Chapter 3

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

Semantics **- the meaning of the expressions, statements, and program units**

Who must use language definitions?

1. **Other language designers**
2. **Implementors**
3. **Programmers (the users of the language)**

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 approaches to describing syntax:

1. **Recognizers - used in compilers**
2. **Generators - what we'll study**

Chapter 3

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 Normal Form (1959)

- Invented by John Backus to describe Algol 58
- BNF is equivalent to context-free grammars

A *metalanguage* is a language 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*)

e.g.

```
<while_stmt> -> while <logic_expr> do <stmt>
```

This is a *rule*; it describes the structure of a `while` statement

Chapter 3

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
```

Syntactic lists are described in BNF 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)

Chapter 3

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
```

Every string of symbols in the derivation is a *sentential form*

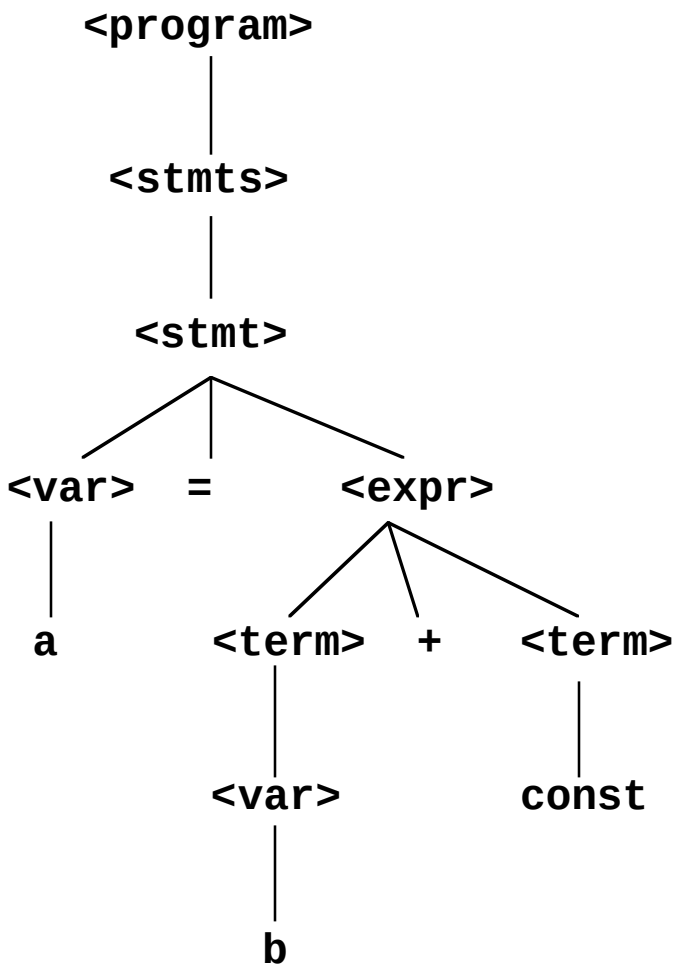
A *sentence* is a sentential form that has only terminal symbols

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

Chapter 3

A *parse tree* is a hierarchical representation of a derivation

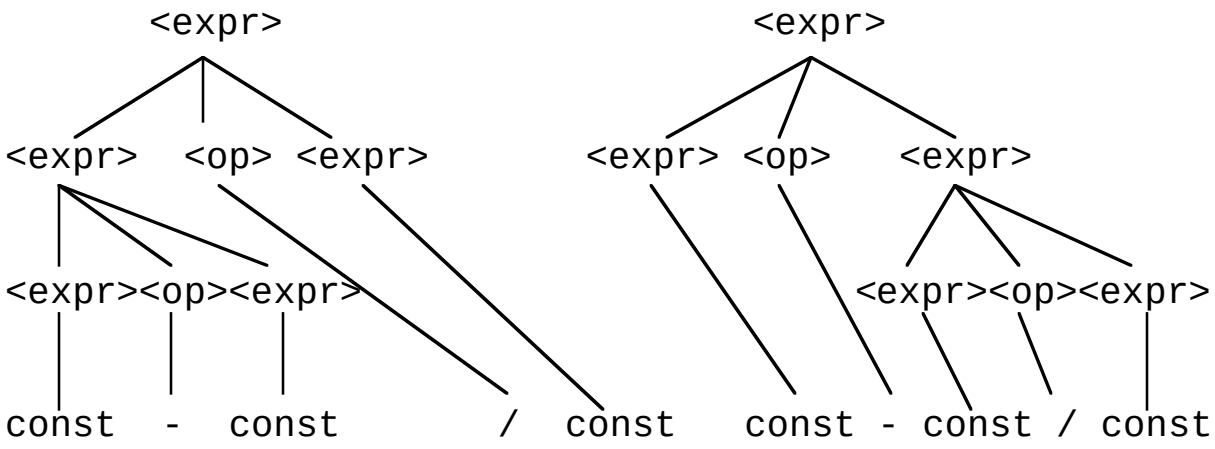


A grammar is *ambiguous* iff it generates a sentential form that has two or more distinct parse trees

Chapter 3

An ambiguous expression grammar:

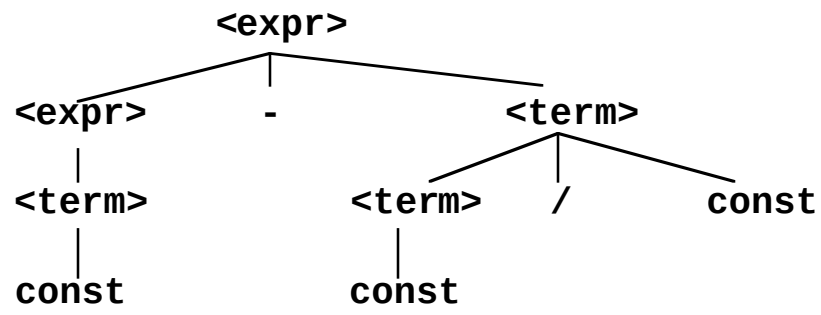
```
<expr> -> <expr> <op> <expr>  |  const
<op> -> /  |  -
```



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

An unambiguous expression grammar:

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



!

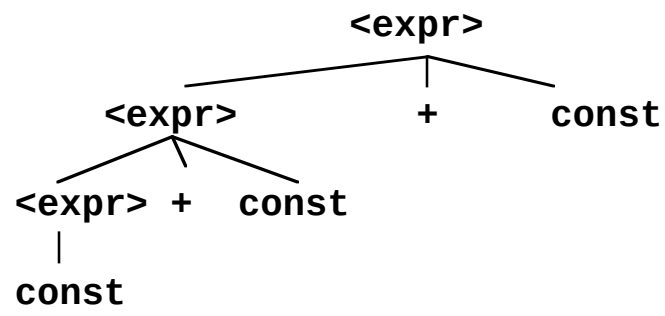
Chapter 3

```
<expr> => <expr> - <term> => <term> - <term>
       => const - <term>
       => const - <term> / const
       => const - const / const
```

Operator associativity can also be indicated by a grammar

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

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



Extended BNF (just abbreviations):

1. **Optional parts are placed in brackets (`[]`)**
   `<proc_call> -> ident [ ( <expr_list>)]`
2. **Put alternative parts of RHSs in parentheses and separate them with vertical bars**
   `<term> -> <term> (+ | -) const`
3. **Put repetitions (0 or more) in braces (`{}`)**
   `<ident> -> letter {letter | digit}`

Chapter 3

BNF:

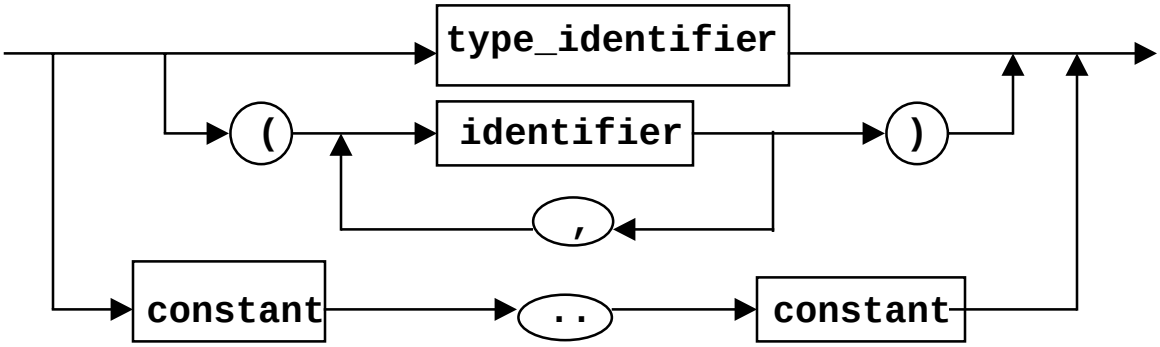
```
<expr> -> <expr> + <term>
        | <expr> - <term>
        | <term>
<term> -> <term> * <factor>
        | <term> / <factor>
        | <factor>
```

EBNF:

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

***Syntax Graphs* - put the terminals in circles or ellipses and put the nonterminals in rectangles; connect with lines with arrowheads**

e.g., Pascal type declarations



Chapter 3

Recursive Descent Parsing

- Parsing is the process of tracing or constructing a parse tree for a given input string

- Parsers usually do not analyze lexemes; that is done by a lexical analyzer, which is called by the parser

- A *recursive descent parser* traces out a parse tree in top-down order; it is a top-down parser

- Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all sentential forms that the nonterminal can generate

- The recursive descent parsing subprograms are built directly from the grammar rules

- Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars

Example: For the grammar:

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

We could use the following recursive descent parsing subprogram (this one is written in C)

```
void term() {
  factor();  /* parse the first factor*/
  while (next_token == ast_code ||
         next_token == slash_code) {
    lexical();  /* get next token */
    factor();  /* parse the next factor */
  }
}
```

Static semantics (have nothing to do with meaning)

Categories:

1. **Context-free but cumbersome (e.g. type checking)**
2. **Noncontext-free (e.g. variables must be declared before they are used)**

Chapter 3

Attribute Grammars (AGs) (Knuth, 1968)

- Cfgs cannot describe all of the syntax of programming languages

- Additions to cfgs to carry some semantic info along through parse trees

Primary value of AGs:

1. Static semantics specification
2. Compiler design(static semantics checking)

Def: An *attribute grammar* is a cfg G = (S, N, T, P) with the following additions:

1. For each grammar symbol x there is a set A(x) of attribute values

2. Each rule has a set of functions that define certain attributes of the nonterminals in the rule

3. Each rule has a (possibly empty) set of predicates to check for attribute consistency

Chapter 3

Let X_0 -> X_1 ... X_n be a rule.

Functions of the form $S(X_0) = f(A(X_1), ... A(X_n))$ define *synthesized attributes*

Functions of the form $I(X_j) = f(A(X_0), ... , A(X_n))$, for $i <= j <= n$, define *inherited attributes*

Initially, there are *intrinsic attributes* on the leaves

***Example:* expressions of the form `id + id`**
- **`id`'s can be either int_type or real_type**
- **types of the two `id`'s must be the same**
- **type of the expression must match it's expected type**

BNF:

```
<expr> -> <var> + <var>
<var> -> id
```

Attributes:

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

Chapter 3

Attribute Grammar:

1. 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**

2. Syntax rule: `<var> -> id`
Semantic rule:
`<var>`.**actual_type** ← **lookup (**`id, <var>`**)**

How are attribute values computed?

1. **If all attributes were inherited, the tree could be decorated in top-down order.**
2. **If all attributes were synthesized, the tree could be decorated in bottom-up order.**
3. **In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.**

Chapter 3

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

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

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

Dynamic Semantics

- No single widely acceptable notation or formalism for describing semantics

I. *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

Chapter 3

- To use operational semantics for a high-level language, a virtual machine in needed

- A *hardware* pure interpreter would be too expensive

- A *software* pure interpreter also has problems:

  1. The detailed characteristics of the particular computer would make actions difficult to understand
  2. Such a semantic definition would be machine-dependent

- *A better alternative*: A complete computer simulation

- *The process:*
  1. Build a translator (translates source code to the machine code of an idealized computer)
  2. Build a simulator for the idealized computer

- *Evaluation of operational semantics:*
  - Good if used informally
  - Extremely complex if used formally (e.g., VDL)

Chapter 3

Axiomatic Semantics

- Based on formal logic (first order predicate calculus)
- *Original purpose:* formal program verification
- *Approach:* Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions)

- The expressions are called *assertions*

- 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

- Pre-post form: {P} statement {Q}

- *An example:* a := b + 1 {a > 1}
  One possible precondition: {b > 10}
  Weakest precondition: {b > 0}

Chapter 3

Program proof process: **The postcondition for the whole program is the desired results. Work back through the program to the first statement. If the precondition on the first statement is the same as the program spec, the program is correct.**

- An axiom for assignment statements:

$\{Q_{x->E}\}$ **x := E {Q}**

- The Rule of Consequence:

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

- An inference rule for sequences

- For a sequence S1;S2:

{P1} S1 {P2}
{P2} S2 {P3}

the inference rule is:

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

Chapter 3

- An inference rule for logical pretest loops

For the loop construct:

{P} while B do S end {Q}

the inference rule is:

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

where I is the loop invariant.

Characteristics of the loop invariant

I must meet the following conditions:

1. **P => I (the loop invariant must be true initially)**
2. **{I} B {I} (evaluation of the Boolean must not change the validity of I)**
3. **{I and B} S {I} (I is not changed by executing the body of the loop)**
4. **(I and (not B)) => Q (if I is true and B is false, Q is implied)**
5. **The loop terminates (this can be difficult to prove)**

Chapter 3

- 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:*

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

  2. 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

Denotational Semantics

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

Chapter 3

- The process of building a denotational spec for a language:
  1. Define a mathematical object for each language entity
  2. 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

- The difference between denotational and operational semantics: In operational semantics, the state changes are defined by coded algorithms; in denotational semantics, they are defined by rigorous mathematical functions

- 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

  $VARMAP(i_j, s) = v_j$

Chapter 3

1. 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

Chapter 3

2. Expressions

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)

Chapter 3

3 Assignment Statements

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

4 Logical Pretest Loops

$M_l(\text{while B do L}, s) \triangle =$
  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 $M_l(\text{while B do L}, M_{sl}(L, s))$

Chapter 3

- 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