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)

Formal approaches to describing syntax:

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

Copyright © 1995 by Addison-Wesley Publishing Co.

Chapter 3

Chomsky's hierarchy (late 1950s)

1. Recursively enumerable
2. Context-sensitive
3. Context-free
4. Regular

Backus Normal Form (1959) - John Backus

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

A *metalanguage* is a language used to describe another language.

Abstractions are used to represent classes of syntactic structures--they act like syntactic variables

e.g.

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

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

Copyright © 1995 by Addison-Wesley Publishing Co.

Chapter 3

Rules consist of *terminal* and *nonterminal* symbols

An abstraction can have more than one alternative RHSs

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

Syntactic lists are described in BNF using recursion

```
<ident_list> -> ident
                | ident, <ident_list>
```

A *grammar* is a finite nonempty set of rules

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

Copyright © 1995 by Addison-Wesley Publishing Co.

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*

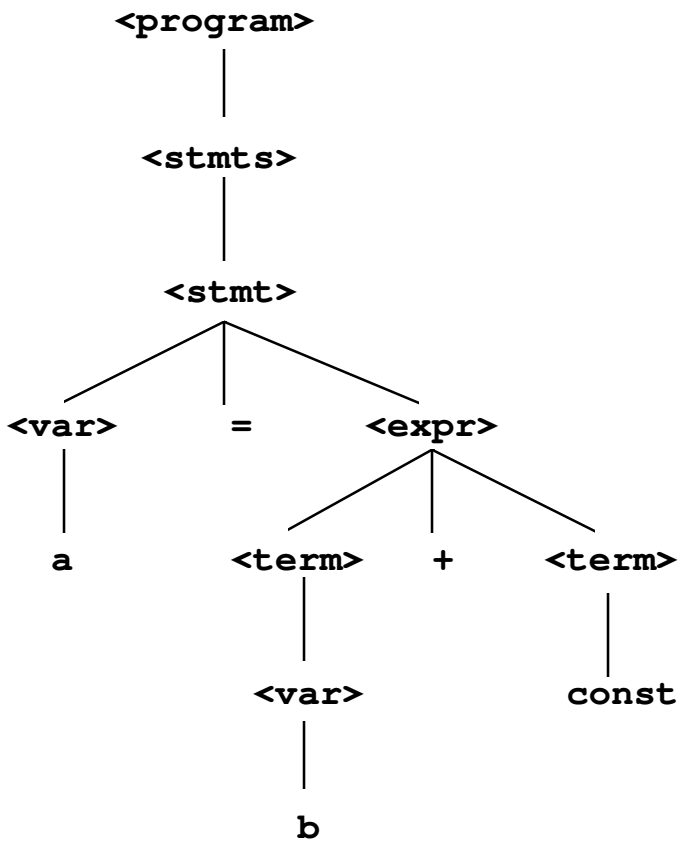
A *sentence* is a sentential form that has only terminal symbols

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

Copyright © 1995 by Addison-Wesley Publishing Co.

Chapter 3

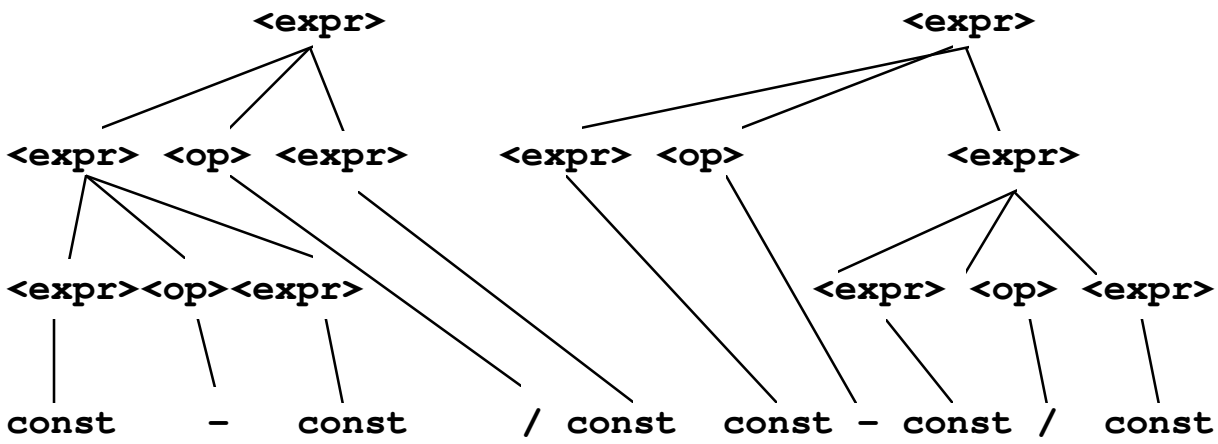


A grammar is *ambiguous* iff it generates a sentential form that has two or more distinct parse trees

Copyright © 1995 by Addison-Wesley Publishing Co.

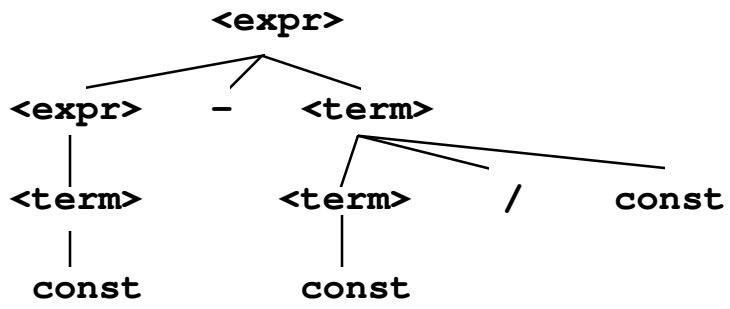
Chapter 3

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



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
```



No other parse tree is possible!

Copyright © 1995 by Addison-Wesley Publishing Co.

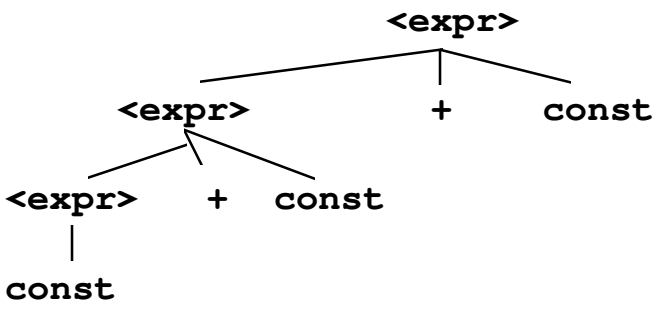
Chapter 3

```
<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:

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}`

Copyright © 1995 by Addison-Wesley Publishing Co.

Chapter 3

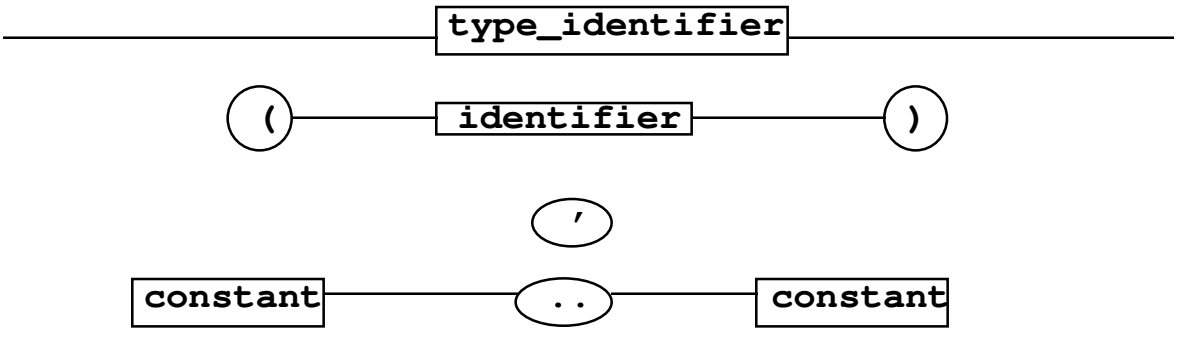
BNF:

```
<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**



Copyright © 1995 by Addison-Wesley Publishing Co.

Chapter 3

Static semantics **- have nothing to do with meaning.**
Kinds:
1. **Context-free (e.g. type checking)**
2. **Noncontext-free (e.g. variables must be declared before they are used)**

Attribute Grammars **(AGs) (Knuth, 1968)**

- cfgs cannot even 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 checking**
2. **Compiler design**

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**

Copyright © 1995 by Addison-Wesley Publishing Co.

Chapter 3

Let
X0 -> X1 ... Xn
be a rule.

Functions of the form S(X0) = f(A(X1), ... A(Xn)) define *synthesized attributes*

Functions of the form I(Xj) = f(A(X0), ... , A(Xn)), 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>`
`env` **- inherited for** `<expr>` **and** `<var>`

Copyright © 1995 by Addison-Wesley Publishing Co.

Chapter 3

Attribute Grammar:

1. syntax rule: `<expr> -> <var>[1] + <var>[2]`
   semantic rules:

```
<var>[1].env <- <expr>.env
<var>[2].env <- <expr>.env
<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>.env)
```

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, at it is some combination of top-down and bottom-up that must be done.

Copyright © 1995 by Addison-Wesley Publishing Co.

Chapter 3

1. `<expr>.env <- inherited from parent`
   `<expr>.expected_type <- inherited from parent`

2. `<var>[1].env <- <expr>.env`
   `<var>[2].env <- <expr>.env`

3. `<var>[1].actual_type <- lookup (A, <var>[1].env)`
   `<var>[2].actual_type <- lookup (B, <var>[2].env)`
   `<var>[1].actual_type =? <var>[2].actual_type`

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

Semantics

- No single widely acceptable notation or formalism for describing semantics

I. *Operational Semantics*
- describe the meaning of a statement by executing it on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement

Copyright © 1995 by Addison-Wesley Publishing Co.

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. 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 (VDL)

Copyright © 1995 by Addison-Wesley Publishing Co.

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 Boolean 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*

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

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

Copyright © 1995 by Addison-Wesley Publishing Co.

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

- An inference rule for sequences

- An inference rule for logical pretest loops

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 (notB)) => Q (if I is true and B is false, Q is implied)
5. The loop terminates (this can be difficult to prove)

Copyright © 1995 by Addison-Wesley Publishing Co.

- 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, 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 method

- Originally developed by Scott and Strachey (1970)

Copyright © 1995 by Addison-Wesley Publishing Co.

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 that language entity onto instances of the mathematical object

- 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

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

Copyright © 1995 by Addison-Wesley Publishing Co.

Chapter 3

1. Expressions

M_e (E, s) = if VARMAP(i, s) = undef for some i in E
then error
else E', where E' is the result of
evaluating E after setting each
variable i in E to VARMAP(i, s)

2. Assignment Statements

M_a(x := E, s) =
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

3. Logical Pretest Loops

M_l(while B do L, s) =
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))

Copyright © 1995 by Addison-Wesley Publishing Co.

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

Copyright © 1995 by Addison-Wesley Publishing Co.