Denotational Semantics

Basic Idea
Map syntactic objects into domains of mathematical objects.

\[ \text{meaning} : \text{Syntax} \rightarrow \text{Semantics} \]

Example
\[ \begin{align*}
\text{meaning } [26/2] &= \text{meaning } [(10+3)] \\
\end{align*} \]

The phrase “10+3” denotes the mathematical object 13.

The abstract object 13 (the number 13) is the denotation of the phrase “10+3”.

Syntactic World

Syntactic categories or syntactic domains
Collections of syntactic objects that may occur in phrases in the definition of the syntax of the language:

- Numeral, Command, and Expression.

Each syntactic domain has a special metavariable associated with it to stand for elements in the domain:

\[ \begin{align*}
C & : \text{Command} \\
E & : \text{Expression} \\
N & : \text{Numeral} \\
I & : \text{Identifier} \\
O & : \text{Operator}.
\end{align*} \]

Colon means “element of”.
Subscripts are allowed.

Abstract production rules
Possible patterns that the abstract syntax trees of language phrases may take.

Use the syntactic categories or the metavariables for elements of the categories:

Command ::= while Expression do Command

E ::= N E E E E E E E E E
use E ::= N E E E E E E E E

to distinguish instances

O ::= + | - | * | /

See Chapter 1 for more on abstract syntax.

Semantic World

Semantic domains
“Sets” of mathematical objects.

Sets serving as domains have a lattice-like structure that will be described in Chapter 10.

Boolean = \{ true, false \} is set of truth values

Integer = \{ …, -2, -1, 0, 1, 2, 3, 4, … \} is the set of integers

Store = (Variable → Integer)
Consists of sets of bindings (functions) of variables to integers.

A → B denotes the set of functions with domain A and codomain B.
Language of Numerals

Syntactic Domains

N : Numeral -- nonnegative numerals
D : Digit -- decimal digits

Abstract Production Rules

Numeral ::= Digit | Numeral Digit
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Semantic Domain

Number = { 0,1,2,3,4,... } -- natural numbers

Semantic Functions

value : Numeral → Number
digit : Digit → Number

Semantic Equations

\[
\text{value}[N \ D] = \text{plus}(\text{times}(10, \text{value}[N]), \text{digit}[D])
\]
\[
\text{value}[D] = \text{digit}[D]
\]
\[
\text{digit}[0] = 0 \quad \text{digit}[5] = 5
\]
\[
\text{digit}[1] = 1 \quad \text{digit}[6] = 6
\]
\[
\text{digit}[2] = 2 \quad \text{digit}[7] = 7
\]
\[
\text{digit}[3] = 3 \quad \text{digit}[8] = 8
\]
\[
\text{digit}[4] = 4 \quad \text{digit}[9] = 9
\]

Denotational Evaluation

\[
\text{value}[905] = \text{plus}(\text{times}(10, \text{value}[90]), \text{digit}[5])
\]
\[
= \text{plus}(\text{times}(10, \text{plus}(\text{times}(10, \text{value}[9]), \text{digit}[0])), 5)
\]
\[
= \text{plus}(\text{times}(10, \text{plus}(\text{times}(10, \text{digit}[9]), 0)), 5)
\]
\[
= \text{plus}(\text{times}(10, \text{plus}(\text{times}(10, 90), 0)), 5)
\]
\[
= \text{plus}(\text{times}(10, 900), 5)
\]
\[
= 905
\]
Compositional Definitions

The meaning of a language construct is defined in terms of the meanings of its subphrases.

Three reasons for using compositional definitions:

1. Each phrase of a language is given a meaning that describes its contribution to the meaning of a complete program that contains it.

   The meaning of each phrase is formulated as a function of the meanings of its immediate subphrases.

   As a result, whenever two phrases have the same meaning, one can be replaced by the other without changing the meaning of the program (substitutivity of semantically equivalent phrases).

2. Since a compositional definition parallels the syntactic structure of its BNF specification, properties of constructs in the language can be verified by structural induction.

   Denotational definitions are compositional.

Homomorphisms

Consider a function \( H : A \rightarrow B \) where \( A \) has a binary operation \( f : A \times A \rightarrow A \) and \( B \) has a binary operation \( g : B \times B \rightarrow B \).

The function \( H \) is a homomorphism if \( H(f(x,y)) = g(H(x),H(y)) \).

The semantic function \( \text{value} \) is a homomorphism.

In Figure 9.1 the operation \( f \) is concatenation and \( g(m,n) = \text{plus}(\text{times}(10, m), n) \).

Therefore, \( \text{value}(f(x,y)) = g(\text{value}(x),\text{value}(y)) \), thus demonstrating that \( \text{value} \) is a homomorphism.

A Calculator Language

Three-function calculator

A “program” on this calculator consists of a sequence of keystrokes usually alternating between operands and operators.

Keystrokes: \( 15 + 7 \times 2 + 30 = \)

Resulting Display: 74

Ignore unusual combinations of keystrokes.

Concrete Syntax

\[
\begin{align*}
\langle \text{program} \rangle & ::= \langle \text{expression sequence} \rangle \\
\langle \text{expression sequence} \rangle & ::= \langle \text{expression} \rangle \\
\langle \text{expression} \rangle & ::= \langle \text{term} \rangle \\
\langle \text{term} \rangle & ::= \langle \text{numeral} \rangle \mid \text{M}R \mid \text{Clear} \\
\langle \text{numeral} \rangle & ::= \langle \text{digit} \rangle \mid \langle \text{numeral} \rangle \langle \text{digit} \rangle \\
\langle \text{digit} \rangle & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle \text{operator} \rangle & ::= + \mid - \mid \times \\
\langle \text{answer} \rangle & ::= \text{M}+ \mid = \\
\langle \text{numeral} \rangle & ::= \langle \text{digit} \rangle \mid \langle \text{numeral} \rangle \langle \text{digit} \rangle \\
\langle \text{digit} \rangle & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]
Abstract Syntax

**Abstract Syntactic Domains**

- **P**: Program
- **O**: Operator
- **S**: ExprSequence
- **A**: Answer
- **E**: Expression
- **N**: Numeral

**Abstract Production Rules**

- **Program** ::= ExprSequence
- **ExprSequence** ::= Expression | Expression ExprSequence
- **Expression** ::= Numeral | MR | Clear | Expression Operator Expression | Expression Answer
- **Operator** ::= + | – | x
- **Answer** ::= M+ | = | +/-
- **Numeral** ::= see Figure 9.1

A Keystroke Sequence

15 + 7 x 2 + 30 = +/- M+ 25 +/- x 3 +/- + 40 M+ M^R

Concrete Syntax

<program>
<expr>
<expr>
<ans>
<expr>
<op>
<term>
<expr>
<op>
<term>
<term>
25
+/-
M+
<term>
3
+/-<term>

Abstract Syntax Tree:
Calculator Semantics

A state maintains four values that model the internal working of the calculator:

1. Internal Accumulator
   Maintains a running total value of the operations carried out so far

2. Operator Flag
   Indicates pending operation to be calculated when another operand occurs

3. Current Display
   Portrays the latest numeral entered, partial results, or an answer

4. Memory
   Contains one saved value, initially zero

Semantic Domains

Integer = \{ \ldots , -2, -1, 0, 1, 2, 3, 4, \ldots \} 

Operation = \{ plus, minus, times, nop \} 

State = Integer x Operation x Integer x Integer

Auxiliary Operations (semantics)

\begin{align*}
\text{plus} & : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \\
\text{minus} & : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \\
\text{times} & : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \\
\text{nop} & : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \\
\text{where } \text{nop}(a,d) &= d
\end{align*}

Sample Computation

<table>
<thead>
<tr>
<th>Key</th>
<th>Acc</th>
<th>OprFlag</th>
<th>Dsply</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Initial 0 =&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>nop</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>15</td>
<td>plus</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>plus</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>22</td>
<td>times</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>times</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>44</td>
<td>plus</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>44</td>
<td>plus</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>=</td>
<td>44</td>
<td>nop</td>
<td>74</td>
<td>0</td>
</tr>
<tr>
<td>+/-</td>
<td>44</td>
<td>nop</td>
<td>-74</td>
<td>0</td>
</tr>
<tr>
<td>M^+</td>
<td>44</td>
<td>nop</td>
<td>-74</td>
<td>-74</td>
</tr>
<tr>
<td>25</td>
<td>44</td>
<td>nop</td>
<td>25</td>
<td>-74</td>
</tr>
<tr>
<td>+/-</td>
<td>44</td>
<td>nop</td>
<td>-25</td>
<td>-74</td>
</tr>
<tr>
<td>x</td>
<td>-25</td>
<td>times</td>
<td>-25</td>
<td>-74</td>
</tr>
<tr>
<td>3</td>
<td>-25</td>
<td>times</td>
<td>3</td>
<td>-74</td>
</tr>
<tr>
<td>+/-</td>
<td>-25</td>
<td>times</td>
<td>-3</td>
<td>-74</td>
</tr>
<tr>
<td>+</td>
<td>75</td>
<td>plus</td>
<td>75</td>
<td>-74</td>
</tr>
<tr>
<td>40</td>
<td>75</td>
<td>plus</td>
<td>40</td>
<td>-74</td>
</tr>
<tr>
<td>M^+</td>
<td>75</td>
<td>nop</td>
<td>115</td>
<td>41</td>
</tr>
<tr>
<td>MR</td>
<td>75</td>
<td>nop</td>
<td>41</td>
<td>41</td>
</tr>
</tbody>
</table>

Semantic Functions

One semantic function for each syntactic domain:

\begin{align*}
\text{meaning} & : \text{Program} \rightarrow \text{Integer} \\
\text{perform} & : \text{ExprSequence} \rightarrow (\text{State} \rightarrow \text{State}) \\
\text{evaluate} & : \text{Expression} \rightarrow (\text{State} \rightarrow \text{State}) \\
\text{compute} & : \text{Operator} \rightarrow (\text{State} \rightarrow \text{State}) \\
\text{calculate} & : \text{Answer} \rightarrow (\text{State} \rightarrow \text{State}) \\
\text{value} & : \text{Numeral} \rightarrow \text{Integer} \\
\end{align*}

\text{value} \text{ uses only nonnegative integers}
### Semantic Equations

**meaning** $[P] = d$

where $(a,op,d,m) = perform \ [P] (0,nop,0,0)$

$perform \ [E \ S] = perform \ [S] \circ evaluate \ [E]$  

$perform \ [E] = evaluate \ [E]$  

$evaluate \ [N] (a,op,d,m) = (a,op,v,m)$

where $v = value \ [N]$  

$evaluate \ [M^R] (a,op,d,m) = (a,op,m,m)$  

$evaluate \ [Clear] (a,op,d,m) = (0,nop,0,0)$  

$evaluate \ [E_1 \ O \ E_2] = evaluate \ [E_2] \circ compute \ [O] \circ evaluate \ [E_1]$  

$evaluate \ [E \ A] = calculate \ [A] \circ evaluate \ [E]$  

$compute \ [+] (a,op,d,m) = (op(a,d),plus,op(a,d),m)$  

$compute \ [-] (a,op,d,m) = (op(a,d),minus,op(a,d),m)$  

$compute \ [x] (a,op,d,m) = (op(a,d),times,op(a,d),m)$  

$calculate \ [=] (a,op,d,m) = (a,nop,op(a,d),m)$  

$calculate \ [M^+] (a,op,d,m) = (a,nop,v,plus(m,v))$  

where $v = op(a,d)$  

$calculate \ [^/] (a,op,d,m) = (a,op,minus(0,d),m)$  

$calculate \ [\mathit{value}] \ [N] = \text{usual denotational definition of nonnegative numerals}$

### Denotational Evaluation

Consider the series of keystrokes:

"8 +/- + 5 x 3 ="

Meaning of the sequence given by:

```
meaning [8 +/- + 5 x 3 =] = d where 
(a,op,d,m) = perform [8 +/- + 5 x 3 =] (0,nop,0,0).
```

The evaluation proceeds:

$perform \ [8 +/- + 5 x 3 =] (0,nop,0,0)$  

$= evaluate \ [8 +/- + 5 x 3 =] (0,nop,0,0)$  

$= (calculate [=] \circ evaluate [8 +/- + 5 x 3]) (0,nop,0,0)$  

$= (calculate [=] \circ evaluate [3] \circ compute [x] \circ 
\circ evaluate [8 +/- + 5]) (0,nop,0,0)$  

$= (calculate [=] \circ evaluate [3] \circ compute [x] \circ 
\circ evaluate [5] \circ compute [+] \circ 
\circ evaluate [8 +/-]) (0,nop,0,0)$  

$= (calculate [=] \circ evaluate [3] \circ compute [x] \circ 
\circ evaluate [5] \circ compute [+] \circ 
\circ calculate [\mathit{value}] \ (8)) (0,nop,0,0)$  

$= (calculate [=] \circ evaluate [3] \circ compute [x] \circ 
\circ evaluate [5] \circ compute [+] \circ 
\circ calculate [\mathit{value}] \ (8) (0,nop,8,0))$  

$= (calculate [=] \circ evaluate [3] \circ compute [x] \circ 
\circ evaluate [5] \circ compute [+] \circ 
\circ calculate [\mathit{value}] \ (8 (0,nop,8,0))))$
Abstract Syntax for Wren

Abstract Syntactic Domains
- P: Program
- C: Cmd
- N: Numeral
- D: Declaration
- E: Expr
- I: Identifier
- T: Type
- O: Operator

Abstract Production Rules
- Program ::= program Identifier is Declaration* begin Cmd end
- Declaration ::= var Identifier+: Type ;
- Type ::= integer | boolean
- Cmd ::= Cmd ; Cmd | Identifier ::= Expr
  - skip | if Expr then Cmd else Cmd
  - if Expr then Cmd | while Expr do Cmd
- Expr ::= Numeral | Identifier | true | false
  - Expr | Expr Operator Expr | not(Expr)
- Operator ::= + | – | * | / | or | and
  - <= | < | = | > | >= | <>

Denotational Semantics of Wren

Imperative programming languages
1. Programs consist of commands, hence the term “imperative”.
2. Programs operate on a global data structure, called a store, in which results are generally computed by incrementally updating values until a final result is produced.
3. The dominant command is the assignment instruction, which modifies a location in the store.
4. Program control primarily entails sequencing and iteration, represented by the semicolon and the while command in Wren.

Semantic Domains

Primitive Semantic Domains:
- Integer = { … , -2, -1, 0, 1, 2, 3, 4, … }
- Boolean = { true, false }

Compound Semantic Domains:
Product Domains
- Cartesian products, AxB.
- States in the calculator semantics.
- States when IO added back to Wren.
- Used in the auxiliary functions for Wren.

Sum Domains
- Also called disjoint union or disjoint sum.
- A union where elements are tagged to indicate their source.
- Domain of Storable Values:
  - SV = int(Integer) + bool(Boolean)
Function Domains

- Set of functions from A to B, denoted by \( A \rightarrow B \).

- \( f \) is a member of \( A \rightarrow B \) expressed by \( f : A \rightarrow B \).

- Store for Wren is modeled as a function in \( \text{Store} = \text{Identifier} \rightarrow (\text{SV} + \text{undefined}) \).

- Each \( \text{sto} : \text{Store} \) is \textit{undefined} for all but a finite set of identifiers (called a finite function).

- Notational convention: Represent a store as a set of bindings.
  
  \( \text{sto} = \{ \text{count} \mapsto \text{int}(1), \text{total} \mapsto \text{int}(0) \} \)

Assume that \( \text{sto}(I) = \text{undefined} \) for all other identifiers \( I \).

Let \( \{ \} \) represent an everywhere undefined store.

Operations on Stores

- \( \text{emptySto} : \text{Store} \)
  
  \( \forall I : \text{Identifier}, \text{emptySto} I = \text{undefined} \)

- \( \text{updateSto} : \text{Store} \times \text{Identifier} \times \text{SV} \rightarrow \text{Store} \)
  \( \forall X : \text{Identifier}, \text{updateSto}(\text{sto},I,val) X = \) if \( X = I \) then \( val \) else \( \text{sto}(X) \)

- \( \text{applySto} : \text{Store} \times \text{Identifier} \rightarrow \text{SV} + \text{undefined} \)
  \( \text{applySto}(\text{sto},I) = \text{sto}(I) \)

Example

If \( \text{sto} = \{ a \mapsto \text{int}(3), b \mapsto \text{int}(5) \} \),

\( \text{updateSto}(\text{sto},b,8) = \{ a \mapsto \text{int}(3), b \mapsto \text{int}(8) \} \)

and

\( \text{updateSto}(\text{sto},c,-99) = \{ a \mapsto \text{int}(3), b \mapsto \text{int}(5), c \mapsto \text{int}(-99) \} \)

Motivating the Definition

For \( \text{sto} : \text{Store}, \)

\( \text{sto} : \text{Identifier} \rightarrow (\text{SV} + \text{undefined}) \)

We want \( \text{updateSto}(\text{sto},I,val) \) to be a \text{Store} function as well:

For \( \text{sto} : \text{Store}, I : \text{Identifier}, val : \text{SV}, \)

\( \text{updateSto}(\text{sto},I,val) : \text{Identifier} \rightarrow (\text{SV} + \text{undefined}), \)

Define the function \( \text{updateSto}(\text{sto},I,val) \) by showing what it does on an identifier as an argument:

\( \forall X : \text{Identifier}, \)

\( \text{updateSto}(\text{sto},I,val) X = \)

if \( X = I \) then \( val \) else \( \text{sto}(X) \)

Expressible Values

- Values that expressions can produce.

- Expressible values in Wren:
  
  \( \text{EV} = \text{int}(	ext{Integer}) + \text{bool}(\text{Boolean}) \)

Auxiliary Functions

- \( \text{plus} : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \)

- \( \text{minus} : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \)

- \( \text{times} : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \)

- \( \text{divides} : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \)

- \( \text{less} : \text{Integer} \times \text{Integer} \rightarrow \text{Boolean} \)

- \( \text{lesseq} : \text{Integer} \times \text{Integer} \rightarrow \text{Boolean} \)

- \( \text{greater} : \text{Integer} \times \text{Integer} \rightarrow \text{Boolean} \)

- \( \text{greatereq} : \text{Integer} \times \text{Integer} \rightarrow \text{Boolean} \)

- \( \text{equal} : \text{Integer} \times \text{Integer} \rightarrow \text{Boolean} \)

- \( \text{neq} : \text{Integer} \times \text{Integer} \rightarrow \text{Boolean} \)
Semantic Functions

- Generally, one semantic function for each syntactic category.
- No need to consider declarations in the semantics of Wren.

meaning : Program → Store
execute : Command → (Store → Store)
evaluate : Expression → (Store → EV)
value : Numeral → EV

- Imagine an identity semantic function mapping Identifiers as syntax to Identifiers as semantics.
- Operators are distributed into the binary expressions in the abstract syntax.

Semantic Equations

meaning \([\text{program } I \text{ is } D \text{ begin } C \text{ end}]\) =
\[
\text{execute } [C] \text{ emptySto}
\]
execute \([C_1 ; C_2]\) = execute \([C_2]\) \&\& execute \([C_1]\)
execute \([\text{skip}]\) sto = sto
execute \([I := E]\) sto =
\[
\text{updateSto}(\text{sto}, I, (\text{evaluate } [E] \text{ sto}))
\]
evaluate \([\text{if } E \text{ then } C\] sto =
\[
\text{if } p \text{ then execute } [C] \text{ sto else sto}
\]
where bool\((p) = \text{evaluate } [E] \text{ sto}
execute \([\text{if } E \text{ then } C_1 \text{ else } C_2]\) sto =
\[
\text{if } p \text{ then execute } [C_1] \text{ sto else execute } [C_2] \text{ sto}
\]
where bool\((p) = \text{evaluate } [E] \text{ sto}
execute \([\text{while } E \text{ do } C]\) = loop
\[
\text{where loop } \text{sto} =
\]
\[
\text{if } p \text{ then loop(execute } [C] \text{ sto) else sto}
\]
where bool\((p) = \text{evaluate } [E] \text{ sto}

evaluate \([I]\) sto =
\[
\text{if } \text{val=undefined} \text{ then error else val}
\]
where \text{val = applySto}(\text{sto}, I)
evaluate \([N]\) sto = \text{int(value } [N]\)
evaluate \([\text{true}]\) sto = bool(true)
evaluate \([\text{false}]\) sto = bool(false)
evaluate \([E_1 + E_2]\) sto = \text{int(plus(m,n))}
\[
\text{where int(m) = evaluate } [E_1] \text{ sto and int(n) = evaluate } [E_2] \text{ sto}
\]
evaluate \([E_1 / E_2]\) sto =
\[
\text{if } n=0 \text{ then error else int(divides(m,n))}
\]
\[
\text{where int(m) = evaluate } [E_1] \text{ sto and int(n) = evaluate } [E_2] \text{ sto}
\]
evaluate \([E_1 < E_2]\) sto =
\[
\text{if less(m,n) then bool(true) else bool(false)}
\]
\[
\text{where int(m) = evaluate } [E_1] \text{ sto and int(n) = evaluate } [E_2] \text{ sto}
\]
evaluate \([E_1 \text{ and } E_2]\) sto =
\[
\text{if } p \text{ then bool(q) else bool(false)}
\]
\[
\text{where bool(p) = evaluate } [E_1] \text{ sto and bool(q) = evaluate } [E_2] \text{ sto}
\]
evaluate \([E_1 \text{ or } E_2]\) sto =
\[
\text{if } p \text{ then bool(true) else bool(q)}
\]
\[
\text{where bool(p) = evaluate } [E_1] \text{ sto and bool(q) = evaluate } [E_2] \text{ sto}
\]
evaluate \([\neg E]\) sto = \text{int(minus(0,m))}
\[
\text{where int(m) = evaluate } [E] \text{ sto}
\]
evaluate \([\text{not}(E)]\) sto =
\[
\text{if evaluate } [E] \text{ sto = bool(true)}
\]
\[
\text{then bool(false) else bool(true)}
\]
Notational Conventions

- Function application associates to the left.
- \( \rightarrow \) associates to the right.

\[
\text{execute } [a \leftarrow 0; b \leftarrow 1] \text{ emptySto}
\]
means
\[
(\text{execute } [a \leftarrow 0; b \leftarrow 1]) \text{ emptySto}.
\]

\[
\text{execute } : \text{Command} \rightarrow \text{Store} \rightarrow \text{Store}
\]
means
\[
\text{execute } : \text{Command} \rightarrow (\text{Store} \rightarrow \text{Store}).
\]

These conventions agree:

\[
\text{execute } : \text{Command} \rightarrow \text{Store} \rightarrow \text{Store}
\]
\[
\text{execute } [a \leftarrow 0; b \leftarrow 1] : \text{Store} \rightarrow \text{Store}
\]
\[
\text{execute } [a \leftarrow 0; b \leftarrow 1] \text{ emptySto} : \text{Store}
\]

Noncompositional while Definition

\[
\text{execute } \left[ \text{while } E \text{ do } C \right] \text{ sto } =
\]
\[
\text{if evaluate } [E] \text{ sto } = \text{bool}(\text{true})
\]
\[
\text{then execute } \left[ \text{while } E \text{ do } C \right] (\text{execute } [C] \text{ sto})
\]
\[
\text{else sto}
\]

This noncompositional definition of \text{while} can be transformed into the compositional version shown earlier (see Chapter 10).

Handling Dynamic Errors

- Assume each semantic domain includes a special element \text{error} signifying the occurrence of an error.
- All semantic functions propagate \text{error}.
- Nontermination (for \text{while}) modeled indirectly.
- A nonterminating \text{while} loop is an undefined function on some stores.

Semantic Equivalence

Two language constructs are semantically equivalent if they share the same denotation.

\[
\text{while } E \text{ do } C =
\]
\[
\text{if } E \text{ then } (C; \text{ while } E \text{ do } C) \text{ else skip}
\]
\[
\text{execute } \left[ \text{while } E \text{ do } C \right] \text{ sto } =
\]
\[
\text{loop}_1 \text{ sto}
\]
where \[
\text{loop}_1 \text{ sto } =
\]
\[
\text{if evaluate } [E] \text{ sto } = \text{bool}(\text{true})
\]
\[
\text{then loop}_1 (\text{execute } [C] \text{ sto})
\]
\[
\text{else s}
\]
\[
= \text{if evaluate } [E] \text{ sto } = \text{bool}(\text{true})
\]
\[
\text{then loop}_1 (\text{execute } [C] \text{ sto})
\]
\[
\text{else sto}
\]
\[
\text{where loop}_1 \text{ sto } =
\]
\[
\text{if evaluate } [E] \text{ sto } = \text{bool}(\text{true})
\]
\[
\text{then loop}_1 (\text{execute } [C] \text{ sto})
\]
\[
\text{else s}
\]
\[
= \text{if evaluate } [E] \text{ sto } = \text{bool}(\text{true})
\]
\[
\text{then loop}_1 (\text{execute } [C] \text{ sto})
\]
\[
\text{else sto}
\]
\[
\text{where loop}_2 \text{ sto } =
\]
\[
\text{if evaluate } [E] \text{ sto } = \text{bool}(\text{true})
\]
\[
\text{then loop}_2 (\text{execute } [C] \text{ sto})
\]
\[
\text{else sto}
\]
\[
\text{execute } \left[ \text{if } E \text{ then } (C; \text{ while } E \text{ do } C) \right] \text{ else skip} \right] \text{ sto}
\]
\[
= \text{if evaluate } [E] \text{ sto } = \text{bool}(\text{true})
\]
\[
\text{then execute } \left[ C; \text{ while } E \text{ do } C \right] \text{ sto}
\]
\[
\text{else execute } \left[ \text{skip} \right] \text{ sto}
\]
\[
= \text{if evaluate } [E] \text{ sto } = \text{bool}(\text{true})
\]
\[
\text{then (execute } \left[ \text{while } E \text{ do } C \right] (\text{execute } [C] \text{ sto})
\]
\[
\text{else sto}
\]
\[
= \text{if evaluate } [E] \text{ sto } = \text{bool}(\text{true})
\]
\[
\text{then execute } \left[ \text{while } E \text{ do } C \right]
\]
\[
\text{else execute } [C] \text{ sto}
\]
\[
\text{else sto}
\]
\[
= \text{if evaluate } [E] \text{ sto } = \text{bool}(\text{true})
\]
\[
\text{then loop}_2 (\text{execute } [C] \text{ sto})
\]
\[
\text{else sto}
\]
\[
\text{where loop}_2 \text{ sto } =
\]
\[
\text{if evaluate } [E] \text{ sto } = \text{bool}(\text{true})
\]
\[
\text{then loop}_2 (\text{execute } [C] \text{ sto})
\]
\[
\text{else s}
\]

Now observe that \text{loop}_1 and \text{loop}_2 have the same definition.
Input and Output

Files of integers modeled as sets of finite lists of integers.

- Input = Integer*
- Output = Integer*

Meaning of a program defined in terms of these lists.

\[
\text{meaning : Program } \rightarrow \text{ Input } \rightarrow \text{ Output}
\]

Commands may change the input and output lists, so

\[
\text{execute : Command } \rightarrow \text{ State } \rightarrow \text{ State}
\]

where

\[
\text{State } = \text{ Store } \times \text{ Input } \times \text{ Output}.
\]

Use auxiliary functions to manipulate lists:

- \(\text{head : Integer* } \rightarrow \text{ Integer}\)
- \(\text{tail : Integer* } \rightarrow \text{ Integer*}\)
- \(\text{null : Integer* } \rightarrow \text{ Boolean}\)
- \(\text{affix : Integer* } \times \text{ Integer } \rightarrow \text{ Integer*}\)

New Semantic Equations

- \(\text{meaning } \left[\text{program I is D begin C end}\right]\) inp = outp
  where \((\text{sto, inp}, \text{outp}) = \text{execute } \left[\text{C}\right](\text{emptySto}, \text{inp}, [\ ]))\)

- \(\text{execute } \left[\text{read I}\right](\text{sto,inp,outp}) =\)
  - if \(\text{null}(\text{inp})\)
    - then \(\text{error}\)
    - else
      \((\text{updateSto(\text{sto, I,int(head(\text{inp}))}),tail(\text{inp}),\text{outp})}\)

- \(\text{execute } \left[\text{write E}\right](\text{sto,inp,outp}) =\)
  - \((\text{sto, inp, affix(outp,val)})\)
    - where \(\text{int(val)} = \text{evaluate } \left[E\right] \text{ sto}.\)

Every equation for \(\text{execute}\) needs to be altered.

Elaborating a Denotational Definition

- \(\text{program sample is}\)
  - \(\text{var sum, num : integer;}\)
  - \(\text{begin}\)
    - \(\text{sum := 0; read num;}\)
    - \(\text{while num} \geq 0 \text{ do}\)
      - \(\text{if num} > 9 \text{ and num} < 100\)
        - \(\text{then sum := sum + num}\)
        - \(\text{end if;}\)
      - \(\text{read num}\)
    - \(\text{end while;}\)
    - \(\text{write sum}\)
  - \(\text{end}\)

Input list = [5, 22, -1]

Abbreviations:

- \(d = \text{ var sum, num : integer}\)
- \(c_1 = \text{ sum := 0}\)
- \(c_2 = \text{ read num}\)
- \(c_3 = \text{ while num} \geq 0 \text{ do } c_{3.1} ; c_{3.2}\)
  - \(c_{3.1} = \text{ if num} > 9 \text{ and num} < 100\)
    - \(\text{then sum := sum + num}\)
  - \(c_{3.2} = \text{ read num}\)
- \(c_4 = \text{ write sum}\)
Meaning of the Program:

meaning [[program sample is d begin c₁ ; c₂ ; c₃ ; c₄ end]] [5,22,-1] = outp
where (sto₀, inp₁, outp) = execute [[c₁ ; c₂ ; c₃ ; c₄]] (emptySto, [5,22,-1], [])

execute [[c₁ ; c₂ ; c₃ ; c₄]] (emptySto, [5,22,-1], []) = (execute [[c₄] ° execute [[c₃]] ° execute [[c₂]] ° execute [[c₁]])

The commands are executed from inside out.

execute [[sum := 0]] (emptySto, [5,22,-1], []) = (updateSto(emptySto, sum, int(0)), [5,22,-1], []) = ({sum | int(0)}, [5,22,-1], [])

We work on the boolean expression first.

evaluate [[num]] sto₀,₅ = applySto(sto₀,₅, num) = int(5)
evaluate [[0]] sto₀,₅ = int(0)

execute [[read num]] ((sum → int(0)), [5,22,-1], []) = (updateSto((sum → int(0)), num, int(5)), [22, -1], [])
= ((sum → int(0), num → int(5)), [22,-1], [])
Let sto₀,₅ = (sum → int(0), num → int(5))

execute [[while num≥0 do c₃₁ ; c₃₂ ]] (sto₀,₅, [22,-1], [])
= loop (sto₀,₅, [22,-1], [])
where loop (sto, in, out) = if p
then loop (execute [[c₃₁ ; c₃₂]] (sto, in, out))
else (sto, in, out)
where bool(p) = evaluate [[num≥0]] sto

We need the value of the boolean expression in the if command next.

evaluate [[num≥9]] sto₀,₅ = if greater(m, n) then bool(true) else bool(false)
where int(m) = evaluate [[num]] sto₀,₅
and int(n) = evaluate [[9]] sto₀,₅
= if greater(5, 9) then bool(true) else bool(false) = bool(true)

evaluate [[num<100]] sto₀,₅ = if less(m, n) then bool(true) else bool(false)
where int(m) = evaluate [[num]] sto₀,₅
and int(n) = evaluate [[100]] sto₀,₅
= if less(5, 100) then bool(true) else bool(false) = bool(true)

evaluate [[num≥9 and num<100]] sto₀,₅
= if p then bool(q) else bool(false)
where bool(p) = evaluate [[num≥9]] sto₀,₅
and bool(q) = evaluate [[num<100]] sto₀,₅
= if false then bool(true) else bool(false) = bool(false)
Continuing with the if command, we get:

```plaintext
evaluate [if num>9 and num<100 then
sum := sum+num] (sto0,22
= if greatereq(m,n) then bool(true)
else bool(false)
where int(m) = ...
] = execute [read num]
(execute [if num>9 and num<100 then
sum := sum+num] (sto0,22, [-1], []))
```

Let \( sto_{0,22} = \{ sum|!int(0), num|!int(22) \} \)

Summarizing the execution of the body of the while command, we have the result.

```plaintext
evaluate [c3.1 : c3.2] (sto0,5, [22,-1], [])
= (sto0,22, [-1], [])
```

This completes the first pass through the loop.

```plaintext
loop (sto0,5, [22,-1], [])
= loop(execute [c3.1 : c3.2] (sto0,5, [22,-1], []))
= loop(sto0,22, [-1], [])
```

Again we work of the boolean expression from the while command first.

```plaintext
evaluate [num] sto0,22
= applySto(sto0,22, num) = int(22)
evaluate [0] sto0,22 = int(0)
```

The boolean expression in the if command must be evaluated again.

```plaintext
evaluate [num>9] sto0,22
= if greater(m,n) then bool(true)
else bool(false)
where int(m) = evaluate [num] sto0,22
and int(n) = evaluate [0] sto0,22
= if greater(22,0) then bool(true)
else bool(false)
= bool(true)
evaluate [num<100] sto0,22
= if less(m,n) then bool(true) else bool(false)
where int(m) = evaluate [num] sto0,22
and int(n) = evaluate [100] sto0,22
= if less(22,100) then bool(true) else bool(false)
= bool(true)
evaluate [num>9 and num<100] sto0,22
= if p then bool(q) else bool(false)
where bool(p) = evaluate [num>9] sto0,5
and bool(q) = evaluate [num<100] sto0,5
= if true then bool(true) else bool(false)
= bool(true)
```
This time we execute the **then** clause in the **if** command.

\[
\text{execute } \begin{cases} \text{if num>9 and num<100 then} \\
\text{sum := sum+num}(\text{sto0,22, [-1], []}) \\
\text{else } (\text{sto0,22, [-1], []}) \\
\end{cases}
\]

where **bool**(p) =
\[
\text{evaluate } \begin{cases} \text{num>9 and num<100} \text{ sto0,5} \\
0 \text{ true then } \text{execute } (\text{sto0,22, [-1], []}) \\
\end{cases}
\]

Now we need the value of the right side of the assignment command.

\[
\text{evaluate } \begin{cases} \text{sum+num} \text{ sto0,22} \\
\text{= int(plus(m,n))} \\
\text{where int}(m) = \text{evaluate } (\text{sum}) \text{ sto0,22} \\
\text{and int}(n) = \text{evaluate } (\text{num}) \text{ sto0,22} \\
\text{= int(plus(0,22)))} = \text{int}(22)
\end{cases}
\]

Completing the assignment provides the state produced by the **if** command.

\[
\text{execute } \begin{cases} \text{if num>9 and num<100 then} \\
\text{sum := sum+num}(\text{sto0,22, [-1], []}) \\
\end{cases}
\]

where **bool**(p) =
\[
\text{evaluate } \begin{cases} \text{num>9 and num<100} \text{ sto0,5} \\
0 \text{ true then } \text{execute } (\text{sto0,22, [-1], []}) \\
\end{cases}
\]

Let sto22,22 = \{(\text{int}(22)),\text{num}→\text{int}(22)\}

Continuing with the body of the **while** command for its second pass yields a state with store sto22,-1 after executing the **read** command.

\[
\text{execute } \begin{cases} \text{read num}(\text{sto22,22, [-1], []}) \\
\text{= (updateSto(sto22,22,num,\text{int}(1)), [], []}) \\
\end{cases}
\]

Let sto22,-1 = \{(\text{int}(22)),\text{num}→\text{int}(1)\}

Summarizing the second execution of the body of the **while** command, we have the result.

\[
\text{execute } \begin{cases} [c3.1 ; c3.2] (\text{sto0,22, [-1], []}) = \\
\text{(sto22,-1, [], []})
\end{cases}
\]

This completes the second pass through loop.

\[
\text{loop (sto0,22, [-1], [])} \\
\text{= loop (execute [c3.1 ; c3.2] (sto0,22, [-1], []))} \\
\text{= loop(sto22,-1, [], []])}
\]

Again we work on the boolean expression from the **while** command first.

\[
\text{evaluate } \begin{cases} \text{num} \text{ sto22,-1 = applySto(sto22,-1, num)} \\
\text{= int(-1)}
\end{cases}
\]

\[
\text{evaluate } [0] \text{ sto22,-1 = int(0)}
\]

\[
\text{evaluate } [\text{num}≥0] \text{ sto22,-1} \\
\text{= if greatereq(m,n) then } \text{bool}(true) \\
\text{else } \text{bool}(false) \\
\text{where int}(m) = \text{evaluate } [\text{num}] \text{ sto22,-1} \\
\text{and int}(n) = \text{evaluate } [0] \text{ sto22,-1} \\
\text{= if greatereq(-1,0) then } \text{bool}(true) \\
\text{else } \text{bool}(false) \\
\text{= bool}(false)
\]

When we execute loop for the third time, we exit the **while** command.

\[
\text{loop (sto22,-1, [], []])} \\
\text{= if false then loop(execute [c3.1 ; c3.2] (sto22,22,-1, [], []])} \\
\text{else (sto22,-1, [], []])}
\]

Recapping the execution of the **while** command, we conclude:

\[
\text{execute } \begin{cases} \text{while num}≥0 \text{ do } [c3.1 ; c3.2] (\text{sto0,5, [22,-1], []}) \\
\text{= loop (sto0,5, [22,-1], []])} \\
\text{= (sto22,-1, [], []])}
\end{cases}
\]
Now we can continue with the fourth command in the program.

\[
evaluate{[\text{sum}]}\sto_{22,-1} = applySto(\sto_{22,-1}, \text{sum}) = \text{int}(22)
\]

\[
execute{[\text{write sum}]}(\sto_{22,-1}, [], []) = (\sto_{22,-1}, [], [22]) = (\sto_{22,-1}, [], \text{affix}([], \text{val}))
\]

\[
\text{where int}(\text{val}) = evaluate{[\text{sum}]}\sto_{22,-1} = (\sto_{22,-1}, [], [22])
\]

Finally, we summarize the execution of the four commands to obtain the meaning of the program.

\[
execute{[C_1; C_2; C_3; C_4]}(\text{emptySto}, [5,22,-1], []) = (\sto_{22,-1}, [], [22])
\]

meaning \[
\text{program sample is d}
\begin{align*}
\text{begin} & \quad C_1; C_2; C_3; C_4 \\
\text{end} & \quad [5,22,-1] \\
& \quad = [22]
\end{align*}
\]

---

### Implementing Denotational Semantics

Semantic functions become Prolog predicates.

\[
execute : \text{Command} \rightarrow \text{Store} \rightarrow \text{Store}
\]

becomes the predicate

\[
execute(Cmd, Sto, NewSto).
\]

Semantic equations become clauses.

#### Command Sequencing

\[
execute{[C_1; C_2]} = execute{[C_2]} \circ execute{[C_1]}
\]

becomes

\[
execute([Cmd|Cmds], Sto, NewSto) :- execute(Cmd, Sto, TempSto), execute(Cmds, TempSto, NewSto).
\]

\[
execute([\text{ }], Sto, Sto).
\]

---

### If Command

\[
execute{[\text{if E then } C_1 \text{ else } C_2]} \sto =
\begin{align*}
& \text{if p then } execute{[C_1]} \sto \text{ else } execute{[C_2]} \sto \\
& \text{where bool}(p) = evaluate{[E]} \sto
\end{align*}
\]

becomes

\[
execute(\text{if}(\text{Test}, \text{Then}, \text{Else}), \sto, \text{NewSto}) :-
\begin{align*}
& \text{evaluate}(\text{Test}, \sto, \text{Val}), \\
& \text{branch}(\text{Val}, \text{Then}, \text{Else}, \sto, \text{NewSto}).
\end{align*}
\]

\[
\text{branch}(\text{bool(true)}, \text{Then}, \text{Else}, \sto, \text{NewSto}) :-
\begin{align*}
& \text{execute}(\text{Then}, \sto, \text{NewSto}).
\end{align*}
\]

\[
\text{branch}(\text{bool(false)}, \text{Then}, \text{Else}, \sto, \text{NewSto}) :-
\begin{align*}
& \text{execute}(\text{Else}, \sto, \text{NewSto}).
\end{align*}
\]

---

### Modeling the Store

The store

\[
\{ \text{a} \mapsto \text{int}(3), \text{b} \mapsto \text{int}(8), \text{c} \mapsto \text{bool(false)} \}
\]

is represented by the Prolog structure

\[
sto(a, \text{int}(3), sto(b, \text{int}(8), sto(c, \text{bool(false), nil}))).
\]

Empty store: Prolog atom “nil”.

#### Auxiliary Functions

\[
\text{applySto}(sto(Ide, Val, Sto), Ide, Val).
\]

\[
\text{applySto}(sto(I, V, Sto), Ide, Val) :-
\begin{align*}
& \text{applySto}(\text{Sto}, \text{Id}, \text{Val}).
\end{align*}
\]

\[
\text{applySto}((\text{nil}, \text{Id}, \text{undefined}) :-
\begin{align*}
& \text{write('Undefined variable')}, \text{nl}, \text{abort}.
\end{align*}
\]

\[
\text{applySto}((\text{nil}, \text{Id}, \text{undefined}) :-
\begin{align*}
& \text{write('Undefined variable')}, \text{nl}, \text{abort}.
\end{align*}
\]
Expressions

```prolog
evaluate : Expression → Store → EV
```

does not correspond to any added annotations.

```
becomes

evaluate(ide(Ide),Sto,Val) :- applySto(Sto, Ide, Val).
evaluate(num(N),Sto,int(N)).
evaluate(true,Sto,bool(true)).
evaluate(false,Sto,bool(false)).
evaluate(minus(E),Sto,int(N)) :- evaluate(E,Sto,Val), Val=int(M), N is -M.
evaluate(bnot(E),Sto,NotE) :- evaluate(E,Sto,Val), negate(Val,NotE).
negate(bool(true),bool(false)).
negate(bool(false),bool(true)).
evaluate(exp(Opr,E1,E2),Sto,Val) :- evaluate(E1,Sto,V1), evaluate(E2,Sto,V2), compute(Opr,V1,V2,Val).
```

Input and Output

Two Approaches

- Nondenotational approach:
  Handle input and output interactively as a program is being interpreted.

```prolog
execute(read(Ide),Sto,NewSto) :-
  write('Input: '), nl, readnum(N),
  updateSto(Sto,Ide,int(N),NewSto).
```

- Denotational approach:
  Use input and output lists and a state structure:

```prolog
state(Sto,Inp,Outp).
```

Most clauses will have to be altered. See text for `read` and `write`.

Compute

```prolog
compute(plus,int(M),int(N),int(R)) :- R is M+N.
compute(divides,int(M),int(0),int(0)) :-
  write('Division by zero'), nl, abort.
compute(divides,int(M),int(N),int(R)) :- R is M//N.
compute(equal,int(M),int(N),bool(true)) :- M =:= N.
compute(equal,int(M),int(N),bool(false)).
compute(neq,int(M),int(N),bool(false)) :- M =:= N.
compute(neq,int(M),int(N),bool(true)).
compute(less,int(M),int(N),bool(true)) :- M < N.
compute(less,int(M),int(N),bool(false)).
compute(and,boo(true),bool(true),bool(true)).
compute(and,boo(P),bool(Q),bool(false)).
```
Meaning of a Program

Without input and output or with interactive IO:

\[
\text{meaning}((\text{Dec}, \text{Cmd}), \text{Sto}) : \negaleq \\
\text{execute}(\text{Cmd}, \text{nil}, \text{Sto}).
\]

Let the “go” predicate print the results:

\[
..., \text{write}('\text{Final Store:}'), \text{nl}, \text{printSto}(\text{Sto}).
\]

With denotational input and output:

\[
\text{meaning}((\text{Dec}, \text{Cmd}), \text{In}, \text{Out}) : \negaleq \\
\text{execute}(\text{Cmd}, \text{state}(\text{nil}, \text{In}, [ ]), \text{state}(\text{Sto}, \text{In1}, \text{Out})).
\]

where

“prog(Dec, Cmd)” is the abstract syntax tree created by the parser,

“In” is the Prolog input list read initially,

“Sto” is the final store, and

“Out” is the resulting output list.

Try It:

\[
\text{cp} ~\text{slonnegr}\text{/public/plf/ds}.
\]
\[
\text{cp} ~\text{slonnegr}\text{/public/plf/dsd}.
\]

Denotational Semantics
with Environments

Features of Pelican

1. A program may consist of several scopes corresponding to the syntactic domain Block that occurs:
   * as the main program,
   * as anonymous blocks \texttt{(declare)}, and
   * in procedures.

2. Each block may contain constant declarations indicated by \texttt{const} as well as variable declarations.

3. Pelican permits the declaration of procedures with zero and one value parameter and commands that invoke these procedures.

Abstract Syntax of Pelican

Abstract Syntactic Domains

| P : Program | L : Identifier^+ | N : Numerical |
| B : Block   | C : Cmd          | E : Expr     |
| D : Dec     | O : Operator     | I : Ident    |
| T : Type    |                  |              |

Abstract Production Rules

Program ::= \textit{program} Ident is Block

Block ::= Dec \textit{begin} Cmd \textit{end}

Dec ::= Dec Dec \texttt{I} ε

\[
\texttt{const} \text{Ident} = \text{Expr}
\]

\[
\texttt{var} \text{Ident} : \text{Type}
\]

\[
\texttt{var} \text{Ident Ident^+} : \text{Type}
\]

\[
\texttt{procedure} \text{Ident is Block}
\]

\[
\text{Type ::= integer I boolean}
\]

Cmd ::= Cmd ; Cmd

\[
| \text{Ident} ::= \text{Expr}
| \text{skip}
| \text{if} \text{Expr} \text{then} \text{Cmd} \text{else} \text{Cmd}
| \text{if} \text{Expr} \text{then} \text{Cmd}
| \text{while} \text{Expr} \text{do} \text{Cmd}
| \text{declare} \text{Block}
| \text{ident}
| \text{ident (Expr)}
\]

Expr ::= Numerical I Ident I \texttt{true} I \texttt{false} I \texttt{~} \text{Expr}

\[
| \text{Expr Operator} \text{Expr} \text{not(Expr)}
\]

Operator ::= + I \texttt{~} I * I / I or I and

\[
| \leq I < I = I > I >= I <=
\]

Note: Abstract syntax is designed to make the definition of the semantic equations easier.
Pelican Program

程序 primefacs

```pascal
program primefacs is
  var num : integer;
  const two = 2;
  procedure pf (d : integer) is
    var q : integer;
    begin
      if num>1 then
        q := num/d;
        if num=d*q then
          write d; num:=q; pf(d)
        else
          pf(d+1)
        end
      end if
    end if
  end pf;
  begin
    read num
    pf(two)
  end;

Input an integer: 9100
Output = 2
Output = 2
Output = 5
Output = 5
Output = 7
Output = 13
yes
```

Semantic Domains

- Integer = \{ ... , -2, -1, 0, 1, 2, 3, 4, ... \}
- Boolean = \{ true, false \}

```haskell
EV = int(Integer) + bool(Boolean)
SV = int(Integer) + bool(Boolean)
```

Denotable Values

- DV = EV + var(Location) + Procedure

Location = Natural Number = \{ 0, 1, 2, 3, 4, ... \}

```haskell
Store = Location \rightarrow SV + unused + undefined
Environment = Identifier \rightarrow DV + unbound
Procedure = proc0(Store \rightarrow Store)
           + proc1(Location \rightarrow Store \rightarrow Store)
```

Environments

Sets of bindings of identifiers to denotable values.

In Pelican:

- DV = int(Integer) + bool(Boolean) + var(Location) + proc0(Store \rightarrow Store)
  + proc1(Location \rightarrow Store \rightarrow Store)

Operations on Environments:

- emptyEnv : Env
  \forall I : Identifier, emptyEnv I = unbound

- extendEnv : Env \times Identifier \times DV \rightarrow Env
  \forall X : Identifier, extendEnv(env,I,dval) X =
  if X = I then dval else env(X)

- applyEnv : Env \times Identifier \rightarrow DV + unused
  applyEnv(env,I) = env(I)

Stores

Store = Location \rightarrow SV + unused + undefined

Operations on Stores:

- emptySto : Store
  \forall loc : Location, emptySto loc = unused

- updateSto : Store \times Location \times (SV + undefined + unused) \rightarrow Store
  \forall X : Location, updateSto(sto,loc,val) X =
  if X = loc then val else sto(X)

- applySto : Store \times Location \rightarrow SV + undefined + unused
  applySto(sto,loc) = sto(loc)
allocate : Store → Store x Location
allocate sto = (updateSto(sto,loc,undefined),loc)
where loc = minimum { k | sto(k) = unused }

deallocate : Store x Location → Store
deallocate(sto,loc) = updateSto(sto,loc,unused)

Semantic Functions

meaning : Program → Store
perform : Block → Env → Store → Store
elaborate : Dec → Env → Store → Env x Store
evaluate : Expr → Env → Store → EV
value : Numeral → EV

Semantic Equations

meaning [program I is B] =
perform [B] emptyEnv emptySto

perform [D begin C end] env sto =
evaluate [C] env1 sto1
where (env1, sto1) = elaborate [D] env sto

elaborate [D1 D2] env sto =
elaborate [D2] env1 sto1
where (env1, sto1) = elaborate [D1] env sto

elaborate [I] env sto = (env,sto)
elaborate [const I = E] env sto =
(extendEnv(env,I,evaluate [E] env sto),sto)

elaborate [var I : T] env sto =
(extendEnv(env,I,var(loc)),sto1)
where (sto1,loc) = allocate sto

elaborate [var I L : T] env sto =
elaborate [var L : T] env1 sto1
where (env1,sto1) =
elaborate [var I : T] env sto

execute [C1 ; C2] env sto =
exectue [C2] env (execute [C1] env sto)
or
execute [C1 ; C2] env =
(execute [C2] env) o (execute [C1] env)

execute [skip] env sto = sto

execute [I := E] env sto =
updateSto(sto,loc,(evaluate [E] env sto))
where var(loc) = applyEnv(env,I)

execute [if E then C] env sto =
if p then execute [C] env sto else sto
where bool(p) = evaluate [E] env sto

execute [if E then C1 else C2] env sto =
if p then execute [C1] env sto else execute [C2] env sto
where bool(p) = evaluate [E] env sto

execute [while E do C] = loop
where loop env sto =
if p then loop env (execute [C] env sto) else sto
where bool(p) = evaluate [E] env sto

execute [declare B] env sto =
perform [B] env sto
Since programs submitted for semantic analysis are assumed syntactically correct, no need to check:

- All identifiers used are bound to the right kind of denotalable values, so dval ≠ unbound and dval is not a procedure.
- Identifiers are of the appropriate type.

Still need to determine:

- Whether an identifier in an expression represents a constant or a variable
- Whether the location bound to a variable identifier has a value when it is accessed.

\[ \text{evaluate } [I] \text{ env sto } = \]
\[ \begin{align*}
& \text{if dval = } \text{int}(n) \text{ or dval = } \text{bool}(p) \\
& \quad \text{then dval} \\
& \quad \text{else if dval = } \text{var}(\text{loc}) \\
& \quad \quad \text{then if } \text{applySto}(	ext{sto, loc}) = \text{undefined} \\
& \quad \quad \quad \text{then error} \\
& \quad \quad \quad \text{else } \text{applySto}(	ext{sto, loc}) \\
& \quad \text{where dval = } \text{applyEnv}(	ext{env, I})
\end{align*} \]

Procedures

\[ \text{elaborate } \begin{cases} \text{procedure } I \text{ is } B \end{cases} \text{ env sto } = \]
\[ \begin{cases} \\ (\text{env}_1, \text{sto}) \\
\text{where } \text{env}_1 = \text{extendEnv}(\text{env, I, proc}0(\text{proc})) \\
\text{and proc} = \text{perform } [B] \text{ env}_1 \\
\end{cases} \]

\[ \text{elaborate } \begin{cases} \text{procedure } I_1(I_2 : T) \text{ is } B \end{cases} \text{ env sto } = \]
\[ \begin{cases} \\ (\text{env}_1, \text{sto}) \\
\text{where } \text{env}_1 = \text{extendEnv}(\text{env, I}_1, \text{proc}1(\text{proc})) \\
\text{and proc loc} = \text{perform } [B] \text{ extendEnv}(\text{env}_1, I_2, \text{var}(\text{loc})) \\
\end{cases} \]

1. Since a procedure object carries along the environment in effect at its definition, an extension of "env", we get static scoping.

That means nonlocal variables in the procedure will refer to variables in the scope of the declaration, not in the scope of the call of the procedure (dynamic scoping).

2. Since the environment "env1" inserted into the procedure object contains the binding of the procedure identifier with this object, recursive references to the procedure are permitted.

If recursion is forbidden, the procedure object can be defined by:

\[ \text{proc} = \text{perform } [B] \text{ env} \]
Procedure Calls

execute \([I]\) env sto = proc sto

where \(proc0(pro) = applyEnv(env,I)\)

execute \([I(E)]\) env sto =

proc loc

updateSto(sto1,loc,evaluate \([E]\) env sto)

where \(proc1(pro) = applyEnv(env,I)\)

and \((sto1,loc) = allocate\ sto\)

---

Example

program prfacs is

var n : integer;

begin

if n>1

q := n/d;

if n=d*q

pf(d+1)

end if

end;

begin n := 20; pf(2) end

var q : integer; {0 |!int(1),13 |!int(5),14 |!undef}

if n>1 is false causing termination.

---

Store

<table>
<thead>
<tr>
<th>var n : integer; {0 \mapsto undefined}</th>
</tr>
</thead>
<tbody>
<tr>
<td>n := 20; {0 \mapsto int(20)}</td>
</tr>
<tr>
<td>pf(2)</td>
</tr>
<tr>
<td>(d : integer) {0 \mapsto int(20),1 \mapsto int(2)}</td>
</tr>
<tr>
<td>var q : integer; {0 \mapsto int(20),1 \mapsto int(2),2 \mapsto undefined}</td>
</tr>
<tr>
<td>if n&gt;1</td>
</tr>
<tr>
<td>if n=d*q</td>
</tr>
<tr>
<td>pf(d)</td>
</tr>
<tr>
<td>(d : integer) {0 \mapsto int(10),3 \mapsto int(2)}</td>
</tr>
<tr>
<td>var q : integer; {0 \mapsto int(10),3 \mapsto int(2),4 \mapsto undefined}</td>
</tr>
<tr>
<td>if n&gt;1</td>
</tr>
<tr>
<td>if n=d*q</td>
</tr>
<tr>
<td>pf(d)</td>
</tr>
<tr>
<td>(d : integer) {0 \mapsto int(5),5 \mapsto int(2)}</td>
</tr>
<tr>
<td>var q : integer; {0 \mapsto int(5),5 \mapsto int(2),6 \mapsto undefined}</td>
</tr>
<tr>
<td>if n&gt;1</td>
</tr>
<tr>
<td>if n=d*q</td>
</tr>
<tr>
<td>pf(d+1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment</th>
<th>[n \mapsto var(0)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>env1</td>
<td>env2L</td>
</tr>
<tr>
<td>env1 = [pf \mapsto pf1(proc), n \mapsto var(0)]</td>
<td></td>
</tr>
<tr>
<td>env2L = [d \mapsto var(L), q \mapsto var(L+1),</td>
<td></td>
</tr>
<tr>
<td>pf \mapsto pf1(proc), n \mapsto var(0)]</td>
<td></td>
</tr>
<tr>
<td>for L = 1,3,5,7,9,11,13</td>
<td></td>
</tr>
</tbody>
</table>
Checking Context Constraints

Modify Pelican
- No procedures
- Include read and write

Denotational Semantics
- No need for a store
- Environments record types

Semantic Domains
- Boolean = { true, false }
- Sort = { integer, boolean, intvar, boolvar, program, unbound }
- Environment = Identifier → Sort

Context Conditions for Pelican
1. The program name identifier lies in a scope outside the main block.
2. All identifiers that appear in a block must be declared in that block or in an enclosing block.
3. No identifier may be declared more than once at the top level of a block.
4. The identifier on the left side of an assignment command must be declared as a variable, and the expression on the right side must be of the same type.
5. An identifier occurring as an (integer) element must be an integer variable or an integer constant.
6. An identifier occurring as a Boolean element must be a Boolean variable or a Boolean constant.
7. An identifier occurring in a read command must be an integer variable.
8. An identifier used in a procedure call must be defined in a procedure declaration with the proper number of parameters.
9. The identifier defined as the formal parameter in a procedure declaration is considered to belong to the top-level declarations of the block that forms the body of the procedure.
10. The expression in a procedure call must match the type of the formal parameter in the procedure’s declaration.

Semantic Functions
- validate : Program → Boolean
- examine : Block → Env → Boolean
- elaborate : Dec → (Env × Env) → (Env × Env)
- check : Cmd → Env → Boolean
- typify : Expr → Env → Sort

where Sort =
{ integer, boolean, intvar, boolvar, program, unbound }

A program P satisfies its context constraints if
validate [P] = true
and fails to satisfy them if
validate [P] = false
or
validate [P] = error

Two environments to elaborate each block:
1. One environment (locenv) holds the identifiers local to the block so that duplicate identifier declarations can be detected. It begins the block as an empty environment with no bindings.
2. The other environment (env) collects the accumulated bindings from all of the enclosing blocks. This environment is required so that the expressions in constant declarations can be typified.

Both type environments are built in the same way by adding a new binding using extendEnv as each declaration is elaborated.

The semantic equations show that each time a block is initialized, we build a local type environment starting with the empty environment.

The first equation indicates that the program identifier is viewed as lying in a block of its own, and so it does not conflict with any other occurrences of identifiers.
Semantic Equations

**validate** \([\text{program I is B}]\) =

**examine** \([\text{B}]\) extendEnv(emptyEnv,I,program)

**examine** \([\text{D begin C end}]\) env =

check \([\text{C}]\) env1,

where (locenv1, env1) =

elaborate \([\text{D}]\) (emptyEnv, env)

**elaborate** \([\text{D} \_ D2]\) =

(\(\text{elaborate} [\text{D2}]\)(\(\text{elaborate} [\text{D1}]\))

**elaborate** \([\text{c}]\) (locenv, env) = (locenv, env)

**elaborate** \([\text{const I = E}]\) (locenv, env) =

if applyEnv(locenv, I) = unbound

then (extendEnv(locenv, I, typify \([\text{E}]\) env),

\(\text{extendEnv}(\text{env}, \text{I}, \text{typify}[\text{E}]\text{env})\))

else error

**elaborate** \([\text{var I : T}]\) (locenv, env) =

if applyEnv(locenv, I) = unbound

then (extendEnv(locenv, I, type(T)),

\(\text{extendEnv}(\text{env}, \text{I}, \text{type(T)})\),

else error

**elaborate** \([\text{var I L : T}]\) =

(\(\text{elaborate}[\text{var L : T}]\)) ◦ (\(\text{elaborate}[\text{var I : T}]\))

**check** \([\text{C1 ; C2}]\) env =

(\(\text{check}[\text{C1}]\) env) and (\(\text{check}[\text{C2}]\) env)

**check** \([\text{skip}]\) env = true

**check** \([\text{if E then C}]\) env =

(\(\text{typify}[\text{E}]\) env = boolean) and (\(\text{check}[\text{C}]\) env)

**check** \([\text{if E then C1 else C2}]\) env =

(\(\text{typify}[\text{E}]\) env = boolean) and

(\(\text{check}[\text{C1}]\) env) and (\(\text{check}[\text{C2}]\) env)

**check** \([\text{while E do C}]\) env =

(\(\text{typify}[\text{E}]\) env = boolean) and (\(\text{check}[\text{C}]\) env)

**check** \([\text{declare B}]\) env = examine \([\text{B}]\) env

**check** \([\text{read I}]\) env = (\(\text{applyEnv}(\text{env}, \text{I}) = \text{intvar}\))

**check** \([\text{write E}]\) env = (\(\text{typify}[\text{E}]\) env = integer)

**typify** \([\text{N}]\) env = integer

**typify** \([\text{true}]\) env = boolean

**typify** \([\text{false}]\) env = boolean

**typify** \([\text{E1 + E2}]\) env =

if (\(\text{typify}[\text{E1}]\) env = integer) and (\(\text{typify}[\text{E2}]\) env = integer)

then integer else error

\(\vdots\)

**typify** \([\text{E1 and E2}]\) env =

if (\(\text{typify}[\text{E1}]\) env = boolean) and (\(\text{typify}[\text{E2}]\) env = boolean)

then boolean else error

\(\vdots\)

**typify** \([\text{E1 < E2}]\) env =

if (\(\text{typify}[\text{E1}]\) env = integer) and (\(\text{typify}[\text{E2}]\) env = integer)

then boolean else error

\(\vdots\)
Continuation Semantics

Limitations of direct (denotational) semantics:
1. Errors must be propagated through all of the semantic functions cluttering the definitions and making them less realistic.
2. It is very difficult to model sequencers:
   - goto, stop, return, exit, break, continue, raise, and resume.

Example:
```
begin L_1 : C_1;  L_2 : C_2;  L_3 : C_3;  L_4 : C_4 end
```

Meaning with Direct Semantics:
```
execute [C_4] \circ execute [C_3] \circ execute [C_2] \circ execute [C_1]
```

Store Transformation:
```
st_{0} \rightarrow execute [C_1] \rightarrow execute [C_2] 
\rightarrow execute [C_3] \rightarrow execute [C_4] \rightarrow st_{\text{final}}.
```

Continuations

Semantic Domain:
```
\text{Continuation} = \text{Store} \rightarrow \text{Store}
```

A continuation models the remainder of the program from a point in the code. Labels are bound to continuations in the environment.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Denotable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_1</td>
<td>cont_1 = execute [C_1; C_2; C_3; C_4] env</td>
</tr>
<tr>
<td>L_2</td>
<td>cont_2 = execute [C_2; C_3; C_4] env</td>
</tr>
<tr>
<td>L_3</td>
<td>cont_3 = execute [C_3; C_4] env</td>
</tr>
<tr>
<td>L_4</td>
<td>cont_4 = execute [C_4] env</td>
</tr>
</tbody>
</table>

Continuations depend on the current environment so that labels are accessible for jumps to be performed.

Therefore, env must contain the bindings for L_1, L_2, L_3, and L_4.

What if “C_3” is “if x>0 then goto L_1 else skip”?

Store Transformation if x>0:
```
st_{0} \rightarrow execute [C_1] \rightarrow execute [C_2] 
\rightarrow execute [C_3] \rightarrow execute [C_1] \rightarrow etc.
```

“execute [C_3]” needs to be able to make a choice of where to send its resulting store:
- if x>0, send store to “execute [C_1]”
- if x\leq0, send store to “execute [C_4]”

Meaning of Labels:
For k=1, 2, 3, or 4,
“L_k” denotes the computation starting with the command “C_k” and running to the termination of the program.
Encapsulate this meaning as a function from the current store to a final store for the entire program.
A continuation.

Example (Falsely rejected by version in text)
The program identifier “bug” is ignored to save space.
```
program bug is
const c = 5;  [ c \mapsto \text{int} ]  [ c \mapsto \text{int} ]
var k : \text{integer};  [ k \mapsto \text{ivar}, c \mapsto \text{int} ][ k \mapsto \text{ivar}, c \mapsto \text{int} ]
begin
k := 99;  [ k \mapsto \text{ivar}, c \mapsto \text{int} ]
declare
const d = c+k;  [ d \mapsto \text{int} ]
[ d \mapsto \text{int}, k \mapsto \text{ivar}, c \mapsto \text{int} ]
var m : \text{integer};  [ m \mapsto \text{ivar}, d \mapsto \text{int} ]
[ m \mapsto \text{ivar}, d \mapsto \text{int}, k \mapsto \text{ivar}, c \mapsto \text{int} ]
begin
m := c+d+k;  [ m \mapsto \text{ivar}, d \mapsto \text{int}, k \mapsto \text{ivar}, c \mapsto \text{int} ]
write m  [ m \mapsto \text{ivar}, d \mapsto \text{int}, k \mapsto \text{ivar}, c \mapsto \text{int} ]
end
end
```
Executing Commands

execute : Cmd → Env → Continuation → Store → Store

Executing a command requires:
• The environment to determine the target of jumps.
• The current continuation if computation proceeds to the next command.

execute [C_1 ; C_2] env cont sto =
execute [C_1] env (execute [C_2] env cont) sto

execute [goto L] env cont sto = applyEnv(env,L) sto

execute [skip] env cont sto = cont sto

Gull Programming Language

• Integer variables only.
• No if-then command.
• An anonymous block called a Series.
• A Series provides a scoping region for labels.
• A Series is a Command with its own environment.
• Additional context constraints:
  1. No duplicate labels in a Series.
  2. No jump to an undefined label.

Abstract Syntax

Syntactic Domains:

P : Program L : Label O : Operator
S : Series I : Identifier N : Numeral
C : Command E : Expression

Semantic Domains

EV = int(Integer) + bool(Boolean)
SV = int(Integer)
Store = Identifier → SV + undefined
Continuation = Store → Store
Env = Label → Continuation + unbound

Semantic Functions

meaning : Program → Store
perform : Series → Env → Continuation → Store → Store
execute : Command → Env → Continuation → Store → Store
evaluate : Expr → Store → EV
Auxiliary Functions

emptySto : Store
updateSto : Store x Identifier x SV → Store
applySto : Store x Identifier → SV
emptyEnv : Env
extendEnv : Env x Label+ x Continuation+ → Env
applyEnv : Env x Label → Continuation
identityCont : Continuation

∀sto : Store, identityCont sto = sto
extendEnv handles lists of identifiers and continuations (of the same length).

Semantic Equations

meaning [[program I is begin S end]] =
perform [[S]] emptyEnv identityCont emptySto
perform [[L1 : C1 ; L2 : C2 ; ... ; Ln : Cn]] env cont sto = cont1
where cont1 = execute [[C1]] env1 cont2
cont2 = execute [[C2]] env1 cont3
... contn = execute [[Cn]] env1 cont
and env1 = extendEnv(env, [L1, ..., Ln], [cont1, ..., contn])

execute [[I := E]] env cont sto =
cont updateSto(sto, I, evaluate [[E]] sto)
execute [[skip]] env cont sto = cont sto
execute [[stop]] env cont sto = sto
execute [[if E then S1 else S2]] env cont sto =
if p then perform [[S1]] env cont sto
else perform [[S2]] env cont sto
where bool(p) = evaluate [[E]] sto

Error Continuation

Need expression continuations to treat errors properly.

Scheme (a version of Lisp) has expression continuations as first-class objects.

Without expression continuations, we need to test the results of expressions.

Assignment Command
execute [[I := E]] env cont sto =
if evaluate [[E]] sto = error
then errCont sto
else cont updateSto(sto, I, evaluate [[E]] sto)

If Command
execute [[if E then S1 else S2]] env cont sto =
if evaluate [[E]] sto = error
then errCont sto
else if p
then perform [[S1]] env cont sto
else perform [[S2]] env cont sto
where bool(p) = evaluate [[E]] sto