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

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., `*, sum, begin, for, =`)
  - The lexemes include its numeric literature, operators, identifiers, and special words among others.
- A *token* is a category of lexemes (e.g., identifier)

Formal Definition of Languages

- **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
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages

Backus-Naur Form (BNF)

- Backus-Naur Form (1959)
  - Invented by John Backus to describe Algol 58
  - BNF is equivalent to context-free grammars
  - BNF is a *metalanguage* used to describe another language
  - In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols*)

BNF Fundamentals

- Non-terminals: BNF abstractions
- Terminals: lexemes and tokens
- Grammar: a collection of rules
  - Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

BNF Rules

- A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of *terminal* and *nonterminal* symbols
- A grammar is a finite nonempty set of rules
- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

Describing Lists

- Syntactic lists are described using recursion

```
<ident_list>  → ident
              | ident, <ident_list>
```

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar

```
<program> → <stmts>
 <stmts> → <stmt> | <stmt> ; <stmts>
 <stmt> → <var> = <expr>
 <var> → a | b | c | d
 <expr> → <term> + <term> | <term> - <term>
 <term> → <var> | const
```

An Example Derivation

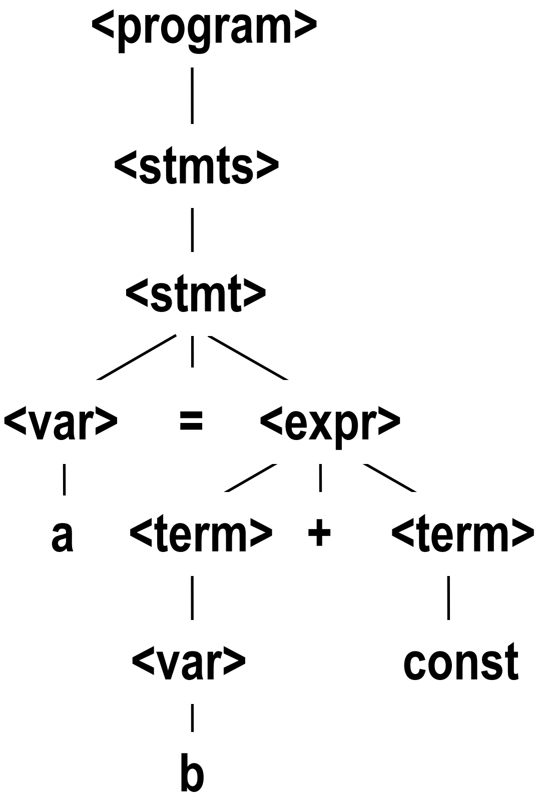
```
<program> => <stmts> => <stmt>
                     => <var> = <expr> => a =<expr>
                     => a = <term> + <term>
                     => a = <var> + <term>
                     => a = b + <term>
                     => a = b + const
```

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

Parse Tree

- A hierarchical representation of a derivation



<program>

<stmts>

<stmt>

<var> = <expr>

a <term> + <term>

<var> const

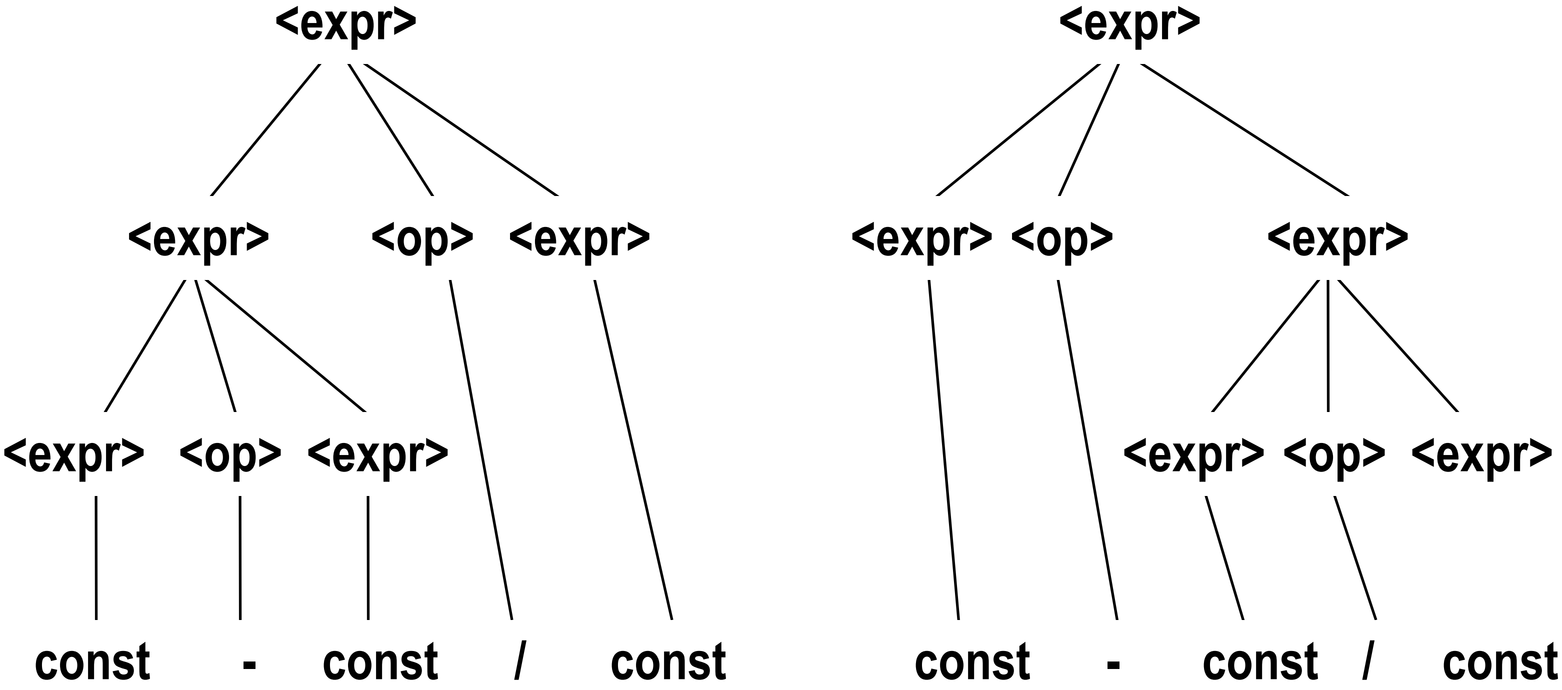
b

Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees
- In many cases, an ambiguous grammar can be rewritten to be unambiguous and still generate the desired language.

An Ambiguous Expression Grammar

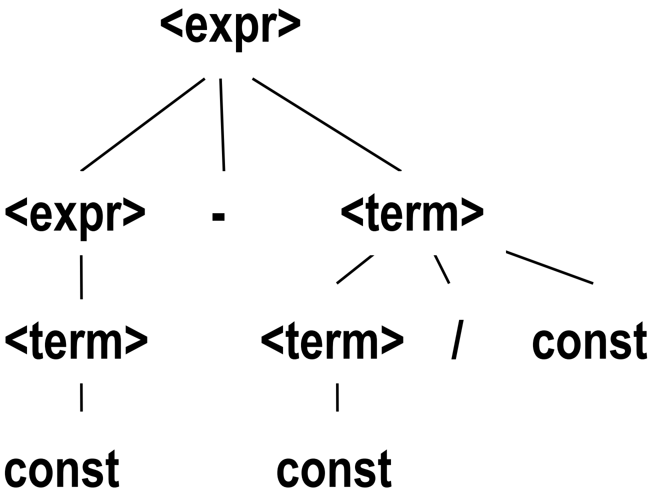
```
<expr> → <expr> <op> <expr>  |  const
<op> → /  |  -
```



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

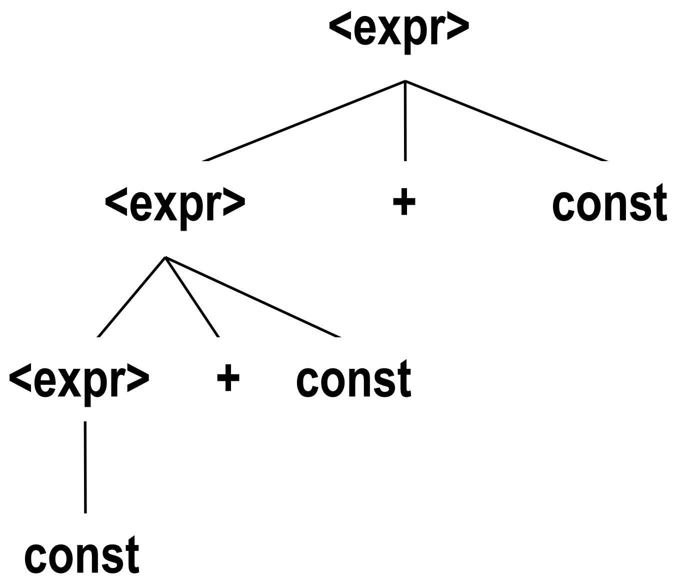
```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Extended BNF

- Optional parts are placed in brackets [ ]

```
<proc_call> -> ident [(<expr_list>)]
```

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

```
<term> → <term> (+|-) const
```

- Repetitions (0 or more) are placed inside braces { }

```
<ident> → letter {letter|digit}
```

BNF and EBNF

- BNF

```
<expr>  → <expr> + <term>
          | <expr> - <term>
            | <term>
 <term>  → <term> * <factor>
          | <term> / <factor>
          | <factor>
```

- EBNF

```
<expr>  → <term> {(+ | -) <term>}
 <term>  → <factor> {(* | /) <factor>}
```

Static Semantics

- Only indirectly related to the meaning of programs during execution ; rather it has to d o with the legal forms of programs (syntax rather than semantics)
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:
  - Context-free, but cumbersome (e.g., types of operands in expressions)
  - Non-context-free (e.g., variables must be declared before they are used)

Attribute Grammars

- Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes

- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

Attribute Grammars : Definition

- Def: 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 non-terminals in the rule called attribute computational functions or semantic functions.
  - Each rule has a (possibly empty) set of predicates to check for attribute consistency , called predicate functions.

Attribute Grammars: Definition

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
- Functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$ define *synthesized attributes*
- Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, *for* $i <= j <= n$, define *inherited attributes*
- Initially, there are *intrinsic attributes* on the leaves

Attribute Grammars: An Example

- Syntax

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

- `actual_type`: synthesized for `<var>` and `<expr>`
- `expected_type`: inherited for `<expr>`

Attribute Grammar (continued)

- **Syntax rule:** `<expr> → <var>[1] + <var>[2]`
  Semantic rules:
  ```
  <expr>.actual_type ←
  <var>[1].actual_type
  ```
  Predicate:
  ```
  <var>[1].actual_type == <var>[2].actual_type
  <expr>.expected_type == <expr>.actual_type
  ```

- **Syntax rule:** `<var> → id`
  Semantic rule:
  ```
  <var>.actual_type ← 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)

```
<expr>.expected_type ← inherited from parent

<var>[1].actual_type ← lookup (A)
<var>[2].actual_type ← lookup (B)
<var>[1].actual_type =? <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type
<expr>.actual_type =? <expr>.expected_type
```

Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Several needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies

Operational 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
- To use operational semantics for a high-level language, an intermediate language could be introduced which should be easy to understand and self descriptive.
  - a virtual machine could be implemented for the intermediate language as well.

Operational Semantics

- There are different levels of uses of operational semantics:
  - At the highest level, the final result of the program execution is of interest: Natural Operational Semantics
  - At the lowest level, the complete sequence of state changes (caused by execution of each instruction) is of interest: Natural Operational Semantics

  - See the example in page 142

Operational Semantics (continued)

- Uses of operational semantics:
  - Language manuals and textbooks
  - Teaching programming languages

- Two different levels of uses of operational semantics:
  - Natural operational semantics
  - Structural operational semantics

- Evaluation
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g.,VDL)

Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)

Denotational Semantics (continued)

- The process of building a denotational specification for a language Define a mathematical object for each language entity
  - Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
- The meaning of language constructs are defined by only the values of the program's variables

Denotation Semantics vs Operational Semantics

- In operational semantics, the state changes are defined by coded algorithms
- In denotational semantics, the state changes are defined by rigorous mathematical functions

Denotational Semantics: Program State

- The state of a program is the values of all its current variables

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

- Let **VARMAP** be a function that, when given a variable name and a state, returns the current value of the variable

  $\mathrm{VARMAP}(i_j, s) = v_j$

Decimal Numbers

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

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, 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

Expressions

- Map expressions onto $Z \cup \{\text{error}\}$
- We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression

<expr>□ <dec_num>|<var>|<binary_expr>

<Binary_expr>□ <left_expr><operator><right_expr>

<left_expr>□ <dec_num>|<var>

<right_expr>□ <dec_num>|<var>

<operator>□+|*

3.5 Semantics (cont.)

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

Assignment Statements

- Maps state sets to state sets

```
Ma(x := E, s) Δ=
    if Me(E, s) == error
       then error
       else s' =
  {<$i_1$',$v_1$'>,<$i_2$',$v_2$'>,...,<$i_n$',$v_n$'>},
                where for j = 1, 2, ..., n,
                    $v_j$' = VARMAP($i_j$, s) if $i_j$ <> x
                         = Me(E, s) if $i_j$ == x
```

Logical Pretest Loops

- Maps state sets to state sets

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

Loop Meaning

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors
- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions
- Recursion, when compared to iteration, is easier to describe with mathematical rigor

Evaluation of Denotational Semantics

- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
- Has been used in compiler generation systems
- Because of its complexity, they are of little use to language users

Axiomatic Semantics

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of expressions to other expressions)
- The expressions are called *assertions*

Axiomatic Semantics (continued)

- 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*
- A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition

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}`

Program Proof Process

- The postcondition for the entire 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 specification, the program is correct.

Axiomatic Semantics: Axioms

- An axiom for assignment statements (x = E): $\{Q_{x->E}\}$ x = E {Q}

- The Rule of Consequence:

Axiomatic Semantics: Axioms

- An inference rule for sequences
  {P1} S1 {P2}
  {P2} S2 {P3}

Axiomatic Semantics: Axioms

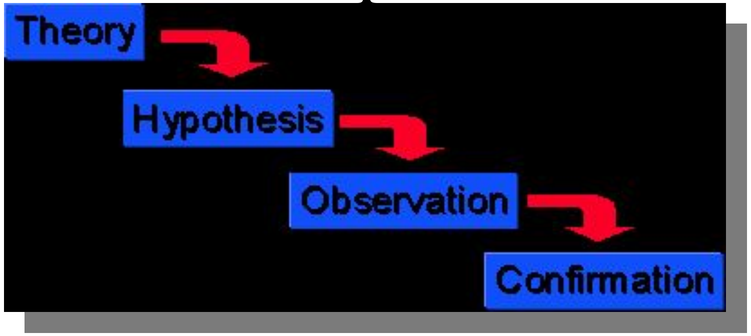
- An inference rule for logical pretest loops

  {P} while B do S end {Q}

  where I is the loop invariant (the inductive hypothesis)

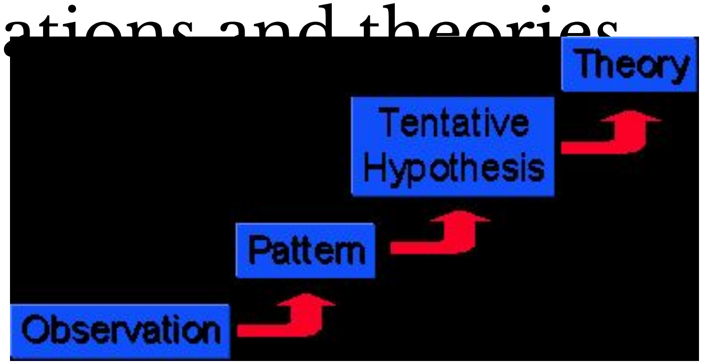
Induction vs. Deduction

- Deductive reasoning works from the more general to the more specific.



Theory
Hypothesis
Observation
Confirmation

- Inductive reasoning works the other way, moving from specific observations to broader generalizations and theories.



Theory
Tentative Hypothesis
Pattern
Observation

Axiomatic Semantics: Axioms

- 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, is implied
  - The loop terminates

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, denotational