**Denotational Semantics**

**Basic Idea**
Map syntactic objects into domains of mathematical objects.

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

**Example**
\[
\text{meaning} \left[ \frac{26}{2} \right] = \text{meaning} \left[ (10+3) \right] \\
= \text{meaning} \left[ 013 \right] = \text{meaning} \left[ 13 \right] = 13.
\]

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:
- C : Command
- E : Expression
- N : Numeral
- I : Identifier
- O : Operator.

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 ::= 
  \[ \text{while} \text{Expression} \text{ do Command}^+ \]

E ::= N | I | E O E | \text{–E}

use E ::= N | I | E_1 O E_2 | \text{–E}_1 

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 \rightarrow Integer)
Consists of sets of bindings (functions) of variables to integers.

A \rightarrow B denotes the set of functions with domain A and codomain B.
Semantic functions
Connection between Syntax and Semantics
Map objects of the syntactic world into objects in the semantic world.

Specifying semantic functions
Signatures

- \textit{meaning} : Program $\rightarrow$ Store
- \textit{evaluate} : Expression $\rightarrow$ (Store $\rightarrow$ Value)

Semantic equations
Define how the functions act on each pattern in the syntactic definition of the language.

Example
\begin{align*}
\text{evaluate } [E_1 \ast E_2] \ sto &= \text{times}(\text{evaluate } [E_1] \ sto, \ \text{evaluate } [E_2] \ sto) \\
\text{The value of an expression } "E_1 \ast E_2" \text{ is the mathematical product of the values of its component subexpressions.}
\end{align*}

Auxiliary Functions

- \textit{plus} : Number x Number $\rightarrow$ Number
- \textit{minus} : Number x Number $\rightarrow$ Number
- \textit{times} : Number x Number $\rightarrow$ Number

Describe operations in the semantic domains.

Improve readability of denotational definitions.

Language of Numerals

Syntactic Domains

- \textbf{N} : Numeral -- nonnegative numerals
- \textbf{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

- \textit{value} : Numeral $\rightarrow$ Number
- \textit{digit} : Digit $\rightarrow$ Number

Semantic Equations

\begin{align*}
\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
\end{align*}

Denotational Evaluation

\begin{align*}
\text{value } [905] &= \text{plus}(\text{times}(10, \text{value } [90]), \text{digit } [5]) \\
&= \text{plus}(\text{times}(10, \\
&\quad \text{plus}(\text{times}(10, \text{value } [9]), \\
&\quad \text{digit } [0])), 5) \\
&= \text{plus}(\text{times}(10, \\
&\quad \text{plus}(\text{times}(10, \text{digit } [9]), 0)), 5) \\
&= \text{plus}(\text{times}(10, \\
&\quad \text{plus}(\text{times}(10,9), 0)), 5) \\
&= \text{plus}(\text{times}(10,90), 5) \\
&= \text{plus}(900, 5) \\
&= 905
\end{align*}
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.

3. Compositionality lends a certain elegance to definitions, since the semantic equations are structured by the syntax of the language.

   This structure allows the individual language constructs to be analyzed and evaluated in relative isolation from other features in the language.

   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.

Concrete Syntax

\[
\begin{align*}
<\text{program}> & ::= <\text{expression sequence}> \\
<\text{expression sequence}> & ::= <\text{expression}> \\
& | <\text{expression}> <\text{expression sequence}> \\
<\text{expression}> & ::= <\text{term}> \\
& | <\text{expression}> <\text{operator}> <\text{term}> \\
& | <\text{expression}> <\text{answer}> \\
& | <\text{expression}> <\text{answer}> +/\ \\
<\text{term}> & ::= <\text{numeral}> | \text{MR} \\
& | \text{Clear} | <\text{term}> +/ \\
<\text{operator}> & ::= + | - | \times \\
<\text{answer}> & ::= \text{M+} | = \\
<\text{numeral}> & ::= <\text{digit}> | <\text{numeral}> <\text{digit}> \\
<\text{digit}> & ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\end{align*}
\]

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.
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  M R
| 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+ MR

Abstract Syntax Tree:

Semantic Domains

Integer = { … , -2, -1, 0, 1, 2, 3, 4, … }
Primitive domain

Operation = { plus, minus, times, nop }
Disjoint union: plus + minus + times + nop

State = Integer x Operation x Integer x Integer
Product domain

Auxiliary Operations (semantics)

plus : Integer x Integer → Integer
minus : Integer x Integer → Integer
times : Integer x Integer → Integer
nop : Integer x Integer → Integer
where nop(a,d) = d
### 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>Initial</td>
<td>0</td>
<td>nop</td>
<td>0</td>
<td>0</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:

- **meaning**: Program → Integer
- **perform**: ExprSequence → (State → State)
- **evaluate**: Expression → (State → State)
- **compute**: Operator → (State → State)
- **calculate**: Answer → (State → State)
- **value**: Numeral → Integer

---

-- uses only nonnegative integers

### Semantic Equations

- \( \text{meaning}[P] = d \)
  where \((a, \text{op}, d, m) = \text{perform}[P](0, \text{nop}, 0, 0)\)

- \( \text{perform}[E \ S] = \text{perform}[S] \circ \text{evaluate}[E] \)

- \( \text{perform}[E] = \text{evaluate}[E] \)

- \( \text{evaluate}[N](a, \text{op}, d, m) = (a, \text{op}, v, m) \)
  where \(v = \text{value}[N]\)

- \( \text{evaluate}[MR](a, \text{op}, d, m) = (a, \text{op}, m, m) \)

- \( \text{evaluate}[\text{Clear}](a, \text{op}, d, m) = (0, \text{nop}, 0, 0) \)

- \( \text{evaluate}[E_1 \ O \ E_2] = \text{evaluate}[E_2] \circ \text{compute}[O] \circ \text{evaluate}[E_1] \)

- \( \text{evaluate}[EA] = \text{calculate}[A] \circ \text{evaluate}[E] \)

- \( \text{compute}[+] (a, \text{op}, d, m) = (\text{op}(a, d), \text{plus}(a, d), m) \)

- \( \text{compute}[-] (a, \text{op}, d, m) = (\text{op}(a, d), \text{minus}(a, d), m) \)

- \( \text{compute}[\times] (a, \text{op}, d, m) = (\text{op}(a, d), \text{times}(a, d), m) \)

- \( \text{calculate}[=] (a, \text{op}, d, m) = (a, \text{nop}, \text{op}(a, d), m) \)

- \( \text{calculate}[M+] (a, \text{op}, d, m) = (a, \text{nop}, \text{plus}(m, v)) \)
  where \(v = \text{op}(a, d)\)

- \( \text{calculate}[*/-] (a, \text{op}, d, m) = (a, \text{op}, \text{minus}(0, d), m) \)

- \( \text{value}[N] = \text{usual denotational definition of nonnegative numerals} \)
Denotational Evaluation

Consider the series of keystrokes:

"8 +/- + 5 x 3 =

Meaning of the sequence given by:

\[
\text{meaning} \ [8 +/- + 5 \times 3 \div] = d \text{ where } (a, \text{op}, d, m) = \text{perform} \ [8 +/- + 5 \times 3 \div] (0, \text{nop}, 0, 0).
\]

The evaluation proceeds:

\[
\text{perform} \ [8 +/- + 5 \times 3 \div] (0, \text{nop}, 0, 0)
\]

\[
= \text{evaluate} \ [8 +/- + 5 \times 3 \div] (0, \text{nop}, 0, 0)
\]

\[
= (-3, \text{nop}, -9, 0)
\]

Therefore \text{meaning} \ [8 +/- + 5 \times 3\div] = -9.

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 \textbf{while} command in Wren.

Ignore input and output in Wren for now.
Abstract Syntax for Wren

Abstract Syntactic Domains

\[ P : \text{Program} \quad C : \text{Cmd} \quad N : \text{Numeral} \]
\[ D : \text{Declaration} \quad E : \text{Expr} \quad I : \text{Identifier} \]
\[ T : \text{Type} \quad O : \text{Operator} \]

Abstract Production Rules

Program ::= \textbf{program} Identifier is \\
\phantom{\textbf{program}}\textbf{Declaration}\ast \textbf{begin} Cmd \textbf{end}

Declaration ::= \textbf{var} Identifier \ast Type \\

Type ::= integer | boolean

Cmd ::= Cmd \ast Cmd | Identifier ::= Expr \\
\phantom{\textbf{Cmd}}\textbf{skip} | \textbf{if} Expr then Cmd \textbf{else} Cmd \\
\phantom{\textbf{Cmd}}\textbf{if} Expr then Cmd | \textbf{while} Expr do Cmd

Expr ::= Numeral | Identifier | true | false \\
\phantom{\textbf{Expr}}\textbf{–} Expr | Expr Operator Expr | \textbf{not}(Expr)

Operator ::= + | – | \ast | / | \textbf{or} | \textbf{and} \\
\phantom{\textbf{Operator}}\textbf{<} | \textbf{<} | \textbf{=} | \textbf{> | } \textbf{=} | \textbf{<} \\

Semantic Domains

Primitive Semantic Domains:

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

Boolean = \{ true, false \}

Compound Semantic Domains:

Product Domains

- Cartesian products, \( A \times B \).
- 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 \textbf{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
  \[ \text{Store} = \text{Identifier} \rightarrow (SV + undefined) \]
- 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.
  \[ \text{sto} = \{ \text{count} \mapsto int(1), \text{total} \mapsto int(0) \} \]
- Assume that \( \text{sto}(I) = \text{undefined} \) for all other identifiers \( I \).
- Let \( \{} \) represent an everywhere undefined store.

Operations on Stores

emptySto : Store
\[ \forall I \in \text{Identifier}, \text{emptySto} I = \text{undefined} \]

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

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

We want $updateSto(sto, I, val)$ to be a Store function as well:

For sto:Store, I:Identifier, val:SV,
$\text{updateSto}(sto, I, val) : \text{Identifier} → (\text{SV} + \text{undefined})$

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

$\forall X : \text{Identifier},$
$updateSto(sto, I, val) X =$
  \begin{align*}
  \text{if } X = I \text{ then } val \text{ else } sto(X)
  \end{align*}$

Expressible Values

- Values that expressions can produce.
- Expressible values in Wren:
  \[ \text{EV} = \text{int}(\text{Integer}) + \text{bool}(\text{Boolean}) \]

Auxiliary Functions

$\text{plus} : \text{Integer} \times \text{Integer} → \text{Integer}$
$\text{minus} : \text{Integer} \times \text{Integer} → \text{Integer}$
$\text{times} : \text{Integer} \times \text{Integer} → \text{Integer}$
$\text{divides} : \text{Integer} \times \text{Integer} → \text{Integer}$
$\text{less} : \text{Integer} \times \text{Integer} → \text{Boolean}$
$\text{lesseq} : \text{Integer} \times \text{Integer} → \text{Boolean}$
$\text{greater} : \text{Integer} \times \text{Integer} → \text{Boolean}$
$\text{greatereq} : \text{Integer} \times \text{Integer} → \text{Boolean}$
$\text{equal} : \text{Integer} \times \text{Integer} → \text{Boolean}$
$\text{neq} : \text{Integer} \times \text{Integer} → \text{Boolean}$

Semantic Functions

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

$\text{meaning} : \text{Program} → \text{Store}$
$\text{execute} : \text{Command} → (\text{Store} → \text{Store})$
$\text{evaluate} : \text{Expression} → (\text{Store} → \text{EV})$
$\text{value} : \text{Numeral} → \text{EV}$

- 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

$\text{meaning}[\text{program} \text{ is } D \ \text{begin} \ C \ \text{end}] =$
$\text{execute}[C] \ \text{emptySto}$

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

$\text{execute}[\text{skip}] \ sto = sto$

$\text{execute}[I := E] \ sto =$
  \[ \text{updateSto}(sto, I, (\text{evaluate}[E] \ sto)) \]

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

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

$\text{execute}[\text{while } E \ \text{do } C] = \text{loop}$
where $\text{loop} \ sto =$
  \begin{align*}
  \text{if } p \text{ then } & \text{loop}(\text{execute}[C] \ sto) \text{ else } sto \\
  \text{where } & \text{bool}(p) = \text{evaluate}[E] \ sto
  \end{align*}$
evaluate $[I]$ sto =
  if val=undefined then error else val
  where val = applySto(sto, I)

evaluate $[N]$ sto = int(value $[N]$)
evaluate $[true]$ sto = bool(true)
evaluate $[false]$ sto = bool(false)
evaluate $[E_1 + E_2]$ sto = int(plus(m,n))
  where int(m) = evaluate $[E_1]$ sto
  and int(n) = evaluate $[E_2]$ sto
nevaluate $[E_1 / E_2]$ sto =
  if n=0 then error else int(divides(m,n))
  where int(m) = evaluate $[E_1]$ sto
  and int(n) = evaluate $[E_2]$ sto

execute $[a := 0; b := 1]$ emptySto
  means
  (execute $[a := 0; b := 1]$) emptySto.

execute : Command → Store → Store
  means
  execute : Command → (Store → Store).

These conventions agree:
execute : Command → Store → Store
execute $[a := 0; b := 1]$ : Store → Store
execute $[a := 0; b := 1]$ emptySto : Store

evaluate $[E_1 and E_2]$ sto =
  if p then bool(q) else bool(false)
  where bool(p) = evaluate $[E_1]$ sto
  and bool(q) = evaluate $[E_2]$ sto
nevaluate $[E_1 or E_2]$ sto =
  if p then bool(true) else bool(q)
  where bool(p) = evaluate $[E_1]$ sto
  and bool(q) = evaluate $[E_2]$ sto

evaluate $[- E]$ sto = int(minus(0,m))
  where int(m) = evaluate $[E]$ sto

evaluate $[not(E)]$ sto =
  if evaluate $[E]$ sto = bool(true)
    then bool(false) else bool(true)

Notational Conventions

• Function application associates to the left.
• “→” associates to the right.
execute $[a := 0; b := 1]$ emptySto
  means
  (execute $[a := 0; b := 1]$) emptySto.
execute : Command → Store → Store
  means
  execute : Command → (Store → Store).

Noncompositional while Definition
execute $[while E do C]$ sto =
  if evaluate $[E]$ sto = bool(true)
    then execute $[while E do C]$(execute $[C]$) sto
  else sto

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

Handling Dynamic Errors

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

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

\[
\text{while } E \text{ do } C \equiv \begin{cases} \text{if } E \text{ then } (C; \text{ while } E \text{ do } C) \text{ else } \text{skip} \\ \text{execute } [\text{while } E \text{ do } C] \text{ sto} \\
\end{cases}
\]

\[
evaluate [E] \text{ sto} = \text{bool}(true) \\
\text{then } \text{loop}_1(\text{execute } [C] \text{ sto}) \\
\text{else } \text{sto}
\]

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

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

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.

\[
\text{Input} = \text{Integer}^* \\
\text{Output} = \text{Integer}^*
\]

Meaning of a program defined in terms of these lists.

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

Commands may change the input and output lists, so

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

where

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

Use auxiliary functions to manipulate lists:

\[
\text{head} : \text{Integer}^* \rightarrow \text{Integer} \\
\text{head} [n_1, n_2, \ldots, n_k] = n_1 \text{ provided } k \geq 1.
\]

\[
\text{tail} : \text{Integer}^* \rightarrow \text{Integer}^* \\
\text{tail} [n_1, n_2, \ldots, n_k] = [n_2, \ldots, n_k] \text{ provided } k \geq 1.
\]

\[
\text{null} : \text{Integer}^* \rightarrow \text{Boolean} \\
\text{null} [n_1, n_2, \ldots, n_k] = (k = 0)
\]

\[
\text{affix} : \text{Integer}^* \times \text{Integer} \rightarrow \text{Integer}^* \\
\text{affix} ([n_1, n_2, \ldots, n_k], m) = [n_1, n_2, \ldots, n_k, m].
\]
New Semantic Equations

meaning \([\text{program} \text{ is } \text{I} \text{ begin } \text{C} \text{ end} ] \) \text{ inp} = \text{ outp}
\quad \text{ where } (\text{sto}, \text{ inp}_1, \text{ outp}) =
\quad \quad \text{ execute } [\text{C}] (\text{emptySto}, \text{ inp}, [ ])

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

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

Every equation for execute needs to be altered.

Elaborating a Denotational Definition

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

Input list = [5,22,-1]

Abbreviations
\quad d = \text{ var } \text{sum, num : integer}
\quad c_1 = \text{ sum := 0}
\quad c_2 = \text{ read num}
\quad c_3 = \text{ while num}\geq 0 \text{ do c_1 ; c_2 ; c_3 ; c_4 }
\quad \quad \text{ if num}\geq 9 \text{ and num}<100
\quad \quad \quad \text{ then sum := sum+num }
\quad \quad \text{ end if}
\quad \quad c_3.1 = \text{ read num}
\quad \quad c_3.2 = \text{ write sum}
\quad c_4 = \text{ write sum}

Meaning of the Program

meaning \([\text{program} \text{ sample } \text{ is } \text{d } \text{ begin c_1 ; c_2 ; c_3 ; c_4 end] [5,22,-1] = \text{ outp} \]
\quad \text{ where } (\text{sto}, \text{ inp}_1, \text{ outp}) =
\quad \quad \text{ execute } [\text{c_1 ; c_2 ; c_3 ; c_4}] (\text{emptySto}, [5,22,-1],[])

execute \([\text{c_1 ; c_2 ; c_3 ; c_4}] (\text{emptySto}, [5,22,-1],[]) =
\quad \quad (\text{execute } [\text{c_4}] \circ \text{execute } [\text{c_3}] \circ 
\quad \quad \quad \text{execute } [\text{c_2}] \circ \text{execute } [\text{c_1}])
\quad \quad (\text{emptySto}, [5,22,-1], [])

The commands are executed from inside out.

execute \([\text{sum := 0]} \) (\text{emptySto}, [5,22,-1], [])
\quad = (\text{updateSto}(\text{emptySto}, \text{sum,}
\quad \quad \quad \text{evaluate } [0] \text{ emptySto}), [5,22,-1], [])
\quad = (\text{updateSto}(\text{emptySto}, \text{sum, int}(0)),
\quad \quad \quad [5,22,-1], [])
\quad = ([\text{sum} \rightarrow \text{int}(0)], [5,22,-1], [])

execute \([\text{read num]} \) ([\text{sum} \rightarrow \text{int}(0)], [5,22,-1], [])
\quad = (\text{updateSto}(\text{sum} \rightarrow \text{int}(0), \text{num, int}(5)),
\quad \quad \quad [22,-1], [])
\quad = ([\text{sum} \rightarrow \text{int}(0), \text{num} \rightarrow \text{int}(5)], [22,-1], [])

Let \text{sto}_{0,5} = [\text{sum} \rightarrow \text{int}(0), \text{num} \rightarrow \text{int}(5)]

execute \([\text{while num}\geq 0 \text{ do c_3.1 ; c_3.2 }]
\quad \quad \text{ (sto}_{0,5}, [22,-1], [])
\quad = \text{ loop (sto}_{0,5}, [22,-1], []
\quad \quad \quad \text{ where loop (sto,in,out) =}
\quad \quad \quad \quad \text{ if p}
\quad \quad \quad \quad \quad \text{ then loop(execute [c_3.1 ; c_3.2] (sto,in,out))}
\quad \quad \quad \quad \quad \text{ else (sto,in,out))}
\quad \quad \quad \text{ where bool(p) = \text{evaluate } [\text{num}\geq 0] \text{ sto}}

We work on the boolean expression first.

evaluate \([\text{num}] \) \text{ sto}_{0,5} = \text{ applySto}(\text{sto}_{0,5}, \text{num}) = \text{ int}(5)

evaluate \([0] \) \text{ sto}_{0,5} = \text{ int}(0)
evaluate \[\text{num} \geq 0\] sto_{0,5} \\
= \text{if } \text{greatereq}(m,n) \text{ then } \text{bool}(true) \text{ else } \text{bool}(false) \\
\text{where } \text{int}(m) = \text{evaluate } \text{num} \text{ sto}_{0,5} \\
\text{and } \text{int}(n) = \text{evaluate } 0 \text{ sto}_{0,5} \\
= \text{if } \text{greatereq}(5,0) \text{ then } \text{bool}(true) \text{ else } \text{bool}(false) \\
= \text{bool}(true)

Now we can execute loop for the first time.

\text{loop}\ (\text{sto}_{0,5}, [22,-1], [])
\text{= if } \text{true} \text{ then } \text{loop}(\text{execute } [c_{3.1}; c_{3.2}] \text{ (sto}_{0,5}, [22,-1], []))
\text{else } (\text{sto}_{0,5}, [22,-1], [])
\text{= loop}(\text{execute } [c_{3.1}; c_{3.2}] \text{ (sto}_{0,5}, [22,-1], []))

To complete the execution of loop, we need to execute the body of the \text{while} command.

\text{execute } [c_{3.1}; c_{3.2}] \text{ (sto}_{0,5}, [22,-1], [])
\text{= execute } \text{read} \text{num}
\text{= execute } \text{if } \text{num} > 9 \text{ and } \text{num} < 100 \\
\text{then } \text{sum} := \text{sum} + \text{num} \text{ (sto}_{0,5}, [22,-1], []))

Summarizing the execution of the body of the \text{while} command, we have the result.

\text{execute } [c_{3.1}; c_{3.2}] \text{ (sto}_{0,5}, [22,-1], [])
\text{= (sto}_{0,22}, [-1], [])

This completes the first pass through the loop.

\text{loop}\ (\text{sto}_{0,5}, [22,-1], [])
\text{= loop}(\text{execute } [c_{3.1}; c_{3.2}] \text{ (sto}_{0,5}, [22,-1], []))
\text{= loop}(\text{sto}_{0,22}, [-1], [])

Again we work of the boolean expression from the \text{while} command first.

\text{evaluate } \text{num} \text{ sto}_{0,22}
\text{= applySto(sto}_{0,22}, \text{num}) = \text{int}(22)
\text{evaluate } 0 \text{ sto}_{0,22} = \text{int}(0)
evaluate \([\text{num}\geq 0]\) \(\text{sto}_{0,22}\)

\[
= \begin{cases} 
  \text{bool}(\text{true}) & \text{if } \text{greatereq}(\text{m},\text{n}) \\
  \text{bool}(\text{false}) & \text{else}
\end{cases}
\]

where \(\text{int}(\text{m}) = \text{evaluate} \ [\text{num}] \ \text{sto}_{0,22}\)

and \(\text{int}(\text{n}) = \text{evaluate} [0] \ \text{sto}_{0,22}\)

\[
= \begin{cases} 
  \text{bool}(\text{true}) & \text{if } \text{greatereq}(22,0) \\
  \text{bool}(\text{false}) & \text{else}
\end{cases}
\]

\(= \text{bool}(\text{true})\)

Now we can execute loop for the second time.

\[
\text{loop}(\text{sto}_{0,22}, [-1], [])
\]

\[
= \begin{cases} 
  \text{loop}(\text{execute} [\text{c}_3.1 ; \text{c}_3.2] (\text{sto}_{0,22}, [-1], [])) & \text{if true} \\
  (\text{sto}_{0,22}, [-1], []) & \text{else}
\end{cases}
\]

Again we execute the body of the while command.

\[
\text{execute} [\text{c}_3.1 ; \text{c}_3.2] (\text{sto}_{0,22}, [-1], [])
\]

\[
= \begin{cases} 
  \text{execute} [\text{read} \ \text{num}] (\text{execute}[\text{if } \text{num}>9 \ \text{and} \ \text{num}<100 \ \text{then} \ \text{sum} := \text{sum}+\text{num}] (\text{sto}_{0,22}, [-1], [])) & \text{if } \text{p} \ \text{then}
\end{cases}
\]

\[
= \begin{cases} 
  \text{sum} := \text{sum}+\text{num} & \text{else}
\end{cases}
\]

where \(\text{bool}(\text{p}) = \text{evaluate} [\text{num}>9 \ \text{and} \ \text{num}<100] \ \text{sto}_{0,5}\)

\[
= \begin{cases} 
  \text{sum} := \text{sum}+\text{num} & \text{if true}
\end{cases}
\]

\(= \text{execute} [\text{sum} := \text{sum}+\text{num}] (\text{sto}_{0,22}, [-1], [])\)

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

\[
\text{evaluate} [\text{sum}+\text{num}] \ \text{sto}_{0,22}
\]

\[
= \text{int}(\text{plus}(\text{m},\text{n}))
\]

\[
= \text{int}(\text{plus}(0,22)) = \text{int}(22)
\]

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

\[
\text{evaluate} [\text{num}>9] \ \text{sto}_{0,22}
\]

\[
= \begin{cases} 
  \text{bool}(\text{true}) & \text{if } \text{less}(\text{m},\text{n}) \\
  \text{bool}(\text{false}) & \text{else}
\end{cases}
\]

where \(\text{int}(\text{m}) = \text{evaluate} [\text{num}] \ \text{sto}_{0,22}\)

and \(\text{int}(\text{n}) = \text{evaluate} [9] \ \text{sto}_{0,22}\)

\[
= \begin{cases} 
  \text{bool}(\text{true}) & \text{if } \text{less}(22,9) \\
  \text{bool}(\text{false}) & \text{else}
\end{cases}
\]

\(= \text{bool}(\text{true})\)

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

\[
\text{execute} [\text{sum} := \text{sum}+\text{num}] (\text{sto}_{0,22}, [-1], [])
\]

\[
= (\text{updateSto}(\text{sto}_{0,22}, \text{sum}, \\
(\text{evaluate} [\text{sum}+\text{num}] \ \text{sto}_{0,22})), [-1], [])
\]

\[
= (\text{updateSto}(\text{sto}_{0,22}, \text{sum}, \text{int}(22)), [-1], [])
\]

\[
= ([\text{sum}] \rightarrow \text{int}(22), \text{num} \rightarrow \text{int}(22)), [-1], []
\]

Let \(\text{sto}_{22,22} = {\text{sum} \rightarrow \text{int}(22), \text{num} \rightarrow \text{int}(22)}\)

Continuing with the body of the while command for its second pass yields a state with store \(\text{sto}_{22,-1}\) after executing the read command.

\[
\text{execute} [\text{read} \ \text{num}] (\text{sto}_{22,22}, [-1], [])
\]

\[
= (\text{updateSto}(\text{sto}_{22,22}, \text{num}, \text{int}(1))), [], []
\]

\[
= ([\text{sum}] \rightarrow \text{int}(22), \text{num} \rightarrow \text{int}(1)), [], []
\]

Let \(\text{sto}_{22,-1} = {\text{sum} \rightarrow \text{int}(22), \text{num} \rightarrow \text{int}(1)}\)

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

\[
\text{execute} [\text{c}_3.1 ; \text{c}_3.2] (\text{sto}_{0,22}, [-1], []) =
\]

\[
(\text{sto}_{22,-1}, [], []).
\]
This completes the second pass through loop.

\[
\text{loop } (\text{sto}_{0,22}, [-1], [])
= \text{loop } (\text{execute } [c_{3.1}; c_{3.2}] (\text{sto}_{0,22}, [-1], []))
= \text{loop } (\text{sto}_{22}, [], []]
\]

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

\[
evaluate [\text{num}] \text{sto}_{22},-1 = \text{applySto}(\text{sto}_{22},-1, \text{num})
= \text{int}(-1)
\]

\[
evaluate [0] \text{sto}_{22},-1 = \text{int}(0)
\]

\[
evaluate [\text{num} \geq 0] \text{sto}_{22},-1
= \text{if } \text{greatereq}(m,n) \text{ then } \text{bool}(true)
\text{ else } \text{bool}(false)
\]

where \(\text{int}(m) = \text{evaluate } [\text{num}] \text{sto}_{22},-1\)
and \(\text{int}(n) = \text{evaluate } [0] \text{sto}_{22},-1\)

\[
= \text{if } \text{greatereq}(-1,0) \text{ then } \text{bool}(true)
\text{ else } \text{bool}(false)
= \text{bool}(false)
\]

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

\[
\text{loop } (\text{sto}_{22},-1, [], [])
= \text{if } \text{false} \text{ then } \text{loop } (\text{execute } [c_{3.1}; c_{3.2}] (\text{sto}_{22},-1, [], []))
\text{ else } (\text{sto}_{22},-1, [], []))
\]

Recapping the execution of the while command, we conclude:

\[
\text{execute } [\text{while} \text{num} \geq 0 \text{ do } c_{3.1}; c_{3.2}] (\text{sto}_{0.5}, [22,-1], [])
= \text{loop } (\text{sto}_{0.5}, [22,-1], [])
= (\text{sto}_{22},-1, [], [])
\]

Now we can continue with the fourth command in the program.

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

\[
\text{execute } [\text{write} \text{sum}] (\text{sto}_{22},-1, [], []))
= (\text{sto}_{22},-1, [], [], \text{affix}([], \text{val}))
\text{ where } \text{int}(\text{val}) = \text{evaluate } [\text{sum}] \text{sto}_{22},-1
= (\text{sto}_{22},-1, [], [22])
\]

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

\[
\text{execute } [c_{1}; c_{2}; c_{3}; c_{4}] (\text{emptySto}, [5,22,-1], [])
= (\text{sto}_{22},-1, [], [22])
\]

\[
\text{meaning } [\text{program} \text{sample is } d \begin{array}{c} \text{begin} \end{array} c_{1}; c_{2}; c_{3}; c_{4} \text{ end} ] [5,22,-1]
= [22]
\]

Implementing Denotational Semantics

Semantic functions become Prolog predicates.

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

becomes the predicate

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

Semantic equations become clauses.

**Command Sequencing**

\[
\text{execute } [c_{1}; c_{2}] = \text{execute } [c_{2}] \circ \text{execute } [c_{1}]
\]

becomes

\[
\text{execute}([\text{Cmd}[\text{Cmds}], \text{Sto}, \text{NewSto}) : -
\text{execute} (\text{Cmd}, \text{Sto}, \text{TempSto}),
\text{execute} (\text{Cmds}, \text{TempSto}, \text{NewSto}).
\text{execute}([], \text{Sto}, \text{Sto}).
\]
If Command

\[
\text{execute } [\text{if } E \text{ then } C_1 \text{ else } C_2] \text{ sto } =
\begin{align*}
\text{if } p \text{ then } \text{execute } [C_1] \text{ sto else execute } [C_2] \text{ sto}
\end{align*}
\]

where \( \text{bool}(p) = \text{evaluate } [E] \text{ sto} \)

becomes

\[
\text{execute}(\text{if}(\text{Test}, \text{Then}, \text{Else}), \text{Sto}, \text{NewSto}) :- \\
\text{evaluate}(\text{Test}, \text{Sto}, \text{Val}), \\
\text{branch}(\text{Val}, \text{Then}, \text{Else}, \text{Sto}, \text{NewSto}).
\]

\[
\text{branch}(\text{bool(true)}, \text{Then}, \text{Else}, \text{Sto}, \text{NewSto}) :- \\
\text{execute}(\text{Then}, \text{Sto}, \text{NewSto}).
\]

\[
\text{branch}(\text{bool(false)}, \text{Then}, \text{Else}, \text{Sto}, \text{NewSto}) :- \\
\text{execute}(\text{Else}, \text{Sto}, \text{NewSto}).
\]

Modeling the Store

The store
\[
\{ a \mapsto \text{int}(3), \ b \mapsto \text{int}(8), \ c \mapsto \text{bool}(\text{false}) \}
\]
is represented by the Prolog structure
\[
\text{sto}(a, \text{int}(3), \text{sto}(b, \text{int}(8), \text{sto}(c, \text{bool}(\text{false}), \text{nil}))).
\]

Empty store: Prolog atom “nil”.

Auxiliary Functions

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

\[
\text{applySto}(\text{sto}(\text{I}, \text{V}, \text{Sto}), \text{Ide}, \text{Val}) :- \\
\text{applySto}(\text{Sto}, \text{Ide}, \text{Val}).
\]

\[
\text{applySto}(\text{nil}, \text{Ide}, \text{undefined}) :- \\
\text{write}('\text{Undefined variable'}), \text{nl}, \text{abort}.
\]

Expressions

\[
\text{evaluate} : \text{Expression} \rightarrow \text{Store} \rightarrow \text{EV}
\]

becomes

\[
\text{evaluate} (\text{ide}(\text{Ide}), \text{Sto}, \text{Val}) :- \\
\text{applySto}(\text{Sto}, \text{Ide}, \text{Val}).
\]

\[
\text{evaluate} (\text{num}(\text{N}), \text{Sto}, \text{int}(\text{N})).
\]

\[
\text{evaluate} (\text{true}, \text{Sto}, \text{bool}(\text{true})).
\]

\[
\text{evaluate} (\text{false}, \text{Sto}, \text{bool}(\text{false})).
\]

\[
\text{evaluate} (\text{minus}(\text{E}), \text{Sto}, \text{int}(\text{N})) :- \\
\text{evaluate}(\text{E}, \text{Sto}, \text{Val}), \text{Val=int(M)}, \text{N is } -\text{M}.
\]

\[
\text{evaluate} (\text{bnot}(\text{E}), \text{Sto}, \text{NotE}) :- \\
\text{evaluate}(\text{E}, \text{Sto}, \text{Val}), \text{negate} (\text{Val}, \text{NotE}).
\]

\[
\text{negate} (\text{bool(true)}, \text{bool(false)}).
\]

\[
\text{negate} (\text{bool(false)}, \text{bool(true)}).
\]

\[
\text{evaluate} (\text{exp}(\text{Opr}, \text{E1}, \text{E2}), \text{Sto}, \text{Val}) :- \\
\text{evaluate}(\text{E1}, \text{Sto}, \text{Val}), \\
\text{evaluate}(\text{E2}, \text{Sto}, \text{Val}), \\
\text{compute} (\text{Opr}, \text{V1}, \text{V2}, \text{Val}).
\]

Assignment Command

\[
\text{execute } [I := E] \text{ sto } = \\
\text{updateSto}(\text{sto}(\text{Ide}, \text{Val}, \text{Sto}), \text{Ide}, \text{Val}).
\]

\[
\text{updateSto}(\text{sto}(\text{I}, \text{V}, \text{Sto}), \text{Ide}, \text{Val}, \text{sto}(\text{I}, \text{V}, \text{NewSto})) :- \\
\text{updateSto}(\text{Sto}, \text{Ide}, \text{Val}, \text{NewSto}).
\]

\[
\text{updateSto}(\text{nil}, \text{Ide}, \text{Val}, \text{sto}(\text{Ide}, \text{Val}, \text{nil})).
\]
**Compute**

\[
\text{compute}(\text{plus}, \text{int}(M), \text{int}(N), \text{int}(R)) \leftarrow R \text{ is } M+N.
\]

\[
\text{compute}(\text{divides}, \text{int}(M), \text{int}(0), \text{int}(0)) \leftarrow
\text{write('Division by zero'), nl, abort.}
\]

\[
\text{compute}(\text{divides}, \text{int}(M), \text{int}(N), \text{int}(R)) \leftarrow R \text{ is } M/N.
\]

\[
\text{compute}(\text{equal}, \text{int}(M), \text{int}(N), \text{bool}(true)) \leftarrow M =:= N.
\]

\[
\text{compute}(\text{equal}, \text{int}(M), \text{int}(N), \text{bool}(false)).
\]

\[
\text{compute}(\text{neq}, \text{int}(M), \text{int}(N), \text{bool}(false)) \leftarrow M =:= N.
\]

\[
\text{compute}(\text{neq}, \text{int}(M), \text{int}(N), \text{bool}(true)).
\]

\[
\text{compute}(\text{less}, \text{int}(M), \text{int}(N), \text{bool}(true)) \leftarrow M < N.
\]

\[
\text{compute}(\text{less}, \text{int}(M), \text{int}(N), \text{bool}(false)).
\]

\[
\text{compute}(\text{and}, \text{bool}(true), \text{bool}(true), \text{bool}(true)).
\]

\[
\text{compute}(\text{and}, \text{bool}(P), \text{bool}(Q), \text{bool}(false)).
\]

**Input and Output**

**Two Approaches**

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

\[
\text{execute}(\text{read(Ide), Sto, NewSto}) \leftarrow
\text{write('Input: '), nl, readnum(N),}
\text{updateSto(Sto, Ide, int(N), NewSto}).
\]

\[
\text{execute}(\text{write(Exp), Sto, Sto}) \leftarrow
\text{evaluate(Exp, Sto, Val), Val=int(M),}
\text{write('Output = '), write(M), nl.}
\]

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

\[
\text{state(Sto, Inp, Outp}).
\]

Most clauses will have to be altered.

See text for read and write.

**Meaning of a Program**

Without input and output or interactive IO:

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

Let the “go” predicate print the results:

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

With denotational input and output:

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

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.

**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 (declare), and
   - in procedures.

2. Each block may contain constant declarations indicated by 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 : Numeral
B : Block    C : Cmd     E : Expr
D : Dec     O : Operator I : Ident
T : Type

Abstract Production Rules

Program ::= program Ident is Block
Block ::= Dec begin Cmd end
Dec ::= Dec Dec | ε

| const Ident = Expr
| var Ident : Type
| var Ident Ident+ : Type
| procedure Ident is Block
| procedure Ident (Ident : Type) is Block

Type ::= integer | boolean

Cmd ::= Cmd ; Cmd
| Ident ::= Expr
| skip
| if Expr then Cmd else Cmd
| if Expr then Cmd
| while Expr do Cmd
| declare Block
| Ident
| Ident (Expr)

Expr ::= Numeral | Ident | true | false | – Expr

| Expr Operator Expr | not(Expr)
| Operator ::= + | – | * | / | or | and
| <= | < | = | > | >= | <>

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

Pelican Program

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 if
end if;
begin read num ; pf(two) end

Semantic Domains

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

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, … }

Store = Location → SV + unused + undefined

Environment = Identifier → DV + unbound

Procedure = proc0(Store → Store) + proc1(Location → Store → Store)
Environments

Sets of bindings of identifiers to denotable values

In Pelican:

\[ DV = \text{int(Integer)} + \text{bool(Boolean)} + \text{var(Location)} + \text{proc0(Store \rightarrow Store)} + \text{proc1(Location \rightarrow Store \rightarrow Store)} \]

Operations on Environments

**emptyEnv : Env**

\[ \forall I \in \text{Identifier}, \ emptyEnv I = \text{unbound} \]

**extendEnv : Env x Identifier x DV \rightarrow Env**

\[ \forall X \in \text{Identifier}, \ extendEnv(env,I,dval) X = \begin{cases} dval & \text{if } X = I \text{ then } dval \text{ else } env(X) \end{cases} \]

**applyEnv : Env x Identifier \rightarrow DV + \text{unbound}**

\[ \text{applyEnv}(env,I) = env(I) \]

Stores

Store = Location \rightarrow SV + unused + undefined

Operations on Stores

**emptySto : Store**

\[ \forall loc \in \text{Location}, \ emptySto \ loc = \text{unused} \]

**updateSto : Store x Location x (SV + undefined + unused) \rightarrow Store**

\[ \forall X \in \text{Location}, \ updateSto(sto,loc,val) X = \begin{cases} val & \text{if } X = \text{loc} \text{ then } val \text{ else } sto(X) \end{cases} \]

**applySto : Store x Location \rightarrow SV + undefined + unused**

\[ \text{applySto}(sto,loc) = sto(loc) \]

Semantic Equations

**meaning [program \ is B] = perform [B] emptyEnv emptySto**

**perform [D \ begin C \ end] env sto = execute [C] env1 sto1**

\[ \text{where } (env1, sto1) = \text{elaborate } [D] \ env \ sto \]

**elaborate [D1 D2] env sto =**

\[ \text{elaborate } [D2] \ env1 \ sto1 \]

\[ \text{where } (env1, sto1) = \text{elaborate } [D1] \ env \ sto \]

**elaborate [c] env sto = (env, sto)**

**elaborate [const I = E] env sto = (extendEnv(env,I,evaluate [E] env \ sto), sto)**
elaborate \[ \text{var} \mathbf{I} : T \] \ env \ sto =
(extendEnv(env, I, var(loc)), sto1)
where (sto1, loc) = allocate sto

elaborate \[ \text{var} \mathbf{L} : T \] \ env \ sto =
elaborate \[ \text{var} \mathbf{L} : T \] \ env1 \ sto1
where (env1, sto1) =
elaborate \[ \text{var} \mathbf{I} : T \] \ env \ sto

execute \[ \mathbf{C}_1 \ ; \mathbf{C}_2 \] \ env \ sto =
exe\[ \mathbf{C}_2 \] \ env (execute \[ \mathbf{C}_1 \] \ env \ sto)

execute \[ \text{skip} \] \ env \ sto = sto

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

execute \[ \text{if} \ \mathbf{E} \ \text{then} \mathbf{C} \ \text{else} \mathbf{C}_2 \] \ env \ sto =
if p then execute \[ \mathbf{C} \] \ env \ sto
else execute \[ \mathbf{C}_2 \] \ env \ sto
where bool(p) = evaluate \[ \mathbf{E} \] \ env \ sto

execute \[ \mathbf{while} \mathbf{E} \ \text{do} \mathbf{C} \] \ = loop
where

execute \[ \text{declare} \mathbf{B} \] \ env \ sto =
perform \[ \mathbf{B} \] \ env \ sto

execute \[ \mathbf{I} := \mathbf{E} \] \ env \ sto =
if dval = int(n) or dval = bool(p)
then dval
else if dval = var(loc)
then if applySto(sto, loc) = undefined
then error
else applySto(sto, loc)
where dval = applyEnv(env, I)

evaluate \[ \mathbf{N} \] \ env \ sto = int(value \[ \mathbf{N} \])
evaluate \[ \mathbf{true} \] \ env \ sto = bool(true)
evaluate \[ \mathbf{false} \] \ env \ sto = bool(false)

evaluate \[ \mathbf{E}_1 + \mathbf{E}_2 \] \ env \ sto = int(plus(m,n))
where int(m) = evaluate \[ \mathbf{E}_1 \] \ env \ sto
and int(n) = evaluate \[ \mathbf{E}_2 \] \ env \ sto

evaluate \[ \mathbf{E}_1 = \mathbf{E}_2 \] \ env \ sto = bool(equal(m,n))
where int(m) = evaluate \[ \mathbf{E}_1 \] \ env \ sto
and int(n) = evaluate \[ \mathbf{E}_2 \] \ env \ sto

evaluate \[ \mathbf{E}_1 \ \text{and} \ \mathbf{E}_2 \] \ env \ sto =
if p then bool(q) else bool(false)
where bool(p) = evaluate \[ \mathbf{E}_1 \] \ env \ sto
and bool(q) = evaluate \[ \mathbf{E}_2 \] \ 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 denotable values, so dval \neq \text{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.

evaluate \[ \mathbf{I} \] \ env \ sto =
if dval = int(n) or dval = bool(p)
then dval
else if dval = var(loc)
then if applySto(sto, loc) = undefined
then error
else applySto(sto, loc)
where dval = applyEnv(env, I)
Procedures

elaborate \[\text{procedure}\] is B | env sto = (env₁, sto)

where \(\text{env₁} = \text{extendEnv}(\text{env}, \text{i}, \text{proc0(proc)})\)

and \(\text{proc} = \text{perform}[\text{B}]\ \text{env₁}\)

elaborate \[\text{procedure}_1(\text{i}_2 : \text{T})\] is B | env sto = (env₁, sto)

where \(\text{env₁} = \text{extendEnv}(\text{env}, \text{i}_1, \text{proc1(proc)})\)

and \(\text{proc}\ \text{loc} = \) perform \[\text{B}\] extendEnv(\text{env₁}, \text{i}_2, \text{var(loc)})

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:

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

Procedure Calls

execute \[\text{I}\] env sto = proc sto

where \(\text{proc0(proc)} = \text{applyEnv}(\text{env}, \text{i})\)

execute \[\text{I}(\text{E})\] env sto = proc loc updateSto(sto₁, loc, evaluate \[\text{E}\] env sto)

where \(\text{proc1(proc)} = \text{applyEnv}(\text{env}, \text{i})\)

and \((\text{sto₁, loc}) = \text{allocate sto}\)

Example

program prfacs is

\[\text{var} n : \text{integer};\]
\[\text{procedure} \text{pf}(\text{d:integer})\] is

\[\text{var} q : \text{integer} \in \text{env}_{2,L}\]

begin
  \(\text{if } n > 1 \text{ end if}\)
  \(\text{then} q := n/d; \text{if } n=d*q \text{ end if}\)
  \(\text{then} \text{write}(\text{d}); n:=q; \text{pf}(\text{d}) \text{ end if}\)
  \(\text{else} \text{pf}(\text{d}+1) \text{ end if}\)
end if

begin \(n := 20; \text{pf}(2)\) end

where

\(\text{proc } L = \text{perform}[\text{var} q:\text{int}; \text{begin if } n > 1 \text{ then } \ldots\] \text{extendEnv}(\text{env}_{1,d,\text{var}(L)})\)

\(\text{env}_1 = [\text{pf} | \rightarrow \text{proc1(proc)}, n | \rightarrow \text{var}(0)]\)

\(\text{env}_{2,L} = [d | \rightarrow \text{var}(L), q | \rightarrow \text{var}(L+1), \ldots \text{pf} | \rightarrow \text{proc1(proc)}, n | \rightarrow \text{var}(0)]\)

for \(L = 1,3,5,7,9,11,13\)
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

\[
\text{validate } [\text{program} \text{ is } B] = \text{examine } [B] \text{ extendEnv}(\text{emptyEnv}, I, \text{program})
\]

\[
\text{examine } [D \text{ begin } C \text{ end}] \text{ env} = \text{check } [C] \text{ env}_1
\]

where (locenvv, envv) =

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

\[
\text{elaborate } [D_1 \ D_2] = (\text{elaborate } [D_2]) \circ (\text{elaborate } [D_1])
\]

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

\[
\text{elaborate } [\text{const} I = E] \text{ (locenv, env)} = \\
\text{if } \text{applyEnv}(\text{locenv}, I) = \text{unbound} \\
\text{then } (\text{extendEnv}(\text{locenv}, I, \text{typify}_{[E]} \text{ env}), \\
\text{extendEnv}(\text{env}, I, \text{typify}_{[E]} \text{ env}) \\
\text{else } \text{error}
\]

\[
\text{elaborate } [\text{var} I : T] \text{ (locenv, env)} = \\
\text{if } \text{applyEnv}(\text{locenv}, I) = \text{unbound} \\
\text{then } (\text{extendEnv}(\text{locenv}, I, \text{type}(T)), \\
\text{extendEnv}(\text{env}, I, \text{type}(T)), \\
\text{else } \text{error}
\]

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

\[
\text{check } [C_1 ; C_2] \text{ env} = \\
(\text{check } [C_1] \text{ env}) \text{ and } (\text{check } [C_2] \text{ env})
\]

\[
\text{check } [\text{skip}] \text{ env} = \text{true}
\]

\[
\text{check } [I : = E] \text{ env} = \\
(\text{applyEnv}(\text{env}, I) = \text{intvar} \\
\text{and } \text{typify}_{[E]} \text{ env} = \text{integer}) \\
\text{or } \\
(\text{applyEnv}(\text{env}, I) = \text{boolvar} \\
\text{and } \text{typify}_{[E]} \text{ env} = \text{boolean})
\]
check \[ \text{if } E \text{ then } C \] env =
\[(\text{typify } [E] \text{ env } = \text{ boolean}) \text{ and } (\text{check } [C] \text{ env})\]

check \[ \text{if } E \text{ then } C_1 \text{ else } C_2 \] env =
\[(\text{typify } [E] \text{ env } = \text{ boolean}) \text{ and } (\text{check } [C_1] \text{ env}) \text{ and } (\text{check } [C_2] \text{ env})\]

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

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

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

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

\[
\text{typify } [I] \text{ env } = \begin{cases} 
\text{case applyEnv(\text{env, } I) of} \\
\text{intvar, integer} : \text{integer} \\
\text{boolvar, boolean} : \text{boolean} \\
\text{program} : \text{program} \\
\text{unbound} : \text{error}
\end{cases}
\]

\[
\text{typify } [N] \text{ env } = \text{integer}
\]

\[
\text{typify } [\text{true}] \text{ env } = \text{boolean}
\]

\[
\text{typify } [\text{false}] \text{ env } = \text{boolean}
\]

\[
\text{typify } [E_1 + E_2] \text{ env } =
\begin{cases} 
\text{if } (\text{typify } [E_1] \text{ env } = \text{integer}) \\
\text{and } (\text{typify } [E_2] \text{ env } = \text{integer}) \\
\text{then integer} \text{ else error}
\end{cases}
\]

\[
\text{typify } [E_1 \text{ and } E_2] \text{ env } =
\begin{cases} 
\text{if } (\text{typify } [E_1] \text{ env } = \text{boolean}) \\
\text{and } (\text{typify } [E_2] \text{ env } = \text{boolean}) \\
\text{then boolean} \text{ else error}
\end{cases}
\]

\[
\text{typify } [E_1 < E_2] \text{ env } =
\begin{cases} 
\text{if } (\text{typify } [E_1] \text{ env } = \text{integer}) \\
\text{and } (\text{typify } [E_2] \text{ env } = \text{integer}) \\
\text{then boolean} \text{ else error}
\end{cases}
\]

Example (Falsely rejected by version in text)

The program identifier “bug” is ignored to save space.

\[
\begin{align*}
\text{program} & \quad \text{bug is} \\
\text{const} & \quad c = 5; \quad [c \rightarrow \text{int}] \quad [c \rightarrow \text{int}] \\
\text{var} & \quad k : \text{integer}; \quad [k \rightarrow \text{ivar}, c \rightarrow \text{int}] \quad [k \rightarrow \text{ivar}, c \rightarrow \text{int}] \\
\text{begin} & \quad k := 99; \quad [k \rightarrow \text{ivar}, c \rightarrow \text{int}] \\
\text{declare} & \quad \text{[]} \quad [k \rightarrow \text{ivar}, c \rightarrow \text{int}] \\
\text{const} & \quad d = c+k; \quad [d \rightarrow \text{int}] \quad [d \rightarrow \text{int}, k \rightarrow \text{ivar}, c \rightarrow \text{int}] \\
\text{var} & \quad m : \text{integer}; \quad [m \rightarrow \text{ivar}, d \rightarrow \text{int}] \quad [m \rightarrow \text{ivar}, d \rightarrow \text{int}, k \rightarrow \text{ivar}, c \rightarrow \text{int}] \\
\text{begin} & \quad m := c+d+k; \quad [m \rightarrow \text{ivar}, d \rightarrow \text{int}, k \rightarrow \text{ivar}, c \rightarrow \text{int}] \\
\text{write} & \quad m \quad [m \rightarrow \text{ivar}, d \rightarrow \text{int}, k \rightarrow \text{ivar}, c \rightarrow \text{int}] \\
\text{end} & \quad \text{end}
\end{align*}
\]

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: \text{goto, stop, return exit break continue raise and resume}

Example

\[
\begin{align*}
\text{begin} & \quad L_1 : C_1; \quad L_2 : C_2; \quad L_3 : C_3; \quad L_4 : C_4 \text{ end}
\end{align*}
\]

Meaning with Direct Semantics

\[
\text{execute } [C_4] \circ \text{execute } [C_3] \\
\circ \text{execute } [C_2] \circ \text{execute } [C_1]
\]

Store Transformation

\[
\text{sto}_0 \rightarrow \text{execute } [C_1] \rightarrow \text{execute } [C_2] \\
\rightarrow \text{execute } [C_3] \rightarrow \text{execute } [C_4] \rightarrow \text{sto}_{\text{final}}
\]
What if “C 3 ” is “if x>0 then goto L 1 else skip”?

**Store Transformation if x>0**

\[
\text{sto}_0 \rightarrow \text{execute } [C_1] \rightarrow \text{execute } [C_2] \\
\rightarrow \text{execute } [C_3] \rightarrow \text{execute } [C_1] \rightarrow \text{etc.}
\]

“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 \(\leq\) 0, 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.

**Continuations**

**Semantic Domain**

Continuation = Store \(\rightarrow\) Store

A continuation models the remainder of the program from a point in the code.

Labels are bound to continuations in the environment.

**Identifier** \hspace{1.5em} **Denotable Value**

\[
\begin{align*}
L_1 & \quad \text{cont}_1 = \text{execute } [C_1; C_2; C_3; C_4] \text{ env} \\
L_2 & \quad \text{cont}_2 = \text{execute } [C_2; C_3; C_4] \text{ env} \\
L_3 & \quad \text{cont}_3 = \text{execute } [C_3; C_4] \text{ env} \\
L_4 & \quad \text{cont}_4 = \text{execute } [C_4] \text{ env}
\end{align*}
\]

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.

**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**

\[
\begin{align*}
P : \text{Program} & \quad L : \text{Label} & \quad O : \text{Operator} \\
S : \text{Series} & \quad I : \text{Identifier} & \quad N : \text{Numeral} \\
C : \text{Command} & \quad E : \text{Expression}
\end{align*}
\]
Abstract Production Rules

Program ::= 
    program Identifier is begin Series end

Series ::= Command

Command ::= Command ; Command
    | Identifier ::= Expr
    | while Expr do Series
    | if Expr then Series else Series | skip
    | stop | goto Label
    | begin Series end | Label : Command

Expr ::= Identifier | Numeral | – Expr
    | Expr Operator Expr

Operator ::= + | – | * | / | = | <= | < | > | >= | <>

Label ::= Identifier

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 → Continuation → Store → Store
evaluate : Expr → Store → EV

Semantic Equations

meaning [program I is begin S end] = perform [S] emptyEnv identityCont emptySto
perform [L1:C1; L2:C2; … ; Ln:Cn] env cont = 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,1,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

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).
execute [while E do S] env cont sto = loop
  where loop env cont sto =
    if p then perform [S] env {loop env cont} sto
    else cont sto
  where bool(p) = evaluate [E] sto
execute [C₁ ; C₂] env cont sto =
  execute [C₁] env {execute [C₂] env cont} sto
execute [begin S end] env cont sto =
  perform [S] env cont sto
execute [goto L] env cont sto =
  applyEnv(env,L) sto
execute [L : C] = execute [C]
evaluate [I] sto = applySto(sto,I)
evaluate [N] sto = value [N]
evaluate [-E] = minus(0,m)
  where int(m) = evaluate [E] sto
evaluate [E₁ + E₂] sto = int(plus(m,n))
  where int(m) = evaluate [E₁] sto
  and int(n) = evaluate [E₂] 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