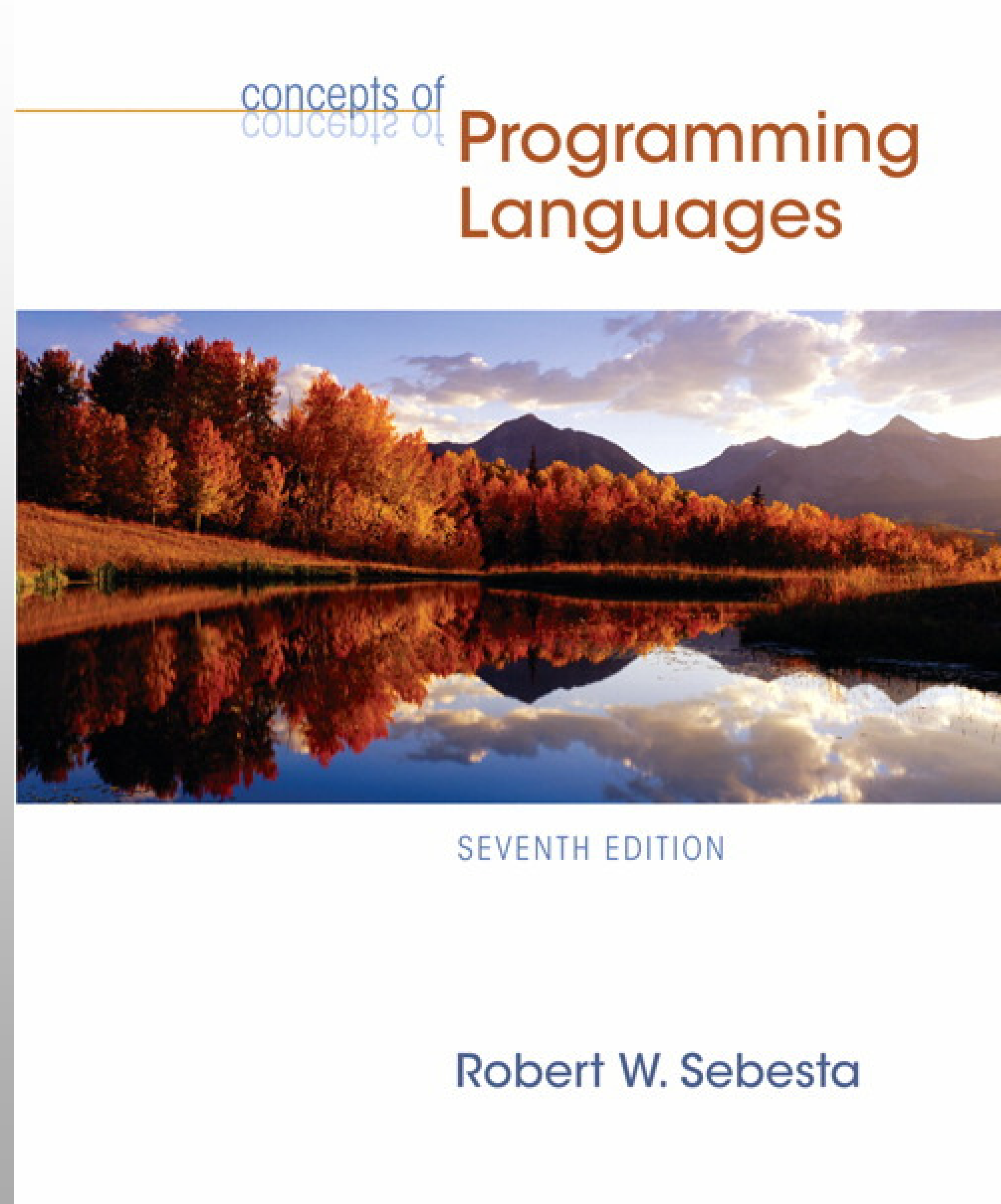
Chapter 3

Describing Syntax and Semantics



ISBN 0-321-33025-0

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`)
- 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>

<number> → <digit> | <number> <digit>

<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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

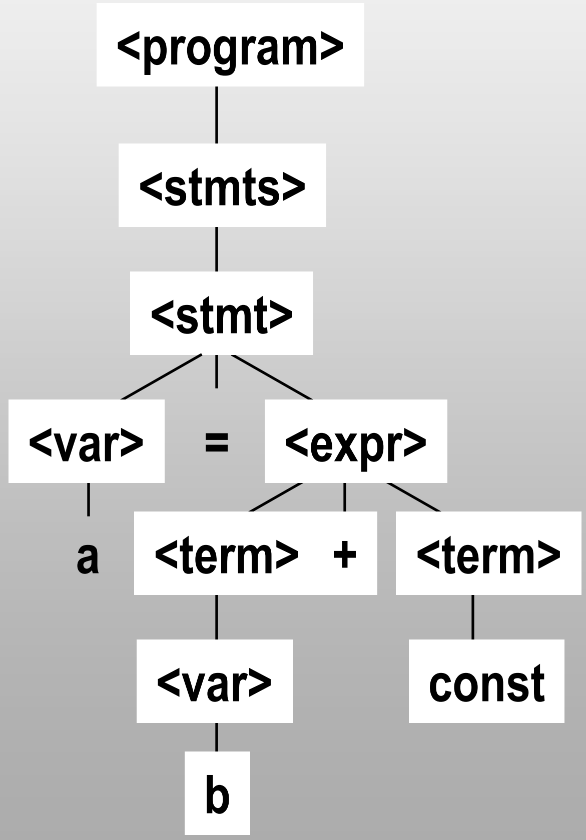
```
<program> => <stmts> => <stmt>
                     => <var> = <expr> => a =<expr>
                     => a = <term> + <term>
                     => a = <var> + <term>
                     => a = b + <term>
                     => a = b + const
```

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



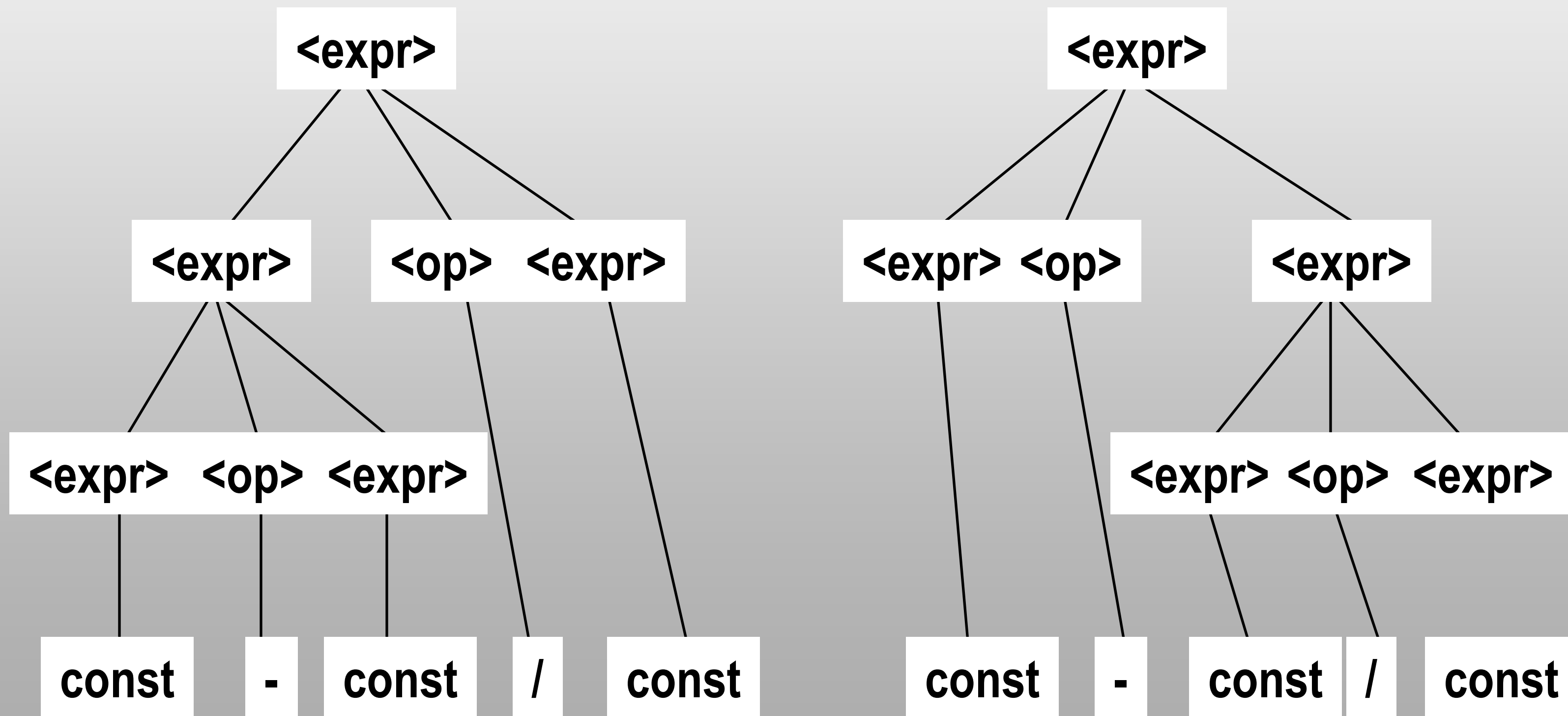
Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

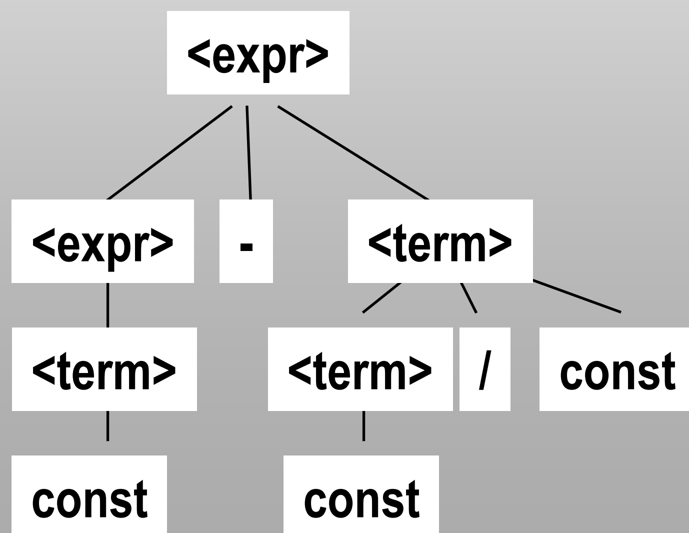
<op> → / | -



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
```



dangling else

- An example in programming languages is the "dangling else"

  if A then if B then C else D

Is this

  if A then (if B then C else D)

or

  if A then (if B then C) else D

?

Sometimes it is possible to rewrite the grammar productions to eliminate ambiguity

if-then-else

The meaning of the *if-then-else* statement is the same in Pascal and Modula-2, but the syntax differs.

Pascal:

if <boolean expression> **then** <statement> **else** <statement>

Modula-2:

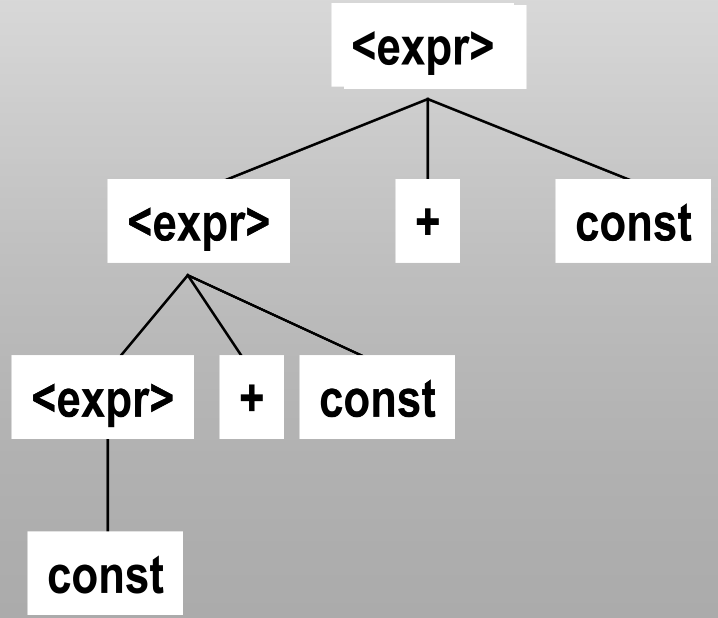
IF <boolean expression> **THEN** <statement sequence> **ELSE**

<statement sequence> **END**

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>}
```

BNF vs EBNF

Extended BNF (EBNF) is a more convenient way of describing CFGs than is BNF.

EBNF is *no more powerful* than BNF: the languages described are still the CFLs, and any EBNF grammar can be transformed into BNF.

EBNF: Grouping

Grouping can be eliminated by introducing a new Non-terminal for each group:

$$A \rightarrow \ldots\ldots\ (\alpha_1)\ \ldots\ldots\ (\alpha_k)\ \ldots\ldots$$

is equivalent to

$$A \rightarrow \ldots A_1\ \ldots\ldots A_k\ \ldots\ldots$$

$$A_1 \rightarrow \alpha_1$$

$$\ldots$$

$$\ldots$$

$$A_k \rightarrow \alpha_k$$

EBNF: Grouping of Alternatives

- Alternatives in a group does not add anything new:
- $A \rightarrow B\ (C \mid D \mid E)\ F$
- is by elimination of grouping equivalent to
- $A \rightarrow BA_1F$
- $A_1 \rightarrow C \mid D \mid E$
- which in turn is just a shorter way of writing
- $A \rightarrow BA_1F$
- $A_1 \rightarrow C$
- $A_1 \rightarrow D$
- $A_1 \rightarrow E$

EBNF: Iteration (1)

The iterative construct can be replaced by explicit recursion:

$$A \rightarrow \ldots\ldots \ \{ B \} \ldots\ldots$$

is equivalent to (left recursion)

$$A \rightarrow \ldots\ldots. \ A_1 \ \ldots \ \ldots$$

$$A_1 \rightarrow \epsilon \mid A_1 \ B$$

or (right recursion)

$$A \rightarrow \ldots\ldots \ A_1 \ \ldots\ldots$$

$$A_1 \rightarrow \epsilon \mid B \ A_1$$

EBNF: Iteration (2)

The grammar G with the single production

$S \rightarrow a\{bb\}c$

generates the language

$L(G) = \{ a(bb)^i c \mid i \geq 0 \}$

$= \{$ ac; abbc, abbbbc, abbbbbbc, ...$\}$

An equivalent left-recursive grammar is

$S \rightarrow aAc$

$A \rightarrow \epsilon \mid Abb$

Substitution

If we use EBNF, we can substitute the RHS of a production for uses of the non-terminal it defines, as long as *all alternatives* are included:

$A \rightarrow X\ B\ Y$

$B \rightarrow C \mid D$

$B \rightarrow E$

can be transformed into

$A \rightarrow X\ (C \mid D \mid E)\ Y$

$B \rightarrow C \mid D$

$B \rightarrow E$

Left Factoring (1)

If we use EBNF, a common prefix among a group of productions can be factored out.

Consider:

$$A \rightarrow XY\ X \mid XY\ ZZY$$

After left factoring:

$$A \rightarrow XY\ (X \mid ZZY)$$

Left Factoring (2)

Example :

single-cmd $\rightarrow$ v-name := expression
| if expression **then** single-cmd
| if expression **then** single-cmd
else single-cmd

After left factoring:

single-cmd $\rightarrow$ v-name := expression
| if expression **then** single-cmd
(ϵ | **else** single-cmd)

Elimination of Left Recursion (1)

- Certain kinds of parsers cannot handle *left-recursive* productions.
- If it is desired to use such a parser, but the grammar is left-recursive, then the grammar first has to be transformed into an equivalent grammar that is *not* left-recursive.
- We will first see how that can be done for *immediate* left recursion; i.e., productions of the form

$$A \rightarrow A\ \alpha \quad \text{(where } \alpha \text{ is not } \epsilon\text{).}$$

Elimination of Left Recursion (2)

For each non-terminal A defined by some left- recursive production, group the productions for A

$A \rightarrow A\ \alpha_1 \mid A\ \alpha_2 \mid \ldots \mid A\ \alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

such that no β_i begins with an A.

Then replace the A productions by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A' \mid \epsilon$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid$$

Assumption: no α_i is .

Elimination of Left Recursion (3)

Consider the (immediately) left-recursive grammar:

$S \rightarrow A \mid B$

$A \rightarrow ABc \mid Add \mid a \mid aa$

$B \rightarrow Bee \mid b$

Terminal strings derivable from B include:

b, bee, beeee, beeeeee

Terminal strings derivable from A include:

a, aa, add, aadd, adddd, aadddd, abc, aabc, abeec, aabeec, abeecbeec, aabeeeecddbeec

Elimination of Left Recursion (4)

Let us do a leftmost derivation of

aabeeeecddbeec:

S ➔ A
➔ ABc
➔ AddBc
➔ ABcddBc
➔ aaBcddBc
➔ aaBeecddBc
➔ aaBeeeecddBc
➔ aabeeeecddBc
➔ aabeeeecddBeec
➔ aabeeeecddbeec

Elimination of Left Recursion (5)

Here is the grammar again:

$S \rightarrow A \mid B$

$A \rightarrow ABc \mid Add \mid a \mid aa$

$B \rightarrow Bee \mid b$

An equivalent right-recursive grammar:

$S \rightarrow A \mid B$

$A \rightarrow aA' \mid aaA'$

$A' \rightarrow BcA' \mid ddA' \mid \epsilon$

$B \rightarrow bB'$

$B' \rightarrow eeB' \mid \epsilon$

Elimination of Left Recursion (6)

Derivation of aabeeeecddbeec in the new grammar:

S ➔ A ➔ aaA’ ➔ aaBcA’
➔ aabB’cA’
➔ aabeeB’cA’
➔ aabeeeeB’cA’
➔ aabeeeecA’
➔ aabeeeecddA’
➔ aabeeeecddBcA’
➔ aabeeeecddbB’cA’
➔ aabeeeecddbeeB’cA’
➔ aabeeeecddbeecA’
➔ aabeeeecddbeec

Elimination of Left Recursion (7)

To eliminate *general* left recursion:

- first transform the grammar into an *Immediately* left-recursive grammar through
- systematic substitution then proceed as before.

Elimination of Left Recursion (8)

For example, the generally left-recursive grammar

$$A \rightarrow Ba$$
$$B \rightarrow Ab \mid Ac \mid \epsilon$$

is first transformed into the immediately left-recursive grammar

$$A \rightarrow Aba$$
$$A \rightarrow Aca$$
$$A \rightarrow a$$

Elimination of Left Rec. example

Identifier $\rightarrow$ Letter
| Identifier Letter
| Identifier Digit

Left factoring yields:

Identifier $\rightarrow$ Letter
| Identifier (Letter | Digit)

The recursion can now be eliminated by using the iterative EBNF construct:

Identifier $\rightarrow$ Letter { Letter | Digit }

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
  - Compiler design (static semantics checking)

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: 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
- 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

- To use operational semantics for a high-level language, a virtual machine is needed
- A *hardware* pure interpreter would be too expensive
- A *software* pure interpreter also has problems
  - The detailed characteristics of the particular computer would make actions difficult to understand
  - Such a semantic definition would be machine-dependent

Operational Semantics (continued)

- A better alternative: A complete computer simulation
- The process:
  - Build a translator (translates source code to the machine code of an idealized computer)
  - Build a simulator for the idealized computer
- Evaluation of operational semantics:
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g., VDL), it was used for describing semantics of PL/I.

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:

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

Axiomatic Semantics: Axioms

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

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

Axiomatic Semantics: Axioms

- An inference rule for logical pretest loops

{P} while B do S end {Q}

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

where I is the loop invariant (the inductive hypothesis)

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

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