restrictive precondition that will guarantee the validity of the associated postcondition.

- For the statement

  ```
  sum = 2 * x + 1 {sum > 1}
  ```

  {x > 10}, {x > 50}, and {x > 1000} are all valid preconditions. The weakest precondition is{x > 0}.

- An **inference rule** is a method of inferring the truth of one logical statement based on the truth of other logical statements.

- General form of an inference rule:

  $$\frac{S1, S2, \ldots, S_n}{S}$$

  This rule states that if S1, S2, …, and S*n* are true, then the truth of S can be inferred.

- An **axiom** is a logical statement that is assumed to be true.

- For some program statements, the computation of a weakest precondition from the statement and a postcondition can be specified by an axiom. In most cases, however, an inference rule must be used.

- An axiomatic definition of the semantics of a programming language must include an axiom or inference rule for each kind of statement in the language.

Assignment Statements

- Let x = E be a general assignment statement and Q be its postcondition. Then its weakest precondition, P, is defined as

  $P = Q_{x \rightarrow E}$

  which means that P is Q with all instances of x replaced by E.

- Example:

  ```
  a = b / 2 - 1 {a < 10}
  ```

  The weakest precondition is {`b / 2 - 1 < 10`}.

- Notation for specifying the axiomatic semantics of a statement form:

  {P} S {Q}

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

- Axiomatic semantics of the assignment statement:

  $\{Q_{x \rightarrow E}\}$ x = E {Q}

- The appearance of the left side of an assignment statement in its right side does not affect the process of computing the weakest precondition.

- Example:

  ```
  x = x + y - 3 {x > 10}
  ```

  The weakest precondition is {`x + y - 3 > 10`}.

The Rule of Consequence

- Consider the logical statement

  `{x > 5} x = x - 3 {x > 0}`

  The precondition `{x > 5}` is not the same as the assertion produced by the assignment axiom. Using this statement in a proof requires an inference rule named the **rule of consequence.**

- Rule of consequence:

  $$\frac{\{P\}\ S\ \{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

  The => symbol means "implies." S can be any program statement.

- The rule of consequence says that a postcondition can always be weakened and a precondition can always be strengthened.

- Example use of the rule of consequence:

  $$\frac{\{x - 3 > 0\}\ x = x - 3\ \{x > 0\},\ (x > 5) \Rightarrow (x - 3 > 0),\ (x > 0) \Rightarrow (x > 0)}{\{x > 5\}\ x = x - 3\ \{x > 0\}}$$

Sequences

- Inference rule for a sequence of two statements:

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

- Consider the following sequence and postcondition:

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

  Weakest precondition for the second assignment: {`y + 3 < 10`}.

  Weakest precondition for the first assignment: {`3 * x + 1 + 3 < 10`}.

Selection

- Inference rule for selection statements with `else` clauses:

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

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

- Example:

```
if x > 0 then
  y = y - 1
else
  y = y + 1
```

  Assume that the postcondition is `{y > 0}`. Applying the axiom for assignment to the `then` clause

```
y = y - 1 {y > 0}
```

  produces the precondition `{y - 1 > 0}`. Applying the same axiom to the `else` clause

```
y = y + 1 {y > 0}
```

  produces `{y + 1 > 0}`. Because `{y - 1 > 0}` => `{y + 1 > 0}`, the rule of consequence allows `{y - 1 > 0}` to be used as the precondition of the selection statement.

Logical Pretest Loops

- Computing the weakest precondition for a logical pretest (`while`) loop is inherently difficult because of the need to find a **loop invariant.**

- Inference rule for a `while` loop:

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

  I is the loop invariant.

- Another complicating factor for `while` loops is the question of loop termination.

  Proving **total correctness** involves showing that the loop satisfies the specified postcondition and always terminates.
  Proving **partial correctness** involves showing that the loop satisfies the specified postcondition, without proving that it always terminates.

- Proving the total correctness of

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

  requires showing that all of the following are true:

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

Logical Pretest Loops (Continued)

- Consider the following loop:

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

  {y <= x} can be used as the loop invariant.

- This loop invariant can also be used as the precondition for the `while` statement. The goal is now to show that the loop

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

  satisfies the four criteria for loop correctness:

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

- It is easy to show that the first three criteria are satisfied. Loop termination is also clear, since `y` increases with each iteration until it is eventually equal to `x`.

Program Proofs

- Consider the problem of proving the following program correct:

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

- Informal proof:

  Applying the assignment axiom to the last statement yields the precondition

  `{x = B AND t = A}`

  Using this as the postcondition for the middle statement yields the precondition

  `{y = B AND t = A}`

  Finally, this can be used as the postcondition for the first statement, which yields the precondition

  `{y = B AND x = A}`

  This assertion is mathematically equivalent to the precondition for the entire program (because AND is commutative).

- Formal proof:

  1. `{y = B AND x = A} t = x; {y = B AND t = A}` — Assignment axiom
  2. `(x = A AND y = B) => (y = B AND x = A)`
  3. `(y = B AND t = A) => (y = B AND t = A)`
  4. `{x = A AND y = B} t = x; {y = B AND t = A}` — Rule of consequence (1, 2, 3)
  5. `{y = B AND t = A} x = y; {x = B AND t = A}` — Assignment axiom
  6. `{x = A AND y = B} t = x; x = y; {x = B AND t = A}` — Sequence rule (4, 5)
  7. `{x = B AND t = A} y = t; {x = B AND y = A}` — Assignment axiom
  8. `{x = A AND y = B} t = x; x = y; y = t; {x = B AND y = A}` — Sequence rule (6, 7)

Program Proofs (Continued)

- The following program contains a loop, making it harder to prove correct:

```
{n >= 0}
count = n;
fact = 1;
while count <> 0 do
    fact = fact * count;
    count = count - 1;
end
{fact = n!}
```

- The loop computes the factorial function in order of the last multiplication first, so part of the invariant can be

  `fact = (count + 1) * (count + 2) * … * (n - 1) * n`

  To prove loop termination, it is important that `count` always be nonnegative, leading to the following invariant:

  I = `(fact = (count + 1) * … * n) AND (count >= 0)`

- Again, P can be the same as I. It is easy to show that the loop

```
{(fact = (count + 1) * … * n) AND (count >= 0)}
while count <> 0 do
    fact = fact * count;
    count = count - 1;
end
{fact = n!}
```

  satisfies the four criteria for loop correctness:

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

Evaluation of Axiomatic Semantics

- Defining the semantics of a complete programming language using the axiomatic method is difficult.

- Axiomatic semantics is a powerful tool for research into program correctness proofs. It also provides an excellent framework for reasoning about programs.

- Its usefulness in describing the meaning of programming languages to language users and compiler writers is, however, highly limited.