Chapter 3

Describing Syntax and Semantics

Introduction

We usually break down the problem of defining a programming language into two parts.

- Defining the PL's syntax
- Defining the PL's semantics

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

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

Note: There is not always a clear boundary between the two.

Why and How

Why? We want specifications for several communities:

- Other language designers
- Implementors
- Programmers (the users of the language)

How? One ways is via natural language descriptions (e.g., user's manuals, text books) but there are a number of techniques for specifying the syntax and semantics that are more formal.

Syntax Overview

- **Language preliminaries**
- **Context-free grammars and BNF**
- **Syntax diagrams**

Introduction

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).

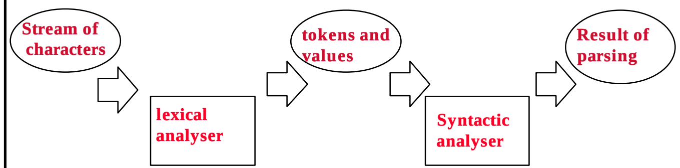
A *token* is a category of lexemes (e.g., identifier).

Formal approaches to describing syntax:

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

Lexical Structure of Programming Languages

- The structure of its lexemes (words or tokens)
  - token is a category of lexeme
- The scanning phase (lexical analyser) collects characters into tokens
- Parsing phase(syntactic analyser)determines syntactic structure



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 Normal/Naur Form (1959)

- Invented by John Backus to describe Algol 58 and refined by Peter Naur for Algol 60.
- BNF is equivalent to context-free grammars

BNF (continued)

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

BNF

- A rule has a left-hand side (LHS) which is a single non-terminal symbol and a right-hand side (RHS), one or more *terminal* or *nonterminal* symbols.
- A *grammar* is a finite nonempty set of rules
- A non-terminal symbol is “defined” by one or more rules.
- Multiple rules can be combined with the | symbol so that

```
<stmts> ::= <stmt>
<stmts> ::= <stmnt> ; <stmnts>
```

And this rule are equivalent

```
<stmts> ::= <stmt> | <stmnt> ; <stmnts>
```


BNF

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)

BNF Example

Here is an example of a simple grammar for a subset of English.

A sentence is noun phrase and verb phrase followed by a period.

```
<sentence>    ::= <noun-phrase><verb-phrase>.
<noun-phrase> ::= <article><noun>
<article>     ::= a | the
<noun>        ::= man | apple | worm | penguin
<verb-phrase> ::= <verb> | <verb><noun-phrase>
<verb>        ::= eats | throws | sees | is
```


Derivation using BNF

<sentence> -> <noun-phrase><verb-phrase>.
<article><noun><verb_phrase>.
the<noun><verb_phrase>.
the man <verb_phrase>.
the man <verb><noun-phrase>.
the man eats <noun-phrase>.
the man eats <article> < noun>.
the man eats the <noun>.
the man eats the apple.

Another BNF Example

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

Note: There is some variation in notation for BNF grammars. Here we are using -> in the rules instead of `::=`.

Here is a 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.*

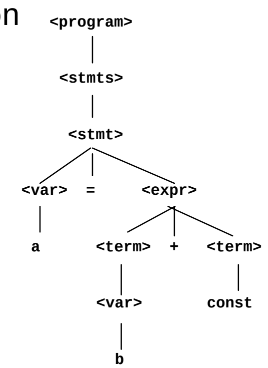
A *sentence* is a sentential form that has only terminal symbols.

A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded.

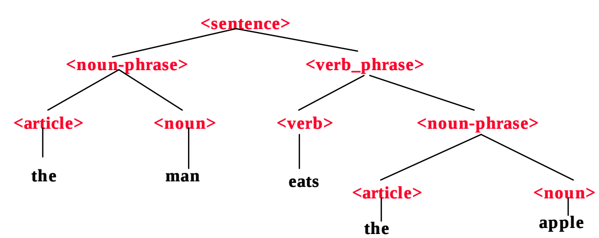
A derivation may be neither leftmost nor rightmost (or something else)

Parse Tree

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



Another Parse Tree



Grammar

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

Ambiguous grammars are, in general, very undesirable in formal languages.

We can eliminate ambiguity by revising the grammar.

Grammar

Here is a simple grammar for expressions that is ambiguous

```
<expr> -> <expr> <op> <expr>
<expr> -> int
<op> -> +|-|*|/
```

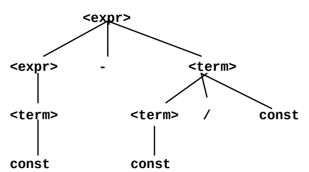
The sentence *1+2*3* can lead to two different parse trees corresponding to *1+(2*3)* and *(1+2)*3*

Grammar

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
```



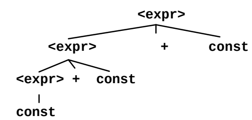
Grammar (continued)

```
<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)
```



An Expression Grammar

Here's a grammar to define simple arithmetic expressions over variables and numbers.

Exp ::= num
Exp ::= id
Exp ::= UnOp Exp
Exp := Exp BinOp Exp
Exp ::= '(' Exp ')'

UnOp ::= '+'
UnOp ::= '-'
BinOp ::= '+' | '-' | '*' | '/'

Here's another common notation variant where single quotes are used to indicate terminal symbols and unquoted symbols are taken as non-terminals.

A derivation

Here's a derivation of a+b*2 using the expression grammar:

```
Exp =>                   // Exp ::= Exp BinOp Exp
 Exp BinOp Exp =>        // Exp ::= id
 id BinOp Exp =>         // BinOp ::= '+'
 id + Exp =>             // Exp ::= Exp BinOp Exp
 id + Exp BinOp Exp =>   // Exp ::= num
 id + Exp BinOp num =>   // Exp ::= id
 id + id BinOp num =>    // BinOp ::= '*'
 id + id * num
 a  + b  * 2
```


A parse tree

A parse tree for a+b*2:

```
              __Exp__
             /   |   \
          Exp  BinOp   Exp
           |     |   /  |    \
     identifier + Exp BinOp Exp
                   |     |   |
              identifier * number
```


Precedence

Precedence refers to the order in which operations are evaluated. The convention is: exponents, mult div, add sub.

- Deal with operations in categories: exponents, mulops, addops.

Here's a revised grammar that follows these conventions:

```
Exp ::= Exp AddOp Exp
Exp ::= Term
Term ::= Term MulOp Term
Term ::= Factor
Factor ::= '(' + Exp + ')‘
Factor ::= num | id
AddOp ::= '+' | '-’
MulOp ::= '*' | '/'
```

Associativity

Associativity refers to the order in which 2 of the same operation should be computed

- 3+4+5 = (3+4)+5, left associative (all BinOps)
- 3^4^5 = 3^(4^5), right associative
- 'if x then if x then y else y' = 'if x then (if x then y else y)', else associates with closest unmatched if (matched if has an else)

Adding associativity to the BinOp expression grammar

```
Exp    ::= Exp AddOp Term
Exp    ::= Term
Term   ::= Term MulOp Factor
Term   ::= Factor
Factor ::= '(' Exp ')'
Factor ::= num | id
AddOp  ::= '+' | '-'
MulOp  ::= '*' | '/'
```


Another example: conditionals

- Goal: to create a correct grammar for conditionals.
- It needs to be non-ambiguous and the precedence is else with nearest unmatched if.

```
Statement    ::= Conditional | 'whatever'
Conditional ::= 'if' test 'then' Statement 'else' Statement
Conditional ::= 'if' test 'then' Statement
```

- The grammar is ambiguous. The 1st Conditional allows unmatched 'if's to be Conditionals.

  if test then (if test then whatever else whatever) = correct

  if test then (if test then whatever) else whatever = incorrect
- The final unambiguous grammar.

```
Statement ::= Matched | Unmatched
Matched ::= 'if' test 'then' Matched 'else' Matched | 'whatever'
Unmatched ::= 'if' test 'then' Statement
             | 'if' test 'then' Matched else Unmatched
```


Extended BNF

Syntactic sugar: doesn't extend the expressive power of the formalism, but does make it easier to use.

Optional parts are placed in brackets ([])

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

Put alternative parts of RHSs in parentheses and separate them with vertical bars

<term> -> <term> (+ | -) const

Put repetitions (0 or more) in braces ({})

<ident> -> letter {letter | digit}

BNF

BNF:

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

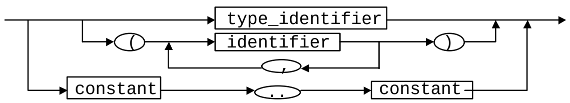
EBNF:

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

Syntax Graphs

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



Parsing

- A grammar describes the strings of tokens that are syntactically legal in a PL
- A *recogniser* simply accepts or rejects strings.
- A *parser* construct a derivation or parse tree.
- Two common types of parsers:
  - bottom-up or data driven
  - top-down or hypothesis driven
- A *recursive descent parser* traces is a way to implement a top-down parser that is particularly simple.

Recursive Decent Parsing

- 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 (why not?)

Recursive Decent Parsing Example

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

Semantics

Semantics Overview

- Syntax is about “form” and semantics about “meaning”.
- The boundary between syntax and semantics is not always clear.
- First we’ll look at issues close to the syntax end, what Sebesta calls “static semantics”, and the technique of attribute grammars.
- Then we’ll sketch three approaches to defining “deeper” semantics
  - Operational semantics
  - Axiomatic semantics
  - Denotational semantics

Static Semantics

Static semantics covers some language features that are difficult or impossible to handle in a BNF/CFG.

It is also a mechanism for building a parser which produces a “abstract syntax tree” of it’s input.

Categories attribute grammars can handle:

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

Attribute Grammars

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:

- Static semantics specification
- Compiler design (static semantics checking)

Attribute Grammar Example

In Ada we have the following rule to describe prodecure definitions:

<proc> -> procedure <procName> <procBody> end <procName> ;

But, of course, the name after "procedure" has to be the same as the name after "end".

This is not possible to capture in a CFG (in practice) because there are too many names.

Solution: associate simple attributes with nodes in the parse tree and add a "semantic" rules or constraints to the syntactic rule in the grammar.

<proc> -> procedure <procName>[1] <procBody> end <procName>[2] ;

<procName][1].string = <procName>[2].string

Attribute Grammars

Def: An *attribute grammar* is a CFG 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

Let X_0 -> X_1 ... 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

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>`

Attribute Grammars

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>)

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

<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.

The general approach to defining the semantics of any language L is to specify a general mechanism to translate any sentence in L into a set of sentences in another language or system that we take to be well defined.

Here are three approaches we'll briefly look at:
- Operational semantics
- Axiomatic semantics
- Denotational semantics

Operational Semantics

- Idea: describe the meaning of a program in language L by specifying how statements effect the state of a machine, (simulated or actual) when executed.
- The change in the state of the machine (memory, registers, stack, heap, etc.) defines the meaning of the statement.
- Similar in spirit to the notion of a *Turing Machine* and also used informally to explain higher-level constructs in terms of simpler ones, as in:

c statement	operational semantics

```
for(e1;e2;e3)              e1;
  {<body>}          loop:  if e2=0 goto exit
                           <body>
                           e3;
                           goto loop
                    exit:
```


Operational Semantics

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

Operational Semantics

A better alternative: A complete computer simulation

- 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
- Extremely complex if used formally (e.g. VDL)

Vienna Definition Language

- VDL was a language developed at IBM Vienna Labs as a language for formal, algebraic definition via operational semantics.
- It was used to specify the semantics of PL/I.
- See: *The Vienna Definition Language*, P. Wegner, ACM Comp Surveys 4(1):5-63 (Mar 1972)
- The VDL specification of PL/I was very large, very complicated, a remarkable technical accomplishment, and of little practical use.

Axiomatic Semantics

- Based on formal logic (first order predicate calculus)
- *Original purpose:* formal program verification
- *Approach:* Define axioms and inference rules in logic for each statement type in the language (to allow transformations of expressions to other expressions)
- The expressions are called *assertions* and are either
  - **Preconditions:** An assertion before a statement states the relationships and constraints among variables that are true at that point in execution
  - **Postconditions:** An assertion following a statement

Logic 101

Propositional logic:

Logical constants: true, false

Propositional symbols: P, Q, S, ... that are either true or false

Logical connectives: $\wedge$ (and) , $\vee$ (or), $\Rightarrow$ (implies), $\Leftrightarrow$ (is equivalent), $\neg$ (not) which are defined by the truth tables below.

Sentences are formed by combining propositional symbols, connectives and parentheses and are either true or false. e.g.: $P \wedge Q \Leftrightarrow \neg (\neg P \vee \neg Q)$

First order logic adds

Variables which can range over objects in the domain of discourse

Quantifiers including: $\forall$ (forall) and $\exists$ (there exists)

Example sentences:

$(\forall p)\ (\forall q)\ p \wedge q \Leftrightarrow \neg (\neg p \vee \neg q)$

$\forall x\ \text{prime}(x) \Rightarrow \exists y\ \text{prime}(y) \wedge y > x$

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
False	*False*	*True*	*False*	*False*	*True*	*True*
False	*True*	*True*	*False*	*True*	*True*	*False*
True	*False*	*False*	*False*	*True*	*False*	*False*
True	*True*	*False*	*True*	*True*	*True*	*True*

Axiomatic Semantics

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

Notation:

{P} Statement {Q}

precondition **postcondition**

Example:

{?} a := b + 1 {a > 1}

We often need to infer what the precondition must be for a given postcondition

One possible precondition: {b > 10}

Weakest precondition: {b > 0}

Axiomatic Semantics

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.

Example: Assignment Statements

Here's how we might define a simple assignment statement of the form ***x := e*** in a programming language.

- $\{Q_{x->E}\}$ x := E {Q}
- Where $Q_{x->E}$ means the result of replacing all occurrences of *x* with *E* in *Q*

So from

{Q} a := b/2-1 {a<10}

We can infer that the weakest precondition Q is

b/2-1<10 or b<22

Axiomatic Semantics

•*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:

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

A notation from symbolic logic for specifying a rule of inference with premise P and consequence Q is

$$\frac{P}{Q}$$

For example, Modus Ponens can be specified as:

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

Conditions

Here's a rule for a conditional statement

$$\frac{\{B \wedge P\}\ S1\ \{Q\},\ \{\neg B \wedge P\}\ S2\ \{Q\}}{\{P\}\ \text{if B then S1 else S2}\ \{Q\}}$$

And an example of it's use for the statement

{P} if x>0 then y=y-1 else y=y+1 {y>0}

So the weakest precondition P can be deduced as follows:

The postcondition of S1 and S2 is Q.

The weakest precondition of S1 is *x>0 Ù y>1* and for S2 is *x>0 Ù y>-1*

The rule of consequence and the fact that *y>1 Þ y>-1* supports the conclusion

That the weakest precondition for the entire conditional is *y>1* .

Loops

For the loop construct {P} while B do S end {Q} the inference rule is:

$$\frac{\{I \wedge B\}\ \ S\ \ \{I\}}{\{I\}\ \text{while B do S}\ \{I \wedge \neg B\}}$$

where I is the *loop invariant*, a proposition necessarily true throughout the loop's execution.

Loop Invariants

A 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)

- 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
- It is much less useful for language users and compiler writers

Denotational Semantics

- A technique for describing the meaning of programs in terms of mathematical functions on programs and program components.
- Programs are translated into functions about which properties can be proved using the standard mathematical theory of functions, and especially domain theory.
- Originally developed by Scott and Strachey (1970) and based on recursive function theory
- The most abstract semantics description method

Denotational Semantics

- The process of building a denotational specification 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

Denotational Semantics (continued)

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

  $s = \{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}$

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

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

Example: 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

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

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

Logical Pretest Loops

M_l(while B do L, s) Δ=

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))

Logical Pretest Loops

- 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

Denotational Semantics

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

Summary

This chapter covered the following

- Backus-Naur Form and Context Free Grammars
- Syntax Graphs and Attribute Grammars
- Semantic Descriptions: Operational, Axiomatic and Denotational