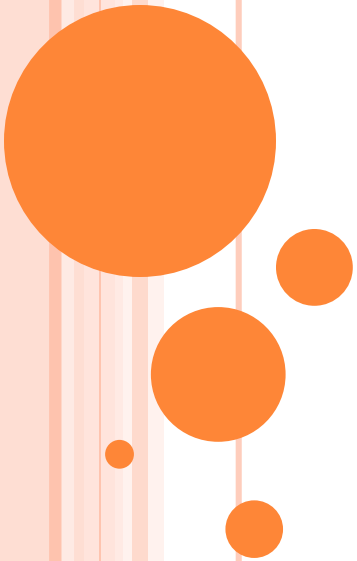


Chapter 2(2)

Describing Syntax and Semantics

Dynamic Semantics

- Describing syntax is relatively simple
- There is no single widely acceptable notation or formalism for describing semantics
- If there were a precise semantics specification of a programming language, programs written in the language potentially could be proven correct without testing, correctness of compilers could be verified.

Approaches in (dynamic) 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
- Denotational semantics
  - Using mathematical and logic approach.
- Axiomatic Semantics
  - Specifies what can be proven about the program, rather than directly specifying the meaning of a program.

Operational Semantics

- To describe the meaning of a statement or program by specifying the effects of running it on a machine.
- Machine language shows sequence of changes in state, but problems to be used as formal description:
  - Too small and too numerous of individual steps in machine language
  - Too large and complex storage architecture
- Different levels of uses:
  - At highest level, use with interest in the final result (natural ops. Semantics).
  - At lowest level, to determine the precise meaning of program (structural ops. Semantics)

Operational Semantics (continued)

- The basic process:
  - Design appropriate intermediate language that is clear and unambiguous

Operational Semantics (continued)

- C statement

```
for ( expr1; expr2; expr3) {
        …;
        …;
}
```

Meaning :

```
        expr1;
loop :  if expr2 == 0 goto out
        …
        expr3;
        goto loop
out :  …
```

Operational Semantics (continued)

- 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.
  - Provides an effective means of describing semantics for language users and language implementers – as long as the descriptions are kept simple and informal.
  - Depends on programming language of lower level, not mathematics – the statement of one programming language are described in terms of the statements of a lower-level programming language.

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

Denotational Semantics versus Operational Semantics

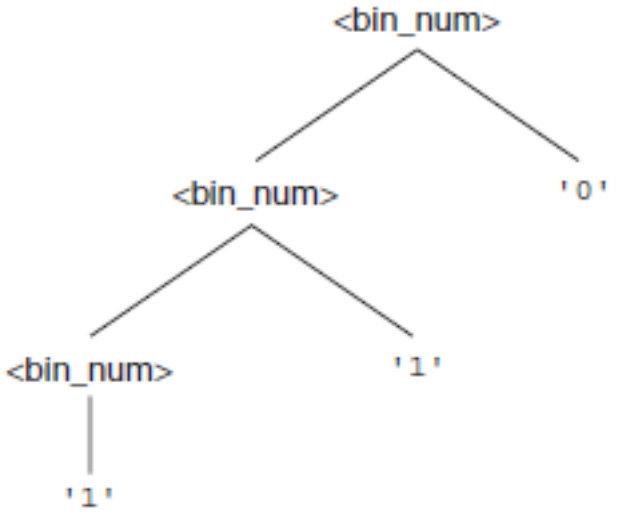
- In operational semantics, the state changes are defined by coded algorithms
- In denotational semantics, the state changes are defined by rigorous mathematical functions

Denotational semantics example

The following rule to describe the binary number

```
<bin_num> →   '0'
            | '1'
            | <bin_num> '0'
            | <bin_num> '1'
```

A parse tree for the example binary number, 110,

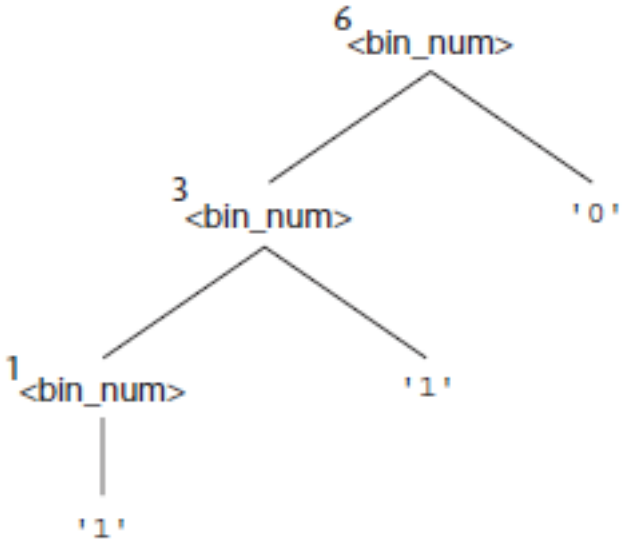


Denotational semantics example

The following rule to describe the binary number of a decimal number

```
Mbin('0') = 0
Mbin('1') = 1
Mbin(<bin_num> '0') = 2 * Mbin(<bin_num>)
Mbin(<bin_num> '1') = 2 * Mbin(<bin_num>) + 1
```

Example the decimal number 6



Denotational semantics example

<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

Denotational Semantics: Program State

- 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

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

EXPRESSIONS

- Map expressions onto Z $\cup$ {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

EXPRESSIONS

- Describe the semantics of the following BNF:

<expr> → <dec_num> | <var> | <binary_expr>

<binary_expr> → <left_expr> <operator> <right_expr>

<left_expr> → <dec_num> | <var>

<right_expr> → <dec_num> | <var>

<operator> → + | *

SEMANTICS (CONT.)

```
Me(<expr>, s) Δ=
    case <expr> of
      <dec_num> => Mdec(<dec_num>, s)
      <var> =>
           if VARMAP(<var>, s) == undef
                then error
                else VARMAP(<var>, s)
     <binary_expr> =>
          if (Me(<binary_expr>.<left_expr>, s) == undef
                OR Me(<binary_expr>.<right_expr>, s) =
                                 undef)
              then error
        else
        if (<binary_expr>.<operator> == '+' then
           Me(<binary_expr>.<left_expr>, s) +
                  Me(<binary_expr>.<right_expr>, s)
        else Me(<binary_expr>.<left_expr>, s) *
             Me(<binary_expr>.<right_expr>, s)
     ...
```

Assignment Statements

- Maps state sets to state sets

$Ma(x := E, s)\ \Delta =$

if $Me(E, s) == error$

then error

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

where for $j = 1, 2, \ldots, n$,

if $i_j == x$

then $v_j = Me(E, s)$

else $v_j = VARMAP(i_j, s)$

Logical Pretest Loops

- Maps state sets to state sets

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))

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 mathematical logic
- Original purpose: formal program verification – to prove the correctness of programs.
- The meaning of a program is based on relationships among program variables and constants, which are the same for every execution of the program.
- Two distinct applications:
  - Program verification
  - Semantics specification

Axiomatic Semantics (continued)

- 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*
- 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* that describes the new constraints on those variables after execution of the statement
- 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.

Proving assignment statements

- The usual notation is

  {P} S {Q}

  E.g.

  a = b / 2 – 1 {a < 10}

  The weakest precondition is computed by substituting b / 2 – 1 for a in the postcondition {a < 10}.

  b / 2 – 1 < 10

  b < 22

- Therefore, the weakest precondition is {b < 22}

Proving sequences

- An inference rule for sequences

  {P1} S1 {P2}

  {P2} S2 {P3}

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

– antecedent

- consequent

If antecedent are true, then truth of consequent can be inferred

PROVING SEQUENCES

- For example, consider the following sequence and postcondition:

  y = 3 * x + 1;

  x = y + 3;

  {x < 10}

- The precondition for the second assignment statement is

  y < 7

- which is used as the postcondition for the first statement. The precondition for

  the first assignment statement can now be computed:

  3 * x + 1 < 7

  x < 2

So, {x < 2} is the precondition of both the first statement and the twostatement

sequence.

PROVING SELECTION

- An inference rule for selection

  if B then S1 else s2

$$\frac{\{B \text{ and } P\}\, S1\, \{Q\},\ \{(\text{not } B) \text{ and } P\}\, S2\, \{Q\}}{\{P\} \text{ if } B \text{ then } S1 \text{ else } S2\, \{Q\}}$$

– antecedent

- consequent

The selection statement must be proven for both when the boolean expression is true and false.

We need a precondition P that can be used in the precondition of both the **then** and **else** clause.

Proving selection (continued)

Example

```
if x >0 then
        y= y - 1
else
        y= y + 1
```

Suppose Q is {y > 0}, using rule of consequence, the precondition for the whole statement is {y > 1}.

Proving Loop

- An inference rule for logical pretest loops

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

- The axiomatic description of a while loop is written as

  {P} while B do S end {Q}

  where I is the loop invariant (the inductive hypothesis)

Proving Loop

- E.g.

```
while y <> x do y = y + 1
```

- Zero iteration, wp is obviously {y = x}
- One iteration, wp is

  y = y + 1, {y = x}) = {y + 1 = x}, or {y = x - 1}
- Two iterations? Three iterations?

- Therefore, {y <= x} is the precondition {P}

Proving Loop

- Characteristics of the loop invariant: I must meet the following conditions:
  - P => I -- the loop invariant must be true initially
  - {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

Proving Loop

- P => I
  - In this case, I is used as precondition. Because P = I, P => I.
- {I and B} S {I}
  - `{y <= x and y <> x} y = y + 1 {y <= x}`
  - Applying the axiom to y = y + 1 `{y <= x}` we get `{y + 1 <= x}`, which is equivalent to `{y < x}`, which is implied by `{y <= x and y <> x}`
  - So, proven
- (I and (not B)) => Q
  - `{(y <= x and (not (y <> x)))} => {y = x}`
  - `{(y <=x and (y = x)) => {y = x}`
  - `{y = x} => {y = x}`
- The loop terminates
  - `{y <= x} while y <> x do y = y + 1 end {y = x}`

Proving Loop

- 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 (others:
    - Domain theory and fixed point, Algebraic, Action & Translational Semantics