Denotational Semantics

Basic Idea

Map syntactic objects into domains of mathematical objects.

meaning : Syntax → Semantics

Example

meaning ⟦**26/2**⟧ = *meaning* ⟦**(10+3)**⟧
= *meaning* ⟦**013**⟧ = *meaning* ⟦**13**⟧ = 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 ::=
while Expression **do** Command+

E ::= N | I | E O E | – E

use E ::= N | I | E_1 O E_2 | – 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 → Integer)

Consists of sets of bindings (functions) of variables to integers.

A → 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

meaning : Program → Store

evaluate : Expression → (Store → Value)

Semantic equations

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

Example

evaluate ⟦$E_1 * E_2$⟧ sto =
  times(*evaluate* ⟦E_1⟧ sto, *evaluate* ⟦E_2⟧ sto)

The value of an expression "$E_1 * E_2$" is the mathematical product of the values of its component subexpressions.

Auxiliary Functions

plus : Number x Number → Number

minus : Number x Number → Number

times : Number x Number → Number

Describe operations in the semantic domains.

Improve readability of denotational definitions.

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

value ⟦N D⟧ =
  plus(*times*(10,*value* ⟦N⟧), *digit* ⟦D⟧)
value ⟦D⟧ = *digit* ⟦D⟧

digit ⟦**0**⟧ = 0	*digit* ⟦**5**⟧ = 5
digit ⟦**1**⟧ = 1	*digit* ⟦**6**⟧ = 6
digit ⟦**2**⟧ = 2	*digit* ⟦**7**⟧ = 7
digit ⟦**3**⟧ = 3	*digit* ⟦**8**⟧ = 8
digit ⟦**4**⟧ = 4	*digit* ⟦**9**⟧ = 9

Denotational Evaluation

value ⟦**905**⟧ = *plus*(*times*(10,*value* ⟦**90**⟧),*digit* ⟦**5**⟧)

= *plus*(*times*(10,
  plus(*times*(10,*value* ⟦**9**⟧),
  digit ⟦**0**⟧)),5)

= *plus*(*times*(10,
  plus(*times*(10,*digit* ⟦**9**⟧),0)),5)

= *plus*(*times*(10,
  plus(*times*(10,9),0)),5)

= *plus*(*times*(10,*plus*(90,0)),5)

= *plus*(*times*(10,90),5)

= *plus*(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**.

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 → B
where A has a binary operation f : A x A → A
and B has a binary operation g : B x B → B.

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

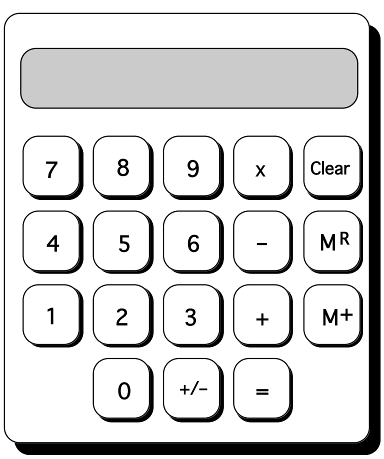
The semantic function *value* is a homomorphism.

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

Therefore, *value*(f(x,y)) = g(*value*(x),*value*(y)), thus demonstrating that *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 x 2 + 30 =**

Resulting Display: 74

Ignore unusual combinations of keystrokes.

Concrete Syntax

<program> ::= <expression sequence>

<expression sequence> ::= <expression>
| <expression> <expression sequence>

<expression> ::= <term>
| <expression> <operator> <term>
| <expression> <answer>
| <expression> <answer> **+/-**

<term> ::= <numeral> | $\mathbf{M^R}$
| **Clear** | <term> **+/-**

<operator> ::= **+** | **–** | **x**

<answer> ::= **M+** | **=**

<numeral> ::= <digit> | <numeral> <digit>

<digit> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

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 | $\mathbf{M^R}$ | **Clear**
| Expression Operator Expression
| Expression Answer

Operator ::= **+** | **–** | **x**

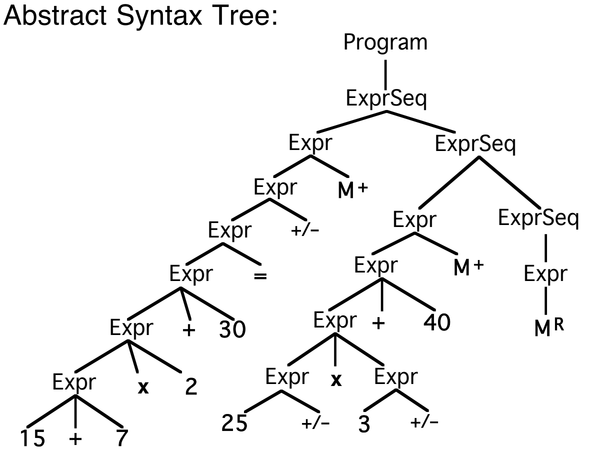
Answer ::= $\mathbf{M^+}$ | **=** | **+/-**

Numeral ::= see Figure 9.1

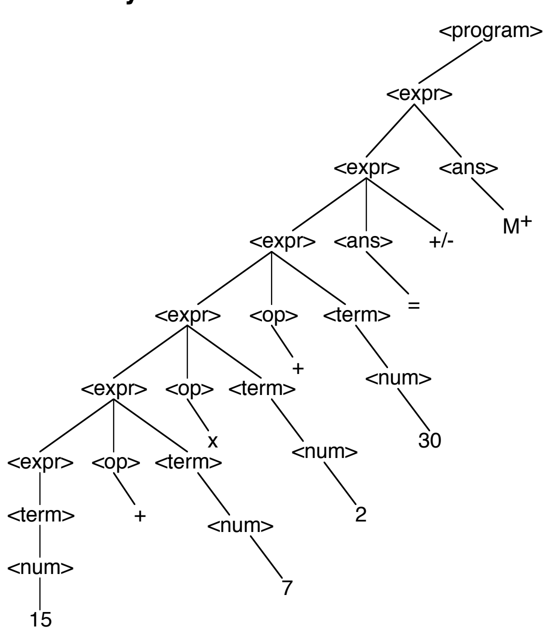
A Keystroke Sequence

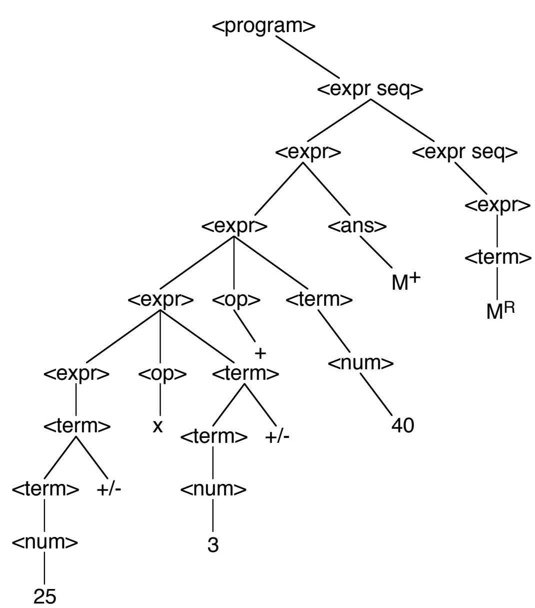
15 + 7 x 2 + 30 = +/- M^+ 25 +/- x 3 +/- + 40 M^+ M^R

Abstract Syntax Tree:



Concrete Syntax





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 = { … , -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

Key	Acc	OprFlag	Dsply	Mem
Initial	⇒ 0	*nop*	0	0
15	0	*nop*	15	0
+	15	*plus*	15	0
7	15	*plus*	7	0
x	22	*times*	22	0
2	22	*times*	2	0
+	44	*plus*	44	0
30	44	*plus*	30	0
=	44	*nop*	74	0
+/-	44	*nop*	-74	0
M+	44	*nop*	-74	-74
25	44	*nop*	25	-74
+/-	44	*nop*	-25	-74
x	-25	*times*	-25	-74
3	-25	*times*	3	-74
+/-	-25	*times*	-3	-74
+	75	*plus*	75	-74
40	75	*plus*	40	-74
M+	75	*nop*	115	41
M^R	75	*nop*	41	41

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

meaning ⟦P⟧ = d
where (a,op,d,m) = *perform* ⟦P⟧ (0,*nop*,0,0)

perform ⟦E S⟧ = *perform* ⟦S⟧ ∘ *evaluate* ⟦E⟧

perform ⟦E⟧ = *evaluate* ⟦E⟧

evaluate ⟦N⟧ (a,op,d,m) = (a,op,v,m)
where v = *value* ⟦N⟧

evaluate ⟦$\mathbf{M^R}$⟧ (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⟧ ∘ *compute* ⟦O⟧ ∘ *evaluate* ⟦E_1⟧

evaluate ⟦E A⟧ = *calculate* ⟦A⟧ ∘ *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 ⟦$\mathbf{M^+}$⟧ (a,op,d,m) = (a,*nop*,v,*plus*(m,v))
where v = op(a,d)

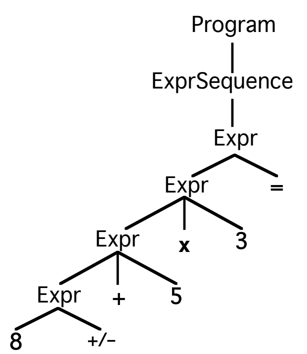
calculate ⟦+/-⟧ (a,op,d,m) = (a,op,*minus*(0,d),m)

value ⟦N⟧ = 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* ⟦**=**⟧ ∘
evaluate ⟦**8 +/- + 5 x 3**⟧) (0,*nop*,0,0)

= (*calculate* ⟦**=**⟧ ∘ *evaluate* ⟦**3**⟧ ∘ *compute* ⟦**x**⟧ ∘
evaluate ⟦**8 +/- + 5**⟧) (0,*nop*,0,0)

= (*calculate* ⟦**=**⟧ ∘ *evaluate* ⟦**3**⟧ ∘ *compute* ⟦**x**⟧ ∘
evaluate ⟦**5**⟧ ∘ *compute* ⟦**+**⟧ ∘
evaluate ⟦**8 +/-**⟧) (0,*nop*,0,0)

= (*calculate* ⟦**=**⟧ ∘ *evaluate* ⟦**3**⟧ ∘ *compute* ⟦**x**⟧ ∘
evaluate ⟦**5**⟧ ∘ *compute* ⟦**+**⟧ ∘
calculate ⟦**+/-**⟧ ∘ *evaluate* ⟦**8**⟧) (0,*nop*,0,0)

= (*calculate* ⟦**=**⟧ (*evaluate* ⟦**3**⟧
(*compute* ⟦**x**⟧ (*evaluate* ⟦**5**⟧
(*compute* ⟦**+**⟧ (*calculate* ⟦**+/-**⟧
(*evaluate* ⟦**8**⟧ (0,*nop*,0,0)))))))

= (*calculate* ⟦**=**⟧ (*evaluate* ⟦**3**⟧ (*compute* ⟦**x**⟧
(*evaluate* ⟦**5**⟧ (*compute* ⟦**+**⟧
(*calculate* ⟦**+/-**⟧ (0,*nop*,8,0)))))))

= (*calculate* ⟦**=**⟧ (*evaluate* ⟦**3**⟧
(*compute* ⟦**x**⟧ (*evaluate* ⟦**5**⟧
(*compute* ⟦**+**⟧ (0,*nop*,-8,0))))))

= (*calculate* ⟦**=**⟧ (*evaluate* ⟦**3**⟧
(*compute* ⟦**x**⟧ (*evaluate* ⟦**5**⟧
(-8,*plus*,-8,0)))))

= (*calculate* ⟦**=**⟧ (*evaluate* ⟦**3**⟧
(*compute* ⟦**x**⟧ (-8,*plus*,5,0))))

= (*calculate* ⟦**=**⟧ (*evaluate* ⟦**3**⟧ (-3,*times*,-3,0)))

= (*calculate* ⟦**=**⟧ (-3,*times*,3,0))

= (-3,*nop*,-9,0)

Therefore *meaning* ⟦**8 +/- + 5 x 3 =** ⟧ = -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 **while** command in Wren.

Ignore input and output in Wren for now.

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**
| **<=** | **<** | **=** | **>** | **>=** | **<>**

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
  Store = Identifier → (SV + *undefined*).
- Each sto : Store is *undefined* for all but a finite set of identifiers (called a finite function).
- Notational convention: Represent a store as a set of bindings.

  sto = { count ↦*int*(1), total ↦*int*(0) }

  Assume that sto(I) = *undefined* for all other identitifers I.

  Let { } represent an everywhere undefined store.

Operations on Stores

emptySto : Store

∀I∈Identifier, *emptySto* I = *undefined*

updateSto : Store x Identifier x SV → Store

∀X∈Identifier,
updateSto(sto,I,val) X =
if X = I then val else sto(X)

applySto : Store x Identifier → SV + *undefined*

applySto(sto,I) = sto(I)

Example

If sto = { a ↦*int*(3), b ↦*int*(5) },

updateSto(sto,b,8) = { a ↦*int*(3), b ↦*int*(8) }

and

updateSto(sto,c,-99) =
{ a ↦*int*(3), b ↦*int*(5), c ↦*int*(-99) }

Motivating the Definition

For sto:Store,
sto : Identifer → (SV + *undefined*)

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

For sto:Store, I:Identifier, val:SV,

updateSto(sto,I,val) : Identifer → (SV+*undefined*)

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

∀X : Identifier,
updateSto(sto,I,val) X =
if X = I then val else sto(X)

Expressible Values

- Values that expressions can produce.
- Expressible values in Wren:
  EV = *int*(Integer) + *bool*(Boolean)

Auxiliary Functions

plus : Integer x Integer → Integer

minus : Integer x Integer → Integer

times : Integer x Integer → Integer

divides : Integer x Integer → Integer

less : Integer x Integer → Boolean

lesseq : Integer x Integer → Boolean

greater : Integer x Integer → Boolean

greatereq : Integer x Integer → Boolean

equal : Integer x Integer → Boolean

neq : Integer x Integer → 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 [[**program** I **is** D **begin** C **end**]] =
execute [[C]] *emptySto*

execute [[C_1 **;** C_2]] = *execute* [[C_2]] ∘ *execute* [[C_1]]

execute [[**skip**]] sto = sto

execute [[I **:=** E]] sto =
updateSto(sto, I, (*evaluate* [[E]] sto))

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

execute [[**if** E **then** C_1 **else** C_2]] sto =
if p then *execute* [[C_1]] sto else *execute* [[C_2]] sto
where *bool*(p) = *evaluate* [[E]] sto

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

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

evaluate [[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

:

evaluate [[E_1 **<** E_2]] sto =
if *less*(m,n) then *bool*(true) else *bool*(false)
where *int*(m) = *evaluate* [[E_1]] sto
and *int*(n) = *evaluate* [[E_2]] sto

:

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

evaluate [[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.
- $\rightarrow$ 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).

These conventions agree:

execute : Command → Store → Store

execute ⟦a **:= 0;** b **:= 1**⟧ : Store → Store

execute ⟦a **:= 0;** b **:= 1**⟧ *emptySto* : 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.

while E **do** C ≡

if E **then** (C**; while** E **do** C) **else skip**

execute ⟦**while** E **do** C⟧ sto
  = loop_1 sto
    where loop_1 s =
      if *evaluate* ⟦E⟧ s = *bool*(true)
        then loop_1(*execute* ⟦C⟧ s)
        else s
  = if *evaluate* ⟦E⟧ sto = *bool*(true)
        then loop_1(*execute* ⟦C⟧ sto)
        else sto
    where loop_1 s =
      if *evaluate* ⟦E⟧ s = *bool*(true)
        then loop_1(*execute* ⟦C⟧ s)
        else s

execute ⟦**if** E **then** (C**; while** E **do** C)
        else skip⟧ sto
  = if *evaluate* ⟦E⟧ sto = *bool*(true)
      then *execute* ⟦C**; while** E **do** C⟧ sto
      else *execute* ⟦**skip**⟧ sto
  = if *evaluate* ⟦E⟧ sto = *bool*(true)
      then (*execute* ⟦**while** E **do** C⟧
        ∘ *execute* ⟦C⟧) sto
      else sto
  = if *evaluate* ⟦E⟧ sto = *bool*(true)
      then *execute* ⟦**while** E **do** C⟧
        (*execute* ⟦C⟧ sto)
      else sto
  = if *evaluate* ⟦E⟧ sto = *bool*(true)
        then loop_2 (*execute* ⟦C⟧ sto)
        else sto
    where loop_2 s =
      if *evaluate* ⟦E⟧ s = *bool*(true)
        then loop_2(*execute* ⟦C⟧ s)
        else s

Now observe that loop_1 and 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.

meaning : Program → Input → Output

Commands may change the input and output lists, so

execute : Command → State → State

where

State = Store x Input x Output.

Use auxiliary functions to manipulate lists:

head : Integer* → Integer

head $[n_1,n_2, ..., n_k] = n_1$ provided $k \geq 1$.

tail : Integer* → Integer*

tail $[n_1,n_2, ..., n_k] = [n_2, ..., n_k]$ provided $k \geq 1$.

null : Integer* → Boolean

null $[n_1,n_2, ..., n_k] = (k=0)$

affix : Integer* x Integer → Integer*

affix $([n_1,n_2, ..., n_k],m) = [n_1,n_2, ..., n_k,m]$.

New Semantic Equations

meaning ⟦**program** I **is** D **begin** C **end**⟧ inp

= outp

where (sto, inp_1, outp) =

execute ⟦C⟧ (*emptySto*, inp, [ ])

execute ⟦**read** I⟧ (sto,inp,outp) =

if *null*(inp)

then *error*

else

(*updateSto*(sto,I,*int*(*head*(inp))),*tail*(inp),outp)

execute ⟦**write** E⟧ (sto,inp,outp) =

(sto, inp, *affix*(outp,val))

where *int*(val) = *evaluate* ⟦E⟧ sto.

Every equation for *execute* needs to be altered.

Elaborating a Denotational Definition

program sample **is**
  var sum,num : **integer**;
 begin
  sum := 0; **read** num;
  while num>=0 **do**
   if num>9 **and** num<100
    then sum := sum+num **end if**;
   read num
  end while;
  write sum
 end

Input list = [5,22,-1]

Abbreviations:

d = **var** sum,num : **integer**
c_1 = sum := 0
c_2 = **read** num
c_3 = **while** num>=0 **do** $c_{3.1}$; $c_{3.2}$
 $c_{3.1}$ = **if** num>9 **and** num<100
     then sum := sum+num
 $c_{3.2}$ = **read** num
c_4 = **write** sum

Meaning of the Program:

meaning ⟦**program** sample **is** d **begin** c_1 ;
c_2 ; c_3 ; c_4 **end**⟧ [5,22,-1] = outp
where (sto, inp_1, outp) =
execute ⟦c_1 ; c_2 ; c_3 ; c_4⟧(*emptySto*,[5,22,-1],[])

execute ⟦c_1 ; c_2 ; c_3 ; c_4⟧(*emptySto*,[5,22,-1],[]) =
(*execute* ⟦c_4⟧ ∘ *execute* ⟦c_3⟧ ∘
execute ⟦c_2⟧ ∘ *execute* ⟦c_1⟧)
(*emptySto*, [5,22,-1], [])

The commands are executed from inside out.

execute ⟦sum := 0⟧ (*emptySto*, [5,22,-1], [])
= (*updateSto*(*emptySto*, sum,
(*evaluate* ⟦0⟧ *emptySto*)), [5,22,-1], [])
= (*updateSto*(*emptySto*, sum, *int*(0)),
[5,22,-1], [])
= ({sum↦*int*(0)}, [5,22,-1], [])

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 $sto_{0,5}$ = {sum↦*int*(0),num↦*int*(5)}

execute ⟦**while** num>=0 **do** $c_{3.1}$; $c_{3.2}$ ⟧
($sto_{0,5}$, [22,-1], [])
= loop ($sto_{0,5}$, [22,-1], [])
where loop (sto,in,out) =
if p
then loop(*execute* ⟦$c_{3.1}$; $c_{3.2}$⟧ (sto,in,out))
else (sto,in,out)
where *bool*(p) = *evaluate* ⟦num>=0⟧ sto

We work on the boolean expression first.

evaluate ⟦num⟧ $sto_{0,5}$ =
applySto($sto_{0,5}$, num) = *int*(5)

evaluate ⟦0⟧ $sto_{0,5}$ = *int*(0)

evaluate ⟦num>=0⟧ $sto_{0,5}$
= if *greatereq*(m,n) then *bool*(true) else *bool*(false
where *int*(m) = *evaluate* ⟦num⟧ $sto_{0,5}$
and *int*(n) = *evaluate* ⟦0⟧ $sto_{0,5}$
= if *greatereq*(5,0) then *bool*(true) else *bool*(false)
= *bool*(true)

Now we can execute loop for the first time.

loop ($sto_{0,5}$, [22,-1], [])
= if true then loop(*execute* ⟦$c_{3.1}$; $c_{3.2}$⟧
($sto_{0,5}$, [22,-1], []))
else ($sto_{0,5}$, [22,-1], [])
= loop(*execute* ⟦$c_{3.1}$; $c_{3.2}$⟧ ($sto_{0,5}$, [22,-1], []))

To complete the execution of loop, we need to execute the body of the **while** command.

execute ⟦$c_{3.1}$; $c_{3.2}$⟧ ($sto_{0,5}$, [22,-1], [])
= *execute* ⟦**read** num⟧
(*execute* ⟦**if** num>9 **and** num<100
then sum := sum+num⟧ ($sto_{0,5}$, [22,-1], []))

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

evaluate ⟦num>9⟧ $sto_{0,5}$
= if *greater*(m,n) then *bool*(true) else *bool*(false)
where *int*(m) = *evaluate* ⟦num⟧ $sto_{0,5}$
and *int*(n) = *evaluate* ⟦9⟧ $sto_{0,5}$
= if *greater*(5,9) then *bool*(true) else *bool*(false)
= *bool*(false)

evaluate ⟦num<100⟧ $sto_{0,5}$
= if *less*(m,n) then *bool*(true) else *bool*(false)
where *int*(m) = *evaluate* ⟦num⟧ $sto_{0,5}$
and *int*(n) = *evaluate* ⟦100⟧ $sto_{0,5}$
= if *less*(5,100) then *bool*(true) else *bool*(false)
= *bool*(true)

evaluate ⟦num>9 **and** num<100⟧ $sto_{0,5}$
= if p then *bool*(q) else *bool*(false)
where *bool*(p) = *evaluate* ⟦num>9⟧ $sto_{0,5}$
and *bool*(q) = *evaluate* ⟦num<100⟧ $sto_{0,5}$
= if false then *bool*(true) else *bool*(false)
= *bool*(false)

Continuing with the **if** command, we get:

execute ⟦**if** num>9 **and** num<100 **then** sum := sum+num⟧ ($sto_{0,5}$, [22,-1], [])
= if p then *execute* ⟦sum := sum+num⟧ ($sto_{0,5}$, [22,-1], [])
else ($sto_{0,5}$, [22,-1], [])
where *bool*(p) = *evaluate* ⟦num>9 **and** num<100⟧ $sto_{0,5}$
= if false then *execute* ⟦sum := sum+num⟧ ($sto_{0,5}$, [22,-1], [])
else ($sto_{0,5}$, [22,-1], [])
= ($sto_{0,5}$, [22,-1], [])

After finishing with the **if** command, we proceed with the second command in the body of **while**.

execute ⟦**read** num⟧ ($sto_{0,5}$, [22,-1], []))
= (*updateSto*($sto_{0,5}$,num,*int*(22)), [-1], [])
= ({sum↦*int*(0),num↦*int*(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.

execute ⟦$c_{3.1}$; $c_{3.2}$⟧ ($sto_{0,5}$, [22,-1], [])
= ($sto_{0,22}$, [-1], [])

This completes the first pass through the loop.

loop ($sto_{0,5}$, [22,-1], [])
= loop(*execute* ⟦$c_{3.1}$; $c_{3.2}$⟧ ($sto_{0,5}$, [22,-1], []))
= loop($sto_{0,22}$, [-1], [])

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

evaluate ⟦num⟧ $sto_{0,22}$
= *applySto*($sto_{0,22}$, num) = *int*(22)

evaluate ⟦0⟧ $sto_{0,22}$ = *int*(0)

evaluate ⟦num>=0⟧ $sto_{0,22}$
= if *greatereq*(m,n) then *bool*(true)
else *bool*(false)
where *int*(m) = *evaluate* ⟦num⟧ $sto_{0,22}$
and *int*(n) = *evaluate* ⟦0⟧ $sto_{0,22}$
= if *greatereq*(22,0) then *bool*(true)
else *bool*(false)
= *bool*(true)

Now we can execute loop for the second time.

loop ($sto_{0,22}$, [-1], [])
= if true then loop(*execute* ⟦$c_{3.1}$; $c_{3.2}$⟧ ($sto_{0,22}$, [-1], []))
else ($sto_{0,22}$, [-1], [])
= loop(*execute* ⟦$c_{3.1}$; $c_{3.2}$⟧ ($sto_{0,22}$, [-1], []))

Again we execute the body of the **while** command.

execute ⟦$c_{3.1}$; $c_{3.2}$⟧ ($sto_{0,22}$, [-1], [])
= *execute* ⟦**read** num⟧
(*execute* ⟦**if** num>9 **and** num<100 **then** sum := sum+num⟧ ($sto_{0,22}$, [-1], []))

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

evaluate ⟦num>9⟧ $sto_{0,22}$
= if *greater*(m,n) then *bool*(true)
else *bool*(false)
where *int*(m) = *evaluate* ⟦num⟧ $sto_{0,22}$
and *int*(n) = *evaluate* ⟦9⟧ $sto_{0,22}$
= if *greater*(22,9) then *bool*(true) else *bool*(false)
= *bool*(true)

evaluate ⟦num<100⟧ $sto_{0,22}$
= if *less*(m,n) then *bool*(true) else *bool*(false)
where *int*(m) = *evaluate* ⟦num⟧ $sto_{0,22}$
and *int*(n) = *evaluate* ⟦100⟧ $sto_{0,22}$
= if *less*(22,100) then *bool*(true) else *bool*(false)
= *bool*(true)

evaluate ⟦num>9 **and** num<100⟧ $sto_{0,22}$
= if p then *bool*(q) else *bool*(false)
where *bool*(p) = *evaluate* ⟦num>9⟧ $sto_{0,5}$
and *bool*(q) = *evaluate* ⟦num<100⟧ $sto_{0,5}$
= if true then *bool*(true) else *bool*(false)
= *bool*(true)

This time we execute the **then** clause in the **if** command.

execute ⟦**if** num>9 **and** num<100 **then**
sum := sum+num⟧ ($sto_{0,22}$, [-1], [])
= if p then *execute* ⟦sum := sum+num⟧
($sto_{0,22}$, [-1], [])
else ($sto_{0,22}$, [-1], [])
where *bool*(p) =
evaluate ⟦num>9 **and** num<100⟧ $sto_{0,5}$
= if true then *execute* ⟦sum := sum+num⟧
($sto_{0,22}$, [-1], [])
else ($sto_{0,22}$, [-1], [])
= *execute* ⟦sum := sum+num⟧ ($sto_{0,22}$, [-1], [])

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

evaluate ⟦sum+num⟧ $sto_{0,22}$
= *int*(*plus*(m,n))
where *int*(m) = *evaluate* ⟦sum⟧ $sto_{0,22}$
and *int*(n) = *evaluate* ⟦num⟧ $sto_{0,22}$
= *int*(*plus*(0,22)) = *int*(22)

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

execute ⟦sum := sum+num⟧ ($sto_{0,22}$, [-1], [])
= (*updateSto*($sto_{0,22}$, sum,
(*evaluate* ⟦sum+num⟧ $sto_{0,22}$)), [-1], [])
= (*updateSto*($sto_{0,22}$, sum, *int*(22)), [-1], [])
= ({sum↦*int*(22),num↦*int*(22)}, [-1], [])

Let $sto_{22,22}$ = {sum↦*int*(22),num↦*int*(22)}

Continuing with the body of the **while** command for its second pass yileds a state with store $sto_{22,-1}$ after executing the **read** command.

execute ⟦**read** num⟧ ($sto_{22,22}$, [-1], []))
= (*updateSto*($sto_{22,22}$,num,*int*(-1)), [], [])
= ({sum↦*int*(22),num↦*int*(-1)}, [], [])

Let $sto_{22,-1}$ = {sum↦*int*(22),num↦*int*(-1)}

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

execute ⟦$c_{3.1}$; $c_{3.2}$⟧ ($sto_{0,22}$, [-1], []) =
($sto_{22,-1}$, [], [])

This completes the second pass through loop.

loop ($sto_{0,22}$, [-1], [])
= loop (*execute* ⟦$c_{3.1}$; $c_{3.2}$⟧ ($sto_{0,22}$, [-1], []))
= loop($sto_{22,-1}$, [], [])

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

evaluate ⟦num⟧ $sto_{22,-1}$ = *applySto*($sto_{22,-1}$, num)
= *int*(-1)

evaluate ⟦0⟧ $sto_{22,-1}$ = *int*(0)

evaluate ⟦num>=0⟧ $sto_{22,-1}$
= if *greatereq*(m,n) then *bool*(true)
else *bool*(false)
where *int*(m) = *evaluate* ⟦num⟧ $sto_{22,-1}$
and *int*(n) = *evaluate* ⟦0⟧ $sto_{22,-1}$
= if *greatereq*(-1,0) then *bool*(true)
else *bool*(false)
= *bool*(false)

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

loop ($sto_{22,-1}$, [], [])
= if false then loop(*execute* ⟦$c_{3.1}$; $c_{3.2}$⟧
($sto_{22,-1}$, [], []))
else ($sto_{22,-1}$, [], [])
= ($sto_{22,-1}$, [], [])

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

execute ⟦**while** num>=0 **do** $c_{3.1}$; $c_{3.2}$ ⟧
($sto_{0,5}$, [22,-1], [])
= loop ($sto_{0,5}$, [22,-1], [])
= ($sto_{22,-1}$, [], [])

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

$evaluate$ ⟦sum⟧ $sto_{22,-1}$ = $applySto(sto_{22,-1}, sum)$
= $int(22)$

$execute$ ⟦**write** sum⟧ ($sto_{22,-1}$, [], [])
= ($sto_{22,-1}$, [], $affix$([],val))
where int(val) = $evaluate$ ⟦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⟧ ($emptySto$, [5,22,-1], [])
= ($sto_{22,-1}$, [], [22]))

$meaning$ ⟦**program** sample **is** d
begin c_1 ; c_2 ; c_3 ; c_4 **end**⟧ [5,22,-1]
= [22]

Implementing Denotational Semantics

Semantic functions become Prolog predicates.

$execute$: Command → Store → Store

becomes the predicate

execute(Cmd, Sto, NewSto).

Semantic equations become clauses.

Command Sequencing

$execute$ ⟦C_1 **;** C_2⟧ = $execute$ ⟦C_2⟧ ∘ $execute$ ⟦C_1⟧

becomes

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


If Command

$execute$ ⟦**if** E **then** C_1 **else** C_2⟧ sto =
if p then $execute$ ⟦C_1⟧ sto else $execute$ ⟦C_2⟧ sto
where $bool$(p) = $evaluate$ ⟦E⟧ sto

becomes

```
execute(if(Test,Then,Else),Sto,NewSto) :-
        evaluate(Test,Sto,Val),
        branch(Val,Then,Else,Sto,NewSto).

branch(bool(true),Then,Else,Sto,NewSto) :-
        execute(Then,Sto,NewSto).

branch(bool(false),Then,Else,Sto,NewSto) :-
        execute(Else,Sto,NewSto).
```


Modeling the Store

The store

{ a ↦ int(3), b ↦ int(8), c ↦ $bool$(false) }

is represented by the Prolog structure

sto(a, int(3), sto(b, int(8), sto(c, bool(false), nil))).

Empty store: Prolog atom “nil”.

Auxiliary Functions

```
applySto(sto(Ide,Val,Sto),Ide,Val).

applySto(sto(I,V,Sto),Ide,Val) :-
        applySto(Sto,Ide,Val).

applySto(nil,Ide,undefined) :-
        write('Undefined variable'), nl, abort.
```

```
updateSto(sto(Ide,V,Sto),Ide,Val,
                                sto(Ide,Val,Sto)).

updateSto(sto(I,V,Sto),Ide,Val,sto(I,V,NewSto))
              :- updateSto(Sto,Ide,Val,NewSto).

updateSto(nil,Ide,Val,sto(Ide,Val,nil)).
```

Assignment Command

execute ⟦I **:=** E⟧ sto =
updateSto(sto, I, (*evaluate* ⟦E⟧ sto))

becomes

```
execute(assign(Ide,Exp),Sto,NewSto) :-
          evaluate(Exp,Sto,Val),
          updateSto(Sto,Ide,Val,NewSto).
```


Expressions

evaluate : Expression → Store → EV

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).
```


Compute

```
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,bool(true),bool(true),bool(true)).
compute(and,bool(P),bool(Q),bool(false)).
```


Input and Output

Two Approaches

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

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

execute(write(Exp),Sto,Sto) :-
          evaluate(Exp,Sto,Val), Val=int(M),
          write('Output = '), write(M), nl.
```

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

  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 with interactive IO:

meaning(prog(Dec,Cmd),Sto) :-
execute(Cmd,nil,Sto).

Let the "go" predicate print the results:

…, write('Final Store:'), nl, printSto(Sto).

With denotational input and output:

meaning(prog(Dec,Cmd),In,Out)) :-
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.

Try It: cp ~slonnegr/public/plf/ds .

cp ~slonnegr/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 (**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
   end;
  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 }

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 = *int*(Integer)
+ *bool*(Boolean)
+ *var*(Location)
+ *proc0*(Store → Store)
+ *proc1*(Location → Store → Store)

Operations on Environments:

emptyEnv : Env

∀I∈Identifier, *emptyEnv* I = *unbound*

extendEnv : Env x Identifier x DV → Env

∀X∈Identifier, *extendEnv*(env,I,dval) X =
if X = I then dval else env(X)

applyEnv : Env x Identifier → DV + *unbound*

applyEnv(env,I) = env(I)

Stores

Store = Location → SV + *unused* + *undefined*

Operations on Stores:

emptySto : Store

∀loc∈Location, *emptySto* loc = *unused*

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

∀X∈Location, *updateSto*(sto,loc,val) X =
if X = loc then val else sto(X)

applySto : Store x Location →
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

execute : Cmd → Env → Store → 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 =
execute ⟦C⟧ env_1 sto_1
where (env_1, sto_1) = *elaborate* ⟦D⟧ env sto

elaborate ⟦D_1 D_2⟧ env sto =
elaborate ⟦D_2⟧ env_1 sto_1
where (env_1, sto_1) = *elaborate* ⟦D_1⟧ env sto

elaborate ⟦ε⟧ 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)), sto_1)
where (sto_1, loc) = allocate sto

elaborate ⟦**var** I L **:** T⟧ env sto =
elaborate ⟦**var** L **:** T⟧ env_1 sto_1
where (env_1,sto_1) =
elaborate ⟦**var** I **:** T⟧ env sto

execute ⟦C_1 **;** C_2⟧ env sto =
execute ⟦C_2⟧ env (*execute* ⟦C_1⟧ env sto)
or
execute ⟦C_1 **;** C_2⟧ env =
(*execute* ⟦C_2⟧ env) ∘ (*execute* ⟦C_1⟧ 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** C_1 **else** C_2⟧ env sto =
if p then *execute* ⟦C_1⟧ env sto
else *execute* ⟦C_2⟧ 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 denotable 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.

evaluate ⟦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)

evaluate ⟦N⟧ env sto = *int*(*value* ⟦N⟧)

evaluate ⟦**true**⟧ env sto = *bool*(true)

evaluate ⟦**false**⟧ env sto = *bool*(false)

evaluate ⟦E_1 + E_2⟧ env sto = *int*(*plus*(m,n))
where *int*(m) = *evaluate* ⟦E_1⟧ env sto
and *int*(n) = *evaluate* ⟦E_2⟧ env sto

:

evaluate ⟦E_1 = E_2⟧ env sto = *bool*(*equal*(m,n))
where *int*(m) = *evaluate* ⟦E_1⟧ env sto
and *int*(n) = *evaluate* ⟦E_2⟧ env sto

:

evaluate ⟦E_1 **and** E_2⟧ env sto =
if p then *bool*(q) else *bool*(false)
where *bool*(p) = *evaluate* ⟦E_1⟧ env sto
and *bool*(q) = *evaluate* ⟦E_2⟧ env sto

:

Procedures

elaborate ⟦**procedure** I **is** B⟧ env sto =
(env₁, sto)

where env_1 = *extendEnv*(env,I,*proc0*(proc))

and proc = *perform* ⟦B⟧ env_1

elaborate ⟦**procedure** I_1**(**I_2 **:** T**) is** B⟧ env sto =
(env_1, sto)

where env_1 = *extendEnv*(env,I_1,*proc1*(proc))

and proc loc =
perform ⟦B⟧ *extendEnv*(env_1,I_2,*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 "env_1" 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:

   proc = *perform* ⟦B⟧ env

Procedure Calls

execute $\llbracket$I$\rrbracket$ env sto = proc sto

where *proc0*(proc) = *applyEnv*(env,I)

execute $\llbracket$I(E)$\rrbracket$ env sto =
proc loc
updateSto(sto_1,loc,*evaluate* $\llbracket$E$\rrbracket$ env sto)

where *proc1*(proc) = *applyEnv*(env,I)

and (sto_1,loc) = allocate sto

Example

	Environment
program prfacs **is**	
var n : **integer;**	[n $\mapsto$ *var*(0)]
procedure pf(d:**integer**) **is**	env_1
var q : **integer;**	$env_{2,L}$
begin	
if n>1	$env_{2,L}$
then q := n/d;	$env_{2,L}$
if n=d*q	$env_{2,L}$
then	
write d; n:=q; pf(d)	$env_{2,L}$
else pf(d+1)	$env_{2,L}$
end if	
end if	
end;	
begin n **:=** 20; pf(2) **end**	env_1

where
proc L = *perform* $\llbracket$**var** q:**int; begin if** n>1 **then** …$\rrbracket$
extendEnv(env_1,d,var(L))
env_1= [pf $\mapsto$ *proc1*(proc), n $\mapsto$ *var*(0)]
$env_{2,L}$ = [d $\mapsto$ *var*(L), q $\mapsto$ *var*(L+1),
pf $\mapsto$ *proc1*(proc), n $\mapsto$ *var*(0)]
for L = 1,3,5,7,9,11,13

	Store
var n : **integer**;	{ 0 $\mapsto$ *undef* }
n **:=** 20;	{ 0 $\mapsto$ *int*(20) }
pf(2)	
(d : **integer**)	{0 $\mapsto$ *int*(20),1 $\mapsto$ *int*(2)}
var q : **integer**;	{0 $\mapsto$ *int*(20),1 $\mapsto$ *int*(2),2 $\mapsto$ *undef*}
if n>1	
q **:=** n/d;	{0 $\mapsto$ *int*(20),1 $\mapsto$ *int*(2),2 $\mapsto$ *int*(10)}
if n=d*q	
write d;	
n:=q;	{0 $\mapsto$ *int*(10),1 $\mapsto$ *int*(2),2 $\mapsto$ *int*(10)}
pf(d)	
(d : **integer**)	{0 $\mapsto$ *int*(10),3 $\mapsto$ *int*(2)}
var q : **integer**;	{0 $\mapsto$ *int*(10),3 $\mapsto$ *int*(2),4 $\mapsto$ *undef*}
if n>1	
q **:=** n/d;	{0 $\mapsto$ *int*(10),3 $\mapsto$ *int*(2),4 $\mapsto$ *int*(5)}
if n=d*q	
write d;	
n:=q;	{0 $\mapsto$ *int*(5),3 $\mapsto$ *int*(2),4 $\mapsto$ *int*(5)}
pf(d)	
(d : **integer**)	{0 $\mapsto$ *int*(5),5 $\mapsto$ *int*(2)}
var q : **integer**;	{0 $\mapsto$ *int*(5),5 $\mapsto$ *int*(2),6 $\mapsto$ *undef*}
if n>1	
q **:=** n/d;	{0 $\mapsto$ *int*(5),5 $\mapsto$ *int*(2),6 $\mapsto$ *int*(2)}
if n=d*q	no
pf(d+1)	

(d : **integer**)	{0 $\mapsto$ *int*(5),7 $\mapsto$ *int*(3)}
var q : **integer**;	{0 $\mapsto$ *int*(5),7 $\mapsto$ *int*(3),8 $\mapsto$ *undef*}
if n>1	
q **:=** n/d;	{0 $\mapsto$ *int*(5),7 $\mapsto$ *int*(3),8 $\mapsto$ *int*(1)}
if n=d*q	no
pf(d+1)	
(d : **integer**)	{0 $\mapsto$ *int*(5),9 $\mapsto$ *int*(4)}
var q : **integer**;	{0 $\mapsto$ *int*(5),9 $\mapsto$ *int*(4),10 $\mapsto$ *undef*}
if n>1	
q **:=** n/d;	{0 $\mapsto$ *int*(5),9 $\mapsto$ *int*(4),10 $\mapsto$ *int*(1)}
if n=d*q	no
pf(d+1)	
(d : **integer**)	{0 $\mapsto$ *int*(5),11 $\mapsto$ *int*(5)}
var q : **integer**;	{0 $\mapsto$ *int*(5),11 $\mapsto$ *int*(5),12 $\mapsto$ *undef*}
if n>1	
q **:=** n/d;	{0 $\mapsto$ *int*(5),11 $\mapsto$ *int*(5),12 $\mapsto$ *int*(1)}
if n=d*q	
write d;	
n:=q;	{0 $\mapsto$ *int*(1),11 $\mapsto$ *int*(5),12 $\mapsto$ *int*(1)}
pf(d)	
(d : **integer**)	{0 $\mapsto$ *int*(1),13 $\mapsto$ *int*(5)}
var q : **integer**;	{0 $\mapsto$ *int*(1),13 $\mapsto$ *int*(5),14 $\mapsto$ *undef*}
if n>1	is false causing termination.

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 x Env) → (Env x 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 ⟦**program** I **is** B⟧ =
examine ⟦B⟧ *extendEnv*(*emptyEnv*,I,*program*)

examine ⟦D **begin** C **end**⟧ env =
check ⟦C⟧ env_1
where ($locenv_1$, env_1) =
elaborate ⟦D⟧ (emptyEnv, env)

elaborate ⟦D_1 D_2⟧ =
(*elaborate* ⟦D_2⟧) ∘ (*elaborate* ⟦D_1⟧)

elaborate ⟦ε⟧ (locenv, env) = (locenv, env)

elaborate ⟦**const** I **=** E⟧ (locenv, env) =
if *applyEnv*(locenv,I) = *unbound*
then (*extendEnv*(locenv,I,*typify* ⟦E⟧ env),
extendEnv(env,I,*typify* ⟦E⟧ env)
else *error*

elaborate ⟦**var** I **:** T⟧ (locenv, env)=
if *applyEnv*(locenv,I) = *unbound*
then (*extendEnv*(locenv,I,*type*(T)),
extendEnv(env,I,*type*(T)),
else *error*

elaborate ⟦**var** I L **:** T⟧ =
(*elaborate* ⟦**var** L **:** T⟧) ∘ (*elaborate* ⟦**var** I **:** T⟧)

check ⟦C_1 **;** C_2⟧ env =
(*check* ⟦C_1⟧ env) and (*check* ⟦C_2⟧ env)

check ⟦**skip**⟧ env = true

check ⟦I **:=** E⟧ env =
(*applyEnv*(env,I) = *intvar*
and *typify* ⟦E⟧ env = *integer*)
or
(*applyEnv*(env,I) = *boolvar*
and *typify* ⟦E⟧ env = *boolean*)

check ⟦**if** E **then** C⟧ env =
(*typify* ⟦E⟧ env = *boolean*) and (*check* ⟦C⟧ env)

check ⟦**if** E **then** C_1 **else** C_2⟧ env =
(*typify* ⟦E⟧ env = *boolean*) and
(*check* ⟦C_1⟧ env) and (*check* ⟦C_2⟧ env)

check ⟦**while** E **do** C⟧ env =
(*typify* ⟦E⟧ env = *boolean*) and (*check* ⟦C⟧ env)

check ⟦**declare** B⟧ env = *examine* ⟦B⟧ env

check ⟦**read** I⟧ env = (*applyEnv*(env, I) = *intvar*)

check ⟦**write** E⟧ env = (*typify* ⟦E⟧ env = *integer*)

typify ⟦I⟧ env =
case *applyEnv*(env,I) of
intvar, integer : *integer*
boolvar, boolean : *boolean*
program : *program*
unbound : *error*

typify ⟦N⟧ env = *integer*

typify ⟦**true**⟧ env = *boolean*

typify ⟦**false**⟧ env = *boolean*

typify ⟦E_1 **+** E_2⟧ env =
if (*typify* ⟦E_1⟧ env = *integer*)
and (*typify* ⟦E_2⟧ env = *integer*)
then *integer* else *error*
:

typify ⟦E_1 **and** E_2⟧ env =
if (*typify* ⟦E_1⟧ env = *boolean*)
and (*typify* ⟦E_2⟧ env = *boolean*)
then *boolean* else *error*
:

typify ⟦E_1 **<** E_2⟧ env =
if (*typify* ⟦E_1⟧ env = *integer*)
and (*typify* ⟦E_2⟧ env = *integer*)
then *boolean* else *error*
:

Example (Falsely rejected by version in text)

The program identifier "bug" is ignored to save space.

	locenv	**env**
program bug **is**		
const c = 5;	[c $\mapsto$ *int*]	[c $\mapsto$ *int*]
var k : **integer**;	[k $\mapsto$ *ivar*, c $\mapsto$ *int*]	[k $\mapsto$ *ivar*, c $\mapsto$ *int*]
begin		
k := 99;		[k $\mapsto$ *ivar*, c $\mapsto$ *int*]
declare	[ ]	[k $\mapsto$ *ivar*, c $\mapsto$ *int*]
const d = c+k;	[d $\mapsto$ *int*]	[d $\mapsto$ *int*, k $\mapsto$ *ivar*, c $\mapsto$ *int*]
var m : **integer**;	[m $\mapsto$ *ivar*, d $\mapsto$ *int*]	[m $\mapsto$ *ivar*, d $\mapsto$ *int*, k $\mapsto$ *ivar*, c $\mapsto$ *int*]
begin		
m := c+d+k;		[m $\mapsto$ *ivar*, d $\mapsto$ *int*, k $\mapsto$ *ivar*, c $\mapsto$ *int*]
write m		[m $\mapsto$ *ivar*, d $\mapsto$ *int*, k $\mapsto$ *ivar*, c $\mapsto$ *int*]
end		
end		

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:

$$sto_0 \rightarrow execute [\![C_1]\!] \rightarrow execute [\![C_2]\!] \rightarrow execute [\![C_3]\!] \rightarrow execute [\![C_4]\!] \rightarrow sto_{final}.$$

What if "C_3" is "**if** x>0 **then goto** L_1 **else skip**"?

Store Transformation if x>0:

$$sto_0 \rightarrow execute [\![C_1]\!] \rightarrow execute [\![C_2]\!] \rightarrow execute [\![C_3]\!] \rightarrow execute [\![C_1]\!] \rightarrow \text{etc.}$$

"*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≤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 → Store

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

Labels are bound to continuations in the environment.

Identifier	**Denotable Value**
L_1	$cont_1$ = *execute* $[\![C_1; C_2; C_3; C_4]\!]$ env
L_2	$cont_2$ = *execute* $[\![C_2; C_3; C_4]\!]$ env
L_3	$cont_3$ = *execute* $[\![C_3; C_4]\!]$ env
L_4	$cont_4$ = *execute* $[\![C_4]\!]$ env

Continutations 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.

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

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 → 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 ⟦ L_1**:**C_1**;** L_2**:**C_2**;** … **;** L_n**:**C_n ⟧ env cont =
cont_1
where cont_1 = *execute* ⟦C_1⟧ env_1 cont_2
cont_2 = *execute* ⟦C_2⟧ env_1 cont_3
…
cont_n = *execute* ⟦C_n⟧ env_1 cont
and env_1 =
extendEnv(env,[L_1, … , L_n],[cont_1, … , cont_n])

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** S_1 **else** S_2 ⟧ env cont sto =
if p then *perform* ⟦S_1⟧ env cont sto
else *perform* ⟦S_2⟧ env cont sto
where *bool*(p) = *evaluate* ⟦E⟧ sto

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 ⟦C_1 **;** C_2⟧ env cont sto =
execute ⟦C_1⟧ env {*execute* ⟦C_2⟧ 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 ⟦E_1 + E_2⟧ sto = *int*(*plus*(m,n))
where *int*(m) = *evaluate* ⟦E_1⟧ sto
and *int*(n) = *evaluate* ⟦E_2⟧ 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** S_1 **else** S_2 ⟧ env cont sto =
if *evaluate* ⟦E⟧ sto=*error*
then *errCont* sto
else if p
then *perform* ⟦S_1⟧ env cont sto
else *perform* ⟦S_2⟧ env cont sto
where *bool*(p) = *evaluate* ⟦E⟧ sto