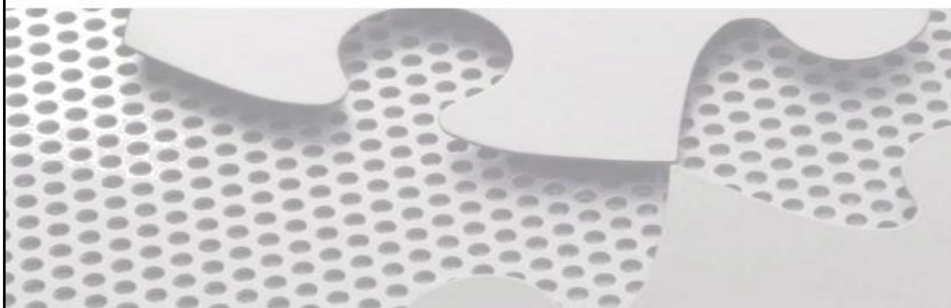
Programming Languages Third Edition



Chapter 12

Formal Semantics

Objectives

- Become familiar with a sample small language for the purpose of semantic specification
- Understand operational semantics
- Understand denotational semantics
- Understand axiomatic semantics
- Become familiar with proofs of program correctness

Introduction

- In previous chapters, we discussed semantics from an informal, or descriptive, point of view
  - Historically, this has been the usual approach
- There is a need for a more mathematical description of the behavior of programs and programming languages, to make the definition of a language so precise that:
  - Programs can be **proven** correct in a mathematical way
  - Translators can be **validated** to produce exactly the behavior described in the language definition

Introduction (cont'd.)

- Developing such a mathematical system aids the designer in discovering inconsistencies and ambiguities
- There is no single accepted method for formally defining semantics
- Several methods differ in the formalisms used and the kinds of intended applications
- Formal semantic descriptions are more often supplied after the fact, and only for a portion of a language

Introduction (cont'd.)

- Formal methods have begun to be used as part of the specification of complex software projects, including language translators
- Three principal methods to describe semantics formally:
  - Operational semantics
  - Denotational semantics
  - Axiomatic semantics

Introduction (cont'd.)

- **Operational semantics**:
  - Defines a language by describing its actions in terms of the operators of an actual or hypothetical machine
  - Requires that the operations of the machine used in the description are also precisely defined
  - A mathematical model called a "reduction machine" is often used for this purpose (similar in spirit to the notion of a Turing machine)

Introduction (cont'd.)

- **Denotational semantics**:
  - Uses mathematical functions on programs and program components to specify semantics
  - Programs are translated into functions about which properties can be proved using standard mathematical theory of functions

Introduction (cont'd.)

- **Axiomatic semantics**:
  - Applies mathematical logic to language definition
  - Assertions, or predicates, are used to describe desired outcomes and initial assumptions for program
  - Language constructs are associated with **predicate transforms** to create new assertions out of old ones
  - Transformers can be used to prove that the desired outcome follows from the initial conditions
  - Is a method aimed specifically at correctness proofs

Introduction (cont'd.)

- All these methods are syntax-directed
  - Semantic definitions are based on a context-free grammar or Backus-Naur Form (BNF) rules
- Formal semantics must then define all properties of a language that are not specified by the BNF
  - Includes static properties such as static types and declaration before use
- Formal methods can describe both static and dynamic properties
- We will view semantics as everything not specified by the BNF

Introduction (cont'd.)

- Two properties of a specification are essential:
  - Must be **complete**: every correct, terminating program must have associated semantics given by the rules
  - Must be **consistent**: the same program cannot be given two different, conflicting semantics
- Additionally, it is advantageous for the semantics to be minimal, or **independent**
  - No rule is derivable from the other rules

Introduction (cont'd.)

- Formal specifications written in operational or denotational style have an additional useful property:
  - They can be translated relatively easily into working programs in a language suitable for prototyping, such as Prolog, ML, or Haskell

A Sample Small Language

- The basic sample language to be used is a version of the integer expression language used in Ch. 6
- BNF rules for this language:

expr → *expr* '+' *term* | *expr* '−' *term* | *term*
term → *term* '*' *factor* | *factor*
factor → '(' *expr* ')' | *number*
number → *number digit* | *digit*
digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Figure 12.1 Basic sample language

A Sample Small Language (cont'd.)

- This results in simple semantics:
  - The value of an expression is a complete representation of its meaning: 2 + 3 * 4 means 14
- Complexity will now be added to this language in stages
- In the first stage, we add variables, statements, and assignments
  - A program is a list of statements separated by semicolons
  - A statement is an assignment of an expression to an identifier

A Sample Small Language (cont'd.)

factor → '(' *expr* ')' | *number* | *identifier*
program → *stmt-list*
stmt-list → *stmt* ';' *stmt-list* | *stmt*
stmt → *identifier* ':=' *expr*
identifier → *identifier letter* | *letter*
letter → 'a' | 'b' | 'c' | . . . | 'z'

Figure 12.2 First extension of the sample language

A Sample Small Language (cont'd.)

- Semantics are now represented by a set of values corresponding to identifiers whose values have been defined, or bound, by assignments
- Example:

```
a := 2+3;
b := a*4;
a := b-5
```

  – Results in bindings `b=20` and `a=15` when it finishes
  – Set of values representing the semantics of the program is `{a=15, b=20}`

A Sample Small Language (cont'd.)

- Such a set is essentially a function from identifiers to integer values, with all unassigned identifiers having a value undefined
  – This function is called an **environment**, denoted by:
  Env: Identifier $\rightarrow$ Integer $\cup$ {undef}
- Note that the `Env` function given by this example program can be defined as:

$$Env(I) = \begin{cases} 15 \text{ if } I = a \\ 20 \text{ if } I = b \\ \text{undef otherwise} \end{cases}$$

A Sample Small Language (cont'd.)

- The operation of looking up the value of an identifier `I` in an environment `Env` is `Env(I)`
- **Empty environment** is denoted by $\texttt{Env}_0$

$$Env_0(I) = \text{undef for all } I$$

- An environment as defined here incorporates both the symbol table and state functions
- Such environments:
  - Do not allow pointer values
  - Do not include scope information
  - Do not permit aliases

A Sample Small Language (cont'd.)

- For this view of the semantics of a program represented by a resulting final environment:
  - Consistency: we cannot derive two different final environments for the same program
  - Completeness: we must be able to derive a final environment for every correct, terminating program
- We now add `if` and `while` control statements
  - Syntax of the `if` and `while` statements borrows the Algol68 convention of writing reserved words backward, instead of `begin` and `end` blocks

A Sample Small Language (cont'd.)

stmt → *assign-stmt* | *if-stmt* | *while-stmt*
assign-stmt → *identifier* ':=' *expr*
if-stmt → 'if' *expr* 'then' *stmt-list* 'else' *stmt-list* 'fi'
while-stmt → 'while' *expr* 'do' *stmt-list* 'od'

Figure 12.3 Second extension of the sample language

A Sample Small Language (cont'd.)

- Meaning of an `if-stmt`:
  - `expr` is evaluated in the current environment
  - If it evaluates to an integer greater than 0, then `stmt-list` after `then` is executed
  - If not, `stmt-list` after the `else` is executed
- Meaning of a `while-stmt`:
  - As long as `expr` evaluates to a quantity greater than 0, `stmt-list` is repeatedly executed and `expr` is reevaluated
- Note that these semantics are nonstandard!

A Sample Small Language (cont'd.)

- Example program in this language:

```
n := 0 - 5;
if n then i := n else i := 0 - n fi;
fact := 1;
while i do
  fact := fact * i;
  i := i - 1
od
```

- Semantics are given by the final environment:

$\{n = -5, i = 0, \text{fact} = 120\}$

A Sample Small Language (cont'd.)

- Difficult to provide semantics for loop constructs
  - We will not always give a complete solution
- Formal semantic methods often use a simplified version of syntax from that given
- An ambiguous grammar can be used to define semantics because:
  - Parsing step is assumed to have already taken place
  - Semantics are defined only for syntactically correct constructs
- Nonterminal symbols can be replaced by single letters

A Sample Small Language (cont'd.)

- Nonterminal symbols can be replaced by single letters
  - May be thought to represent strings of tokens or nodes in a parse tree
- Such a syntactic specification is sometimes called an **abstract syntax**

A Sample Small Language (cont'd.)

- Abstract syntax for our sample language:

$$
\begin{aligned}
&P \to L \\
&L \to L_1 \text{ ';' } L_2 \mid S \\
&S \to I \text{ ':=' } E \mid \text{'if' } E \text{ 'then' } L_1 \text{, 'else' } L_2 \text{ 'fi'} \\
&\qquad \mid \text{'while' } E \text{ 'do' } L \text{ 'od'} \\
&E \to E_1 \text{ '+' } E_2 \mid E_1 \text{ '}-\text{' } E_2 \mid E_1 \text{ '*' } E_2 \\
&\qquad\qquad\qquad\qquad \mid \text{ '(' } E_1 \text{ ')' } \mid N \\
&N \to N_1\, D \mid D \\
&D \to \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'} \\
&I \to I_1\, A \mid A \\
&A \to \text{'}a\text{'} \mid \text{'}b\text{'} \mid \ldots \mid \text{'}z\text{'}
\end{aligned}
$$

P : Program
L : Statement-list
S : Statement
E : Expression
N : Number
D : Digit
I : Identifier
A : Letter

A Sample Small Language (cont'd.)

- To define the semantics of each symbol, we define the semantics of each right-hand side of the abstract syntax rules in terms of the semantics of their parts
  - Thus, syntax-directed semantic definitions are recursive in nature
- Tokens in the grammar are enclosed in quotation marks

Operational Semantics

- Operational semantics specify how an arbitrary program is to be executed on a machine whose operation is completely known
- **Definitional interpreters** or **compilers**: translators for the language written in the machine code of the chosen machine
- Operational semantics can define the behavior of programs in terms of an **abstract machine**

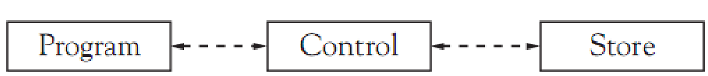


Figure 12-4 Three parts of an abstract machine

Operational Semantics (cont'd.)

- **Reduction machine**: an abstract machine whose control operates directly on a program to reduce it to its semantic "value"
- Example: reduction of the expression `(3+4)*5`

(3 + 4) * 5	=> (7) * 5	— 3 and 4 are added to get 7
	=> 7 * 5	— the parentheses around 7 are dropped
	=> 35	— 7 and 5 are multiplied to get 35

- To specify the operational semantics, we give **reduction rules** that specify how the control reduces constructs of the language to a value

Logical Inference Rules

- Inference rules in logic are written in the form:

$$\frac{\text{premise}}{\text{conclusion}}$$

  - If the premise is true, the conclusion is also true
- Inference rule for the commutative property of addition: $\frac{a + b = c}{b + a = c}$
- Inference rules are used to express the basic rules of prepositional and predicate calculus:

$$\frac{a \rightarrow b,\ b \rightarrow c}{a \rightarrow c}$$

Logical Inference Rules (cont'd.)

- **Axioms**: inference rules with no premise
  - They are always true
  - Example:

$$a + 0 = a$$

  - Axioms can be written as an inference rule with an empty premise:

$$\frac{}{a + 0 = a}$$

  - Or without the horizontal line:

$$a + 0 = a$$

Reduction Rules for Integer Arithmetic Expressions

- **Structured operational semantics**: the notational form for writing reduction rules that we will use
- Semantics rules are based on the abstract syntax for expressions:

$$E \rightarrow E_1 \text{ '+' } E_2 \mid E_1 \text{ '–' } E_2 \mid E_1 \text{ '*' } E_2 \mid \text{'(' } E_1 \text{ ')'}$$
$$N \rightarrow N_1\ D \mid D$$
$$D \rightarrow \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'}$$

- The notation $E => E_1$ states that expression `E` reduces to expression `E1` by some reduction rule

Reduction Rules for Expressions

1. Collect all rules for reducing digits to values in this one rule
   – All are axioms

‘0’ => 0
‘1’ => 1
‘2’ => 2
‘3’ => 3
‘4’ => 4
‘5’ => 5
‘6’ => 6
‘7’ => 7
‘8’ => 8
‘9’ => 9

Reduction Rules for Expressions (cont’d.)

2. Collect all rules for reducing numbers to values in this one rule
   – All are axioms

V ‘0’ => 10 * *V*
V ‘1’ => 10 * *V* + 1
V ‘2’ => 10 * *V* + 2
V ‘3’ => 10 * *V* + 3
V ‘4’ => 10 * *V* + 4
V ‘5’ => 10 * *V* + 5
V ‘6’ => 10 * *V* + 6
V ‘7’ => 10 * *V* + 7
V ‘8’ => 10 * *V* + 8
V ‘9’ => 10 * *V* + 9

Reduction Rules for Expressions (cont'd.)

3. V_1 '+' V_2 => $V_1 + V_2$
4. V_1 '−' V_2 => $V_1 - V_2$
5. V_1 '*' V_2 => $V_1 * V_2$
6. '(' V ')' => V
7. $$\frac{E \Rightarrow E_1}{E \text{ '+' } E_2 \Rightarrow E_1 \text{ '+' } E_2}$$
8. $$\frac{E \Rightarrow E_1}{E \text{ '−' } E_2 \Rightarrow E_1 \text{ '−' } E_2}$$
9. $$\frac{E \Rightarrow E_1}{E \text{ '*' } E_2 \Rightarrow E_1 \text{ '*' } E_2}$$
10. $$\frac{E \Rightarrow E_1}{V \text{ '+' } E \Rightarrow V \text{ '+' } E_1}$$
11. $$\frac{E \Rightarrow E_1}{E \text{ '−' } E \Rightarrow V \text{ '−' } E_1}$$
12. $$\frac{E \Rightarrow E_1}{V \text{ '*' } E \Rightarrow V \text{ '*' } E_1}$$
13. $$\frac{E \Rightarrow E_1}{\text{'(' } E \text{ ')'} \Rightarrow \text{'(' } E_1 \text{ ')'}}$$
14. $$\frac{E \Rightarrow E_1,\ E_1 \Rightarrow E_2}{E \Rightarrow E_2}$$

Reduction Rules for Expressions (cont'd.)

- Rules 1 through 6 are all axioms
- Rules 1 and 2 express the reduction of digits and numbers to values
  - **Character** '0' (a syntactic entity) reduces to the **value** 0 (a semantic entity)
- Rules 3 to 5 allow an expression consisting of two values and an operator symbol to be reduced to a value by applying the appropriate operation whose symbol appears in the expression
- Rule 6 says parentheses around an expression can be dropped

Reduction Rules for Expressions (cont'd.)

- The rest of the reduction rules are inferences that allow the reduction machine to combine separate reductions together to achieve further reductions
- Rule 14 expresses the general fact that reductions can be performed stepwise (sometimes called the **transitivity rule** for reductions)

Reduction Rules for Expressions (cont'd.)

- Applying these reduction rules to the expression:
  2 * (3 + 4) − 5
- First reduce the expression: `3 + 4`:

'3' '+' '4'	=> 3 '+' '4'	(Rules 1 and 7)
	=> 3 '+' 4	(Rules 1 and 10)
	=> 3 + 4 = 7	(Rule 3)

- Thus, by rule 14, we have: '3' '+' '4' => 7

Reduction Rules for Expressions (cont'd.)

- Continuing:

'(' '3' '+' '4' ')' => '(' 7 ')' (Rule 13)
=> 7 (Rule 6)

- Now reduce the expression `2*(3+4)` as follows:

'2' '*' '(' '3' '+' '4' ')' => 2 '*' '(' '3' '+' '4' ')' (Rules 1 and 9)
=> 2 '*' 7 (Rule 12)
=> 2 * 7 = 14 (Rule 5)

- And finally:

'2' '*' '(' '3' '+' '4' ')' '–' '5' => 14 '–' '5' (Rules 1 and 8)
=> 14 '–' 5 (Rule 11)
=> 14 – 5 = 9 (Rule 4)

Environments and Assignment

- Abstract syntax for our sample language:

$P \to L$
$L \to L_1$ ';' $L_2 \mid S$
$S \to I$ ':=' $E \mid$ 'if' E 'then' L_1 'else' L_2 'fi'
$\mid$ 'while' E 'do' L 'od'
$E \to E_1$ '+' $E_2 \mid E_1$ '–' $E_2 \mid E_1$ '*' E_2
$\mid$ '(' E_1 ')' $\mid N$
$N \to N_1 D \mid D$
$D \to$ '0' $\mid$ '1' $\mid \ldots \mid$ '9'
$I \to I_1 A \mid A$
$A \to$ 'a' $\mid$ 'b' $\mid \ldots \mid$ 'z'

P : Program
L : Statement-list
S : Statement
E : Expression
N : Number
D : Digit
I : Identifier
A : Letter

Environments and Assignment (cont'd.)

- We want to extend the operational semantics to include environments and assignments
- Must include the effect of assignments on the storage of the abstract machine
- Our view of storage: an environment that is a function from identifiers to integer values (including the undefined value):

  Env: Identifier $\rightarrow$ Integer $\cup$ {undef}
- The notation $<E \mid Env>$ indicates that expression `E` is evaluated in the presence of environment `Env`

Environments and Assignment (cont'd.)

- Now our reduction rules change to include environments
- Example: rule 7 with environments becomes:

$$\frac{<E \mid Env> => <E_1 \mid Env>}{<E \text{ ‘+’ } E_2 \mid Env> => <E_1 \text{ ‘+’ } E_2 \mid Env>}$$

  – This states that if `E` reduces to `E1` in the presence of `Env`, then `E '+' E2` reduces to `E1 '+' E2` in the same environment

Environments and Assignment (cont'd.)

- The one case of evaluation that explicitly involves the environment is when an expression is an identifier `I`, giving a new rule:

15. $$\frac{Env(I) = V}{<I \mid Env> \; => \; <V \mid Env>}$$

  This states that if the value of identifier `I` is `V` in `Env`, then `I` reduces to `V` in the presence of `Env`

- Next, we add assignment statements and statement sequences to the reduction rules

Environments and Assignment (cont'd.)

- Statements must reduce to environments instead of integer values, since they create and change environments, giving this rule:

16. $<I \text{ ':=' } V \mid Env> \; => \; Env \,\&\, \{I = V\}$

  This states that the assignment of the value `V` to `I` in `Env` reduces to a new environment where `I` is equal to `V`

- Reduction of expressions within assignments uses this rule:

17. $$\frac{<E \mid Env> \; => \; <E_1 \mid Env>}{<I \text{ ':=' } E \mid Env> \; => \; <I \text{ ':=' } E_1 \mid Env>}$$

Environments and Assignment (cont'd.)

- A statement sequence reduces to an environment formed by accumulating the effect of each assignment, giving this rule:

18. $$\frac{<S \mid Env> \Rightarrow Env_1}{<S \text{ ‘;’ } L \mid \mathrm{Env}> \Rightarrow <L \mid Env_1>}$$

- Finally, a program is a statement sequence with no prior environment, giving this rule:

19. $$L \Rightarrow <L \mid Env_0>$$

    It reduces to the effect it has on the empty starting environment

Environments and Assignment (cont'd.)

- Rules for reducing identifier expressions are completely analogous to those for reducing numbers
- Sample program to be reduced to an environment:

```
a := 2+3;
b := a*4;
a := b-5
```

- To simplify the reduction, we will suppress the use of quotes to differentiate between syntactic and semantic entities

Environments and Assignment (cont'd.)

- First, by rule 19, we have:

$a := 2 + 3;\ b := a * 4;\ a := b - 5 \Rightarrow$
$\langle a := 2 + 3;\ b := a * 4;\ a := b - 5 \mid Env_0 \rangle$

- Also, by rules 3, 17, and 16:

$\langle a := 2 + 3 \mid Env_0 \rangle \Rightarrow$
$\langle a := 5 \mid Env_0 \rangle \Rightarrow$
$Env_0 \ \&\ \{a = 5\} = \{a = 5\}$

- Then by rule 18:

$\langle a := 2 + 3;\ b := a * 4;\ a := b - 5 \mid Env_0 \rangle \Rightarrow$
$\langle b := a * 4;\ a := b - 5 \mid \{a = 5\} \rangle$

Environments and Assignment (cont'd.)

- Similarly, by rules 15, 9, 5, 17, and 16:

$\langle b := a * 4 \mid \{a = 5\} \rangle \Rightarrow \langle b := 5 * 4 \mid \{a = 5\} \rangle \Rightarrow$
$\langle b := 20 \mid \{a = 5\} \rangle \Rightarrow \{a = 5\}\ \&\ \{b = 20\} = \{a = 5, b = 20\}$

- Then by rule 18 :

$\langle b := a * 4;\ a := b - 5 \mid \{a = 5\} \rangle \Rightarrow$
$\langle a := b - 5 \mid \{a = 5, b = 20\} \rangle$

- Finally, by a similar application of rules:

$\langle a := b - 5 \mid \{a = 5, b = 20\} \rangle \Rightarrow$
$\langle a := 20 - 5 \mid \{a = 5, b = 20\} \rangle \Rightarrow$
$\langle a := 15 \mid \{a = 5, b = 20\} \rangle \Rightarrow$
$\{a = 5, b = 20\}\ \&\ \{a = 15, b = 20\}$

Control

- Next we add if and while statements, with this abstract syntax:

$$S \rightarrow \text{‘if’ } E \text{ ‘then’ } L_1 \text{ ‘else’ } L_2 \text{ ‘fi’}$$
$$\mid \text{ ‘while’ } E \text{ ‘do’ } L \text{ ‘od’}$$

- Reduction rules for if statements include:

20. $$\frac{<E \mid Env> \; => \; <E_1 \mid Env>}{<\text{‘if’ } E \text{ ‘then’ } L_1 \text{ ‘else’ } L_2 \text{ ‘fi’} \mid Env> \; => \; <\text{‘if’ } E_1 \text{ ‘then’ } L_1 \text{ ‘else’ } L_2 \text{ ‘fi’} \mid Env>}$$

Control (cont'd.)

21. $$\frac{V > 0}{<\text{‘if’ } V \text{ ‘then’ } L_1 \text{ ‘else’ } L_2 \text{ ‘fi’} \mid Env> \; => \; <L_1 \mid Env>}$$

22. $$\frac{V \leq 0}{<\text{‘if’ } V \text{ ‘then’ } L_1 \text{ ‘else’ } L_2 \text{ ‘fi’} \mid Env> \; => \; <L_2 \mid Env>}$$

- Reduction rules for while statements include:

23. $$\frac{<E \mid Env> \; => \; <V \mid Env>, V \leq 0}{<\text{‘while’ } E \text{ ‘do’ } L \text{ ‘od’} \mid Env> \; => \; Env}$$

24. $$\frac{<E \mid Env> \; => \; <V \mid Env>, V > 0}{<\text{‘while’ } E \text{ ‘do’ } L \text{ ‘od’} \mid Env \; => \; < L; \text{ ‘while’ } E \text{ ‘do’ } L \text{ ‘od’} \mid Env>}$$

Implementing Operational Semantics in a Programming Language

- It is possible to implement operational semantic rules directly as a program to get an **executable specification**
- This is useful for two reasons:
  - Allows us to construct a language interpreter directly from a formal specification
  - Allows us to check the correctness of the specification by testing the resulting interpreter
- A possible Prolog implementation for the reduction rules of our sample language will be used

Implementing Operational Semantics in a Programming Language (cont'd.)

- Example: `3*(4+5)` in Prolog:

```
times(3,plus(4,5))
```

- Example: this program:

```
a := 2+3;
b := a*4;
a := b-5
```

  - Can be represented in Prolog as:

```
seq(assign(a,plus(2,3)),
        seq(assign(b,times(a,4)),assign(a,sub(b,5))))
```

- This is actually a tree representation, and no parentheses are necessary to express grouping

Implementing Operational Semantics in a Programming Language (cont'd.)

- We can write reduction rules (ignoring environment rules for the moment)
- A general reduction rule for expressions:

```
reduce(X,Y) :- ...
```

  - Where X is any arithmetic expression (in abstract syntax) and Y is the result of a single reduction step applied to X
- Example:
  - Rule 3 can be written as:

```
reduce(plus(V1,V2),R) :-
        integer(V1), integer(V2), !, R is V1 + V2
```


Implementing Operational Semantics in a Programming Language (cont'd.)

- Rule 7 becomes:

```
reduce(plus(E,E2),plus(E1,E2)) :- reduce(E,E1)
```

- Rule 10 becomes:

```
reduce(plus(V,E),plus(V,E1)) :-
        integer(V), !, reduce(E,E1)
```

- Rule 14 presents a problem if written as:

```
reduce(E,E2) :- reduce(E,E1), reduce(E1,E2)
```

  - Infinite recursive loops will result
- Instead, write rule 14 as two rules:

```
reduce_all(V,V) :- integer(V), !.
reduce_all(E,E2) :- reduce(E,E1), reduce_all(E1,E2)
```

Implementing Operational Semantics in a Programming Language (cont'd.)

- Now extend to environments and control: a pair `<E|Env>` can be thought of as a configuration and written in Prolog as `config(E,Env)`
- Rule 15 then becomes:

```
reduce(config(I,Env),config(V,Env)) :-
                          atom(I), !, lookup(Env, I, V)
```

  - Where `atom(I)` tests for a variable and `lookup` operation finds values in an environment

Implementing Operational Semantics in a Programming Language (cont'd.)

- Rule 16 becomes:

```
reduce(config(assign(I,V),Env),Env1) :-
      integer(V), !, update(Env, value(I,V), Env1)
```

  - Where `update` inserts the new value `V` for `I` into `Env`, yielding `Env1`
- Any dictionary structure for which `lookup` and `update` can be defined can be used to represent an environment in this code

Denotational Semantics

- Denotational semantics use functions to describe the semantics of a programming language
  - A function associates semantic values to syntactically correct constructs
- Example: a function that maps an integer arithmetic expression to its value:

  $Val : \text{Expression} \rightarrow \text{Integer}$
  - **Syntactic domain**: domain of a semantic function
  - **Semantic domain**: range of a semantic function, which is a mathematical structure

Denotational Semantics (cont'd.)

- Example: `val(2+3*4) = 14`
  - Set of integers is the semantic domain
  - `val` maps the syntactic construct `2+3*4` to the semantic value `14`; it **denotes** the value `14`
- A program can be viewed as something that receives input and produces output
- Its semantics can be represented by a function:

  $P : \text{Program} \rightarrow (\text{Input} \rightarrow \text{Output})$
  - Semantic domain is a set of functions from input to output
  - Semantic value is a function

Denotational Semantics (cont'd.)

- Since semantic domains are often functional domains, and values of semantic functions will be functions themselves, we will assume the symbol "→" is right associative and drop the parentheses:

  $P : \text{Program} \to \text{Input} \to \text{Output}$

- Three parts of a denotational definition of a program:
  - Definition of the **syntactic domains**
  - Definition of the **semantic domains**
  - Definition of the semantic functions themselves (sometimes called **valuation functions**)

Syntactic Domains

- **Syntactic domains**:
  - Are defined in denotational definition using notation similar to abstract syntax
  - Are viewed as sets of syntax trees whose structure is given by grammar rules that recursively define elements of the set
- Example: D: Digit

  N: Number

  $N \to N\,D \mid D$

  $D \to \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'}$

Semantic Domains

- **Semantic domains**: sets in which semantic functions take their values
  - Like syntactic domains but may also have additional mathematical structure, depending on use
- Example: integers have arithmetic operations
- Such domains are **algebras**, which are specified by listing their functions and properties
  - Denotational definition of semantic domains lists the sets and operations but usually omits the properties of the operations

Semantic Domains (cont'd.)

- Domains sometimes need special mathematical structures that are the subject of **domain theory**
  - Term domain is sometimes reserved for an algebra with the structure of a complete partial order
  - This structure is needed to define the semantics of recursive functions and loops
- Example: semantic domain of the integers:

Domain v: Integer $= \{\ldots, -2, -1, 0, 1, 2, \ldots\}$

Operations

$+$: Integer $\times$ Integer $\rightarrow$ Integer

$-$: Integer $\times$ Integer $\rightarrow$ Integer

$*$: Integer $\times$ Integer $\rightarrow$ Integer

Semantic Functions

- **Semantic function**: specified for each syntactic domain
- Each function is given a different name based on its associated syntactic domain, usually with boldface letters
- Example: value function from the syntactic domain Digit to the integers:

  $\boldsymbol{D}$: Digit $\rightarrow$ Integer

Semantic Functions (cont'd.)

- Value of a semantic function is specified recursively on the trees of syntactic domains using the structure of grammar rules
- **Semantic equation** corresponding to each grammar rule is given
- Example: grammar rule for digits: $D \rightarrow$ '0' | '1' | . . . | '9'
  - Gives rise to syntax tree nodes:

$$\begin{array}{ccccc} D & & D & \ldots & D \\ | & & | & & | \\ \text{'0'} & & \text{'1'} & & \text{'9'} \end{array}$$

Semantic Functions (cont'd.)

- Example (cont'd.):
  - Semantic function **D** is defined by these semantic equations representing the value of each leaf:

$$\boldsymbol{D}\left(\begin{array}{c} D \\ | \\ \text{`0'} \end{array}\right) = 0, \qquad \boldsymbol{D}\left(\begin{array}{c} D \\ | \\ \text{`1'} \end{array}\right) = 1, \ldots, \qquad \boldsymbol{D}\left(\begin{array}{c} D \\ | \\ \text{`9'} \end{array}\right) = 9$$

  - This notation is shorted to the following:

$$\boldsymbol{D}[[\text{`0'}]] = 0, \boldsymbol{D}[[\text{`1'}]] = 1, \ldots, \boldsymbol{D}[[\text{`9'}]] = 9$$

  - Double brackets `[[…]]` indicate that the argument is a syntactic entity consisting of a syntax tree node with the listed arguments as children

Semantic Functions (cont'd.)

- Example: semantic function from numbers to integers: $N : \text{Number} \to \text{Integer}$
  - Is based on the syntax: $N \to N\,D \mid D$
  - And is given by these equations:

$$\boldsymbol{N}[[ND]] = 10 * \boldsymbol{N}[[N]] + \boldsymbol{N}[[D]]$$
$$\boldsymbol{N}[[D]] = \boldsymbol{D}[[D]]$$

  - Where [[ND]] refers to the tree node $\begin{array}{c} N \\ / \ \backslash \\ N \quad D \end{array}$
  - And [[D]] refers to the node $\begin{array}{c} N \\ | \\ D \end{array}$

Denotational Semantics of Integer Arithmetic Expressions

Syntactic Domains

E: Expression
N: Number
D: Digit

$E \to E_1$ '+' E_2 | E_1 '−' E_2 | E_1 '*' E_2
| '(' E ')' | N
$N \to N\,D$ | D
$D \to$ '0' | 1' | . . . | '9'

Semantic Domains

Domain v: Integer = $\{\ldots, -2, -1, 0, 1, 2, \ldots$
Operations
+ : Integer × Integer → Integer
− : Integer × Integer → Integer
* : Integer × Integer → Integer

Semantic Functions

$\boldsymbol{E}$: Expression → Integer

$\boldsymbol{E}[[E_1 \text{ '+' } E_2]] = \boldsymbol{E}[[E_1]] + \boldsymbol{E}[[E_2]]$
$\boldsymbol{E}[[E_1 \text{ '−' } E_2]] = \boldsymbol{E}[[E_1]] - \boldsymbol{E}[[E_2]]$
$\boldsymbol{E}[[E_1 \text{ '*' } E_2]] = \boldsymbol{E}[[E_1]] * \boldsymbol{E}[[E_2]]$
$\boldsymbol{E}[[\text{'(' } E \text{ ')'}]] = \boldsymbol{E}[[E]]$
$\boldsymbol{E}[[N]] = \boldsymbol{N}[[N]]$

$\boldsymbol{N}$: Number → Integer

$\boldsymbol{N}[[ND]] = 10 * \boldsymbol{N}[[N]] + \boldsymbol{N}[[D]]$
$\boldsymbol{N}[[D]] = \boldsymbol{D}[[D]]$

$\boldsymbol{D}$: Digit → Integer

$\boldsymbol{D}[[\text{'0'}]] = 0, \boldsymbol{D}[[\text{'1'}]] = 1, \ldots, \boldsymbol{D}[[\text{'9'}]] = 9$

Denotational Semantics of Integer Arithmetic Expressions (cont'd.)

- Using these equations to obtain the semantic value of an expression, we compute $\boldsymbol{E}[[(2 + 3)*4]]$ or more precisely, $\boldsymbol{E}[[\text{'(' '2' '+' '3' ')' '*' '4'}]]$

$$
\begin{aligned}
&\boldsymbol{E}[[\text{'(' '2' '+' '3' ')' '*' '4'}]]\\
&\quad = \boldsymbol{E}[[\text{'(' '2' '+' '3' ')'}]] * \boldsymbol{E}[[\text{'4'}]]\\
&\quad = \boldsymbol{E}[[\text{'2' '+' '3'}]] * \boldsymbol{N}[[\text{'4'}]]\\
&\quad = (\boldsymbol{E}[[\text{'2'}]] + \boldsymbol{E}[[\text{'3'}]]) * \boldsymbol{D}[[\text{'4'}]]\\
&\quad = (\boldsymbol{N}[[\text{'2'}]] + \boldsymbol{N}[[\text{'3'}]]) * 4\\
&\quad = \boldsymbol{D}[[\text{'2'}]] + \boldsymbol{D}[[\text{'3'}]]) * 4\\
&\quad = (2 + 3) * 4 = 5 * 4 = 20
\end{aligned}
$$

Environments and Assignments

- First extension to our sample language adds identifiers, assignment statements, and environments
- Environments are functions from identifiers to integers (or undefined)
- Set of environments becomes a new semantic domain:

  Domain *Env*: Environment = Identifier $\rightarrow$ Integer $\cup$ {undef}

Environments and Assignments (cont'd.)

- In denotational semantics, the value `undef` is called **bottom**, from the theory of partial orders, and is denoted by the symbol $\perp$
- Semantic domains with this value are called **lifted domains** and are subscripted with the symbol $\perp$
- The initial environment defined previously can now be defined as: $Env_0(I) = \perp$ for all identifiers I.
- Semantic value of an expression becomes a function from environments to integers:

  $\boldsymbol{E}$: Expression $\rightarrow$ Environment $\rightarrow$ Integer $\perp$

Environments and Assignments (cont'd.)

- The value of an identifier is its value in the environment provided as a parameter:

  $E[[I]](Env) = Env(I)$

- For a number, the environment is immaterial:

  $E[[N]](Env) = N[[N]]$

- For statements and statement lists, the semantic values are functions from environments to environments
  - The "&" notation is used to add values to functions that we have used in previous sections

Syntactic Domains

P: Program
L: Statement-list
S: Statement
E: Expression
N: Number
D: Digit
I: Identifier
A: Letter

$P \to L$
$L \to L_1$ ';' L_2 | S
$S \to I$ ':=' E
$E \to E_1$ '+' E_2 | E_1 '–' E_2 | E_1 '*' E_2
| '(' E ')' | I | N
$N \to N\,D$ | D
$D \to$ '0' | '1' |...| '9'
$I \to I\,A$ | A
$A \to$ 'a' | 'b' |...| 'z'

Figure 12.5 A denotational definition for the sample language extended with assignment statements and environments (*continues*)

Semantic Domains

Domain v: Integer = {. . ., −2, −1, 0, 1, 2,. . .}
Operations

$+$: Integer × Integer → Integer
$-$: Integer × Integer → Integer
$*$: Integer × Integer → Integer

Domain Env: Environment = Identifier → $\text{Integer}_\perp$

Semantic Functions

P : Program → Environment

$P[[L]] = L[[L]](Env_0)$

L : Statement-list → Environment → Environment

$L[[L_1 \text{ ‘;’ } L_2]] = L[[L_2]] \circ L[[L_1]]$
$L[[S]] = S[[S]]$

Figure 12.5 A denotational definition for the sample language extended with assignment statements and environments (*continues*)

S : Statement → Environment → Environment

$S[[\, I \text{ ‘:=’ } E \,]](Env) = Env \ \& \ \{I = E[[E]](Env)\}$

E : Expression → Environment → $\text{Integer}_\perp$

$E[[E_1 \text{ ‘+’ } E_2]](Env) = E[[E_1]](Env) + E[[E_2]](Env)$
$E[[E_1 \text{ ‘−’ } E_2]](Env) = E[[E_1]](Env) - E[[E_2]](Env)$
$E[[E_1 \text{ ‘*’ } E_2]](Env) = E[[E_1]](Env) * E[[E_2]](Env)$
$E[[\text{‘(’ } E \text{ ‘)’}]](Env) = E[[E]](Env)$
$E[[I]](Env) = Env(\, I \,)$
$E[[N]](Env) = N[[N]]$

N: Number → Integer

$N[[ND]] = 10 * N[[N]] + N[[D]]$
$N[[D]] = D[[D]]$

D : Digit → Integer

$D[[\text{‘0’}]] = 0, D[[\text{‘1’}]] = 1, \ldots, D[[\text{‘9’}]] = 9$

Figure 12.5 A denotational definition for the sample language extended with assignment statements and environments

Denotational Semantics of Control Statements

- if and while statements have this abstract syntax:

  S: Statement

  $S \rightarrow I$ ':=' E

  | 'if' E 'then' L_1 'else' L_2 'fi'

  | 'while' E 'do' L 'od'

- Denotational semantics is given by a function from environments to environments:

  S : Statement $\rightarrow$ Environment $\rightarrow$ Environment

- Semantic function of the if statement:

  S[['if' E 'then' L_1 'else' L_2 'fi']](Env) =

  if E[[E]](Env) > 0 then L[[L_1]](Env) else L[[L_2]](Env)

Denotational Semantics of Control Statements (cont'd.)

- Semantic function for the while statement is more difficult
  - Can construct a function as a set by successively extending it to a **least-fixed-point solution**, the "smallest" solution satisfying the equation
  - Here, F will be a function on the semantic domain of environments
- Must also deal with nontermination in loops by assigning the "undefined" value $\perp$

Denotational Semantics of Control Statements (cont'd.)

- The domain of environments becomes a lifted domain:

$$\text{Environment}_{\perp} = (\text{Identifier} \rightarrow \text{Integer}_{\perp})_{\perp}$$

- Semantic function for statements is defined as:

$$S : \text{Statement} \rightarrow \text{Environment}_{\perp} \rightarrow \text{Environment}_{\perp}$$

Implementing Denotational Semantics in a Programming Language

- We will use Haskell for a possible implementation of the denotational functions of the sample language
- Abstract syntax of expressions:

```
data Expr = Val Int | Ident String | Plus Expr Expr
                    | Minus Expr Expr | Times Expr Expr
```

- We ignore the semantics of numbers and simply let values be integers

Implementing Denotational Semantics in a Programming Language (cont'd.)

- Assume we have defined an Environment type with a lookup and update operation
- The `E` evaluation function can be defined as:

```
exprE :: Expr -> Environment -> Int
exprE (Plus e1 e2) env = (exprE e1 env) + (exprE e2 env)
exprE (Minus e1 e2) env = (exprE e1 env) - (exprE e2 env)
exprE (Times e1 e2) env = (exprE e1 env) * (exprE e2 env)
exprE (Val n) env = n
exprE (Ident a) env = lookup env a
```


Axiomatic Semantics

- **Axiomatic semantics**: define the semantics of a program, statement, or language construct by describing the effect its execution has on assertions about the data manipulated by the program
- Elements of mathematical logic are used to specify the semantics, including logical axioms
- We consider logical assertions to be statements about the behavior of the program that are true or false at any moment during execution

Axiomatic Semantics (cont'd.)

- **Preconditions**: assertions about the situation just before execution
- **Postconditions**: assertions about the situation just after execution
- Standard notation is to write the precondition inside curly brackets just before the construct and write the postcondition similarly just after the construct:

$\{x = A\}$ `x := x + 1` $\{x = A + 1\}$ or $\{x = A\}$
`x := x + 1`
$\{x = A + 1\}$

Axiomatic Semantics (cont'd.)

- Example: `x := 1 / y`
  - Semantics become:

    $\{y \neq 0\}$
    `x := 1 / y`
    $\{x = 1/y\}$
- Such pre- and postconditions are often capable of being tested for validity during execution, as a kind of error checking
  - Conditions are usually Boolean expressions
- In C, can use the `assert.h` macro library for checking assertions

Axiomatic Semantics (cont'd.)

- An **axiomatic specification** of the semantics of the language construct `C` is of the form $\{P\}\ C\ \{Q\}$
  - Where `P` and `Q` are assertions
  - If `P` is true just before execution of `C`, then `Q` is true just after execution of `C`
- This representation of the action of `C` is not unique and may not completely specify all actions of `C`
- **Goal-oriented activity**: way to associate to C a general relation between precondition P and postcondition Q
  - Work backward from the goal to the requirements

Axiomatic Semantics (cont'd.)

- There is one precondition `P` that is the **most general** or **weakest** assertion with the property that $\{P\}\ C\ \{Q\}$
  - Called the **weakest precondition** of postcondition `Q` and construct `C`
  - Written as $wp(C,Q)$
- Can now restate the property as

  $\{P\}\ C\ \{Q\}$ if and only if $P \rightarrow wp(C,Q)$

Axiomatic Semantics (cont'd.)

- We define the axiomatic semantics of language construct `C` as the function $wp(C,\_)$ from assertion to assertion
  - Called a **predicate transformer**: takes a predicate as argument and returns a predicate result
  - Computes the weakest precondition from any postcondition
- Example assignment can now be restated as:

$$wp(\texttt{x := 1/y, x = 1/y}) = \{\texttt{y} \neq 0\}$$

General Properties of *wp*

- Predicate transformer $wp(C,Q)$ has certain properties that are true for almost all language constructs `C`
- **Law of the Excluded Miracle**: $wp(C,\text{false}) = \text{false}$
  - There is nothing a construct C can do that will make false into true
- **Distributivity of Conjunction**:

$$wp(C,P \text{ and } Q) = wp(C,P) \text{ and } wp(C,Q)$$

- **Law of Monotonicity**:

$$\text{if } Q \to R \text{ then } wp(C,Q) \to wp(C,R)$$

General Properties of *wp* (cont'd.)

- **Distributivity of Disjunction**:

  $wp(C,P) \text{ or } wp(C,Q) \rightarrow wp(C,P \text{ or } Q)$

- The last two properties regard implication operator "→" and "or" operator with equality if `C` is deterministic
- The question of determinism adds complexity
  - Care must be taken when talking about any language construct

Axiomatic Semantics of the Sample Language

- The specification of the semantics of expressions alone is not commonly included in an axiomatic specification
- Assertions in an axiomatic specificator are primarily statements about the side effects of constructs
  - They are statements involving identifiers and environments

Axiomatic Semantics of the Sample Language (cont'd.)

- Abstract syntax for which we will define the *wp* operator:

$$
\begin{aligned}
&P \rightarrow L \\
&L \rightarrow L_1 \text{ `;' } L_2 \mid S \\
&S \rightarrow I \text{ `:=' } E \\
&\qquad\quad \mid \text{ `if' } E \text{ `then' } L_1 \text{ `else' } L_2 \text{ `fi'} \\
&\qquad\quad \mid \text{ `while' } E \text{ `do' } L \text{ `od'}
\end{aligned}
$$

- The first two rules do not need separate specifications
  - The *wp* operator for program `P` is the same as for its associated statement-list `L`

Axiomatic Semantics of the Sample Language (cont'd.)

- **Statement-lists**: for lists of statements separated by a semicolon, we have:

$$wp(L_1; L_2, Q) = wp(L_1, wp(L_2, Q))$$

  - The weakest precondition of a series of statements is the composition of the weakest preconditions of its parts
- **Assignment statements**: definition of *wp* is:

$$wp(I := E, Q) = Q[E/I]$$

  - $Q[E/I]$ is the assertion `Q`, with `E` replacing all free occurrences of the identifier `I` in `Q`

Axiomatic Semantics of the Sample Language (cont'd.)

- Recall that an identifier `I` is **free** in a logical assertion `Q` if it is not **bound** by either the existential quantifier "there exists" or the universal quantifier "for all"
- $wp(I := E, Q) = Q[E/I]$ says that for `Q` to be true after the assignment `I:=E`, whatever `Q` says about `I` must be true about `E` before the assignment is executed
- **If statements**: our semantics of the if statement state that the expression is true if it is greater than 0 and false otherwise

Axiomatic Semantics of the Sample Language (cont'd.)

- Given the if statement: $\texttt{if } E \texttt{ then } L_1 \texttt{ else } L_2 \texttt{ fi}$
- The weakest precondition is defined as:

$$wp(\texttt{if } E \texttt{ then } L_1 \texttt{ else } L_2 \texttt{ fi}, Q) =$$
$$(E > 0 \rightarrow wp(L_1,Q)) \text{ and } (E \leq 0 \rightarrow wp(L_2,Q))$$

- **While statements**: $\texttt{while } E \texttt{ do } L \texttt{ od}$ executes as long as `E>0`
- Must give an inductive definition based on the number of times the loop executes
- Let $H_i(\texttt{while } E \texttt{ do } L \texttt{ od}, Q)$ be a statement that the loop executes `I` times and terminates satisfying `Q`

Axiomatic Semantics of the Sample Language (cont'd.)

- Then $H_0(\texttt{while}\ E\ \texttt{do}\ L\ \texttt{od},\ Q) = E \le 0 \text{ and } Q$
- And $H_1(\texttt{while}\ E\ \texttt{do}\ L\ \texttt{od},\ Q) = E > 0 \text{ and } wp(L, Q \text{ and } E \le 0)$
  $= E > 0 \text{ and } wp(L, H_0(\texttt{while}\ E\ \texttt{do}\ L\ \texttt{od},\ Q))$
- Continuing, we have in general that:

  $H_{i+1}(\texttt{while}\ E\ \texttt{do}\ L\ \texttt{od},\ Q) =$
  $E > 0 \text{ and } wp(L, H_i(\texttt{while}\ E\ \texttt{do}\ L\ \texttt{od},\ Q))$
- Now we define:

  $wp(\texttt{while}\ E\ \texttt{do}\ L\ \texttt{od},\ Q)$
  $= \text{there exists an } i \text{ such that } H_i(\texttt{while}\ E\ \texttt{do}\ L\ \texttt{od},\ Q)$

Axiomatic Semantics of the Sample Language (cont'd.)

- Note that this definition of the semantics of the while requires that the loop terminates
- A non-terminating loop always has false as its weakest precondition (it can never make a postcondition true)

  $wp(\texttt{while 1 do}\ L\ \texttt{od},\ Q) = \text{false, for all } L \text{ and } Q$
- These semantics for loops are difficult to use in the area of proving correctness of programs

Proofs of Program Correctness

- The major application of axiomatic semantics is as a tool for proving correctness of programs
- Recall that C satisfies a specification $\{P\}\ C\ \{Q\}$, provided $P \rightarrow wp(C,Q)$
- To prove correctness:
  1. Compute *wp* from the axiomatic semantics and general properties of *wp*
  2. Show that $P \rightarrow wp(C,Q)$

Proofs of Program Correctness (cont'd.)

- To show that a while-statement is correct, we only need an **approximation** of its weakest precondition, that is some assertion `W` such that

  $W \rightarrow wp(\texttt{while}\ldots, Q)$

- If we can show that `P`→`W`, we have also shown the correctness of `{P} while… {Q}`, since `P`→`W` and `W`→`wp(while…,Q)` imply that `P`→`wp(while…,Q)`

Proofs of Program Correctness (cont'd.)

- Given the loop `while` E `do` L `od`, we need to find an assertion `W` such that these conditions are true:
  (a) W and $(E > 0) \rightarrow wp(L,W)$
  (b) W and $(E \le 0) \rightarrow Q$
  (c) $P \rightarrow W$
  - Every time the loop executes, `W` continues to be true by condition (a)
  - When the loop terminates, (b) says `Q` must be true
  - (c) implies that `W` is the required approximation for
    $wp(\texttt{while}\ldots, Q)$

Proofs of Program Correctness (cont'd.)

- An assertion `W` satisfying condition (a) is called a loop invariant for the loop, since a repetition of the loop leaves `W` true
  - In general, loops have many invariants `W`
  - Must find an appropriate `W` that also satisfies conditions (b) and (c)