Principles of Programming Languages

Module M09: Denotational Semantics of Imperative Languages

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

March 23, 28 & 30 April 04 & 06, 2022

Table of Contents

Imperative Languages

Imperative Languages

- Most sequential programming languages use a *Data Structure* that exists independently of any program in the language
- The data structure is not explicitly mentioned in the language's syntax, but it is possible to build phrases that access it and update it
- This data structure is called the *Store*, and languages that utilize stores are called *Imperative*
- Fundamental *Store*'s are:
  - Primary memory
  - File stores, and
  - Databases

Imperative Languages: Stores

- The *Store* and a *Program* share an intimate relationship:
  - The store is critical to the evaluation of a *Phrase* in a program
  - A phrase is understood in terms of
    - how it handles the store, and
    - the absence of a proper store makes the phrase non-executable
  - The store serves as a means of communication between the different phrases in the program
    - Values computed by one phrase are deposited in the store so that another phrase may use them
    - The language's sequencing mechanism establishes the order of communication
  - The store is an inherently *large* argument
  - Only one copy of store exists at any point during the evaluation
  - We use lifted domains to model the *Store*

Language + Assignment

Example Language with Assignment: Command

- A declaration-free Pascal subset
- A program in the language is a sequence of commands
- Stores belong to the domain *Store* and serve as arguments to the valuation function:

$$\mathbf{C} : \mathit{Command} \rightarrow \mathit{Store}_{\perp} \rightarrow \mathit{Store}_{\perp}$$

- The purpose of a command is to produce a new store from its store argument
- A command might not terminate its actions upon the store – it can *loop*
- The looping of a command $[[C]]$ with store s has semantics $\mathbf{C}[[C]]s = \perp$
  - *Store* is lifted to $\mathit{Store}_{\perp}$
- Command Composition is:

$$\mathbf{C}[[C_1; C_2]] = \mathbf{C}[[C_2]] \circ \mathbf{C}[[C_1]]$$

Example Language with Assignment: Abstract Syntax

Abstract Syntax:

- Consider the entities as:

$$P \in Program$$
$$C \in Command$$
$$E \in Expression$$
$$B \in Boolean\_expr$$
$$I \in Identifier$$
$$N \in Numeral$$

$$P ::= C.$$
$$C ::= C_1;\ C_2 \mid \textbf{if}\ B\ \textbf{then}\ C \mid \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2 \mid$$
$$I := E \mid \textbf{diverge}$$
$$E ::= E_1 + E_2 \mid I \mid N$$
$$B ::= E_1 = E_2 \mid \neg B$$

diverge is a non-terminating command

Semantic Algebras:

- *Truth values*

  Domain: $t \in Tr = B$

  Operations:

  $true, false : Tr$

  $not : Tr \rightarrow Tr$

- *Identifiers*

  Domain: $i \in Id = Identifier$

- *Natural Numbers*

  Domain: $n \in Nat = \mathcal{N}$

  Operations:

  $zero, one, \ldots : Nat$

  $plus : Nat \times Nat \rightarrow Nat$

  $equals : Nat \times Nat \rightarrow Tr$

Example Language with Assignment: Semantic Algebras

- *Store*

  Domain: $s \in Store = Id \rightarrow Nat$

  Operations:

  $newstore : Store$

  $\quad newstore = \lambda i.zero$

  $access : Id \rightarrow Store \rightarrow Nat$

  $\quad access = \lambda i.\lambda s.s(i)$

  $update : Id \rightarrow Nat \rightarrow Store \rightarrow Store$

  $\quad update = \lambda i.\lambda n.\lambda s.[i \mapsto n]s$

Example Language with Assignment: Valuation Functions

Valuation Functions:

- $\mathbf{P} : \mathit{Program} \to \mathit{Nat} \to \mathit{Nat}_\perp$

  $\mathbf{P}[[P]] = \mathbf{P}[[C.]] =?$

  where the input number n is associated with identifier $[[A]]$ in a new store. As the program body is evaluated, and the answer is extracted from the store at $[[Z]]$

- $\mathbf{C} : \mathit{Command} \to \mathit{Store}_\perp \to \mathit{Store}_\perp$

  $\mathbf{C}[[C_1;\ C_2]] =?$

  $\mathbf{C}[[\mathbf{if}\ B\ \mathbf{then}\ C]] =?$

  $\mathbf{C}[[\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]] =?$

  $\mathbf{C}[[I := E]] =?$

  $\mathbf{C}[[\mathbf{diverge}]] =?$

Example Language with Assignment: Valuation Functions

- The clauses of the **C** function are all strict in their use of the store
- Command composition works as discussed earlier
- The conditional commands are choice functions
- The expression $(e_1 \rightarrow e_2 \; [] \; e_3)$ is non-strict in arguments e_2 and e_3 – the value of $\mathbf{C}[[\mathbf{if}\ B\ \mathbf{then}\ C]]s$ is s when $\mathbf{B}[[B]]s$ is *false*, even if $\mathbf{C}[[C]]s = \perp$
- The assignment statement performs the expected *update*
- The [[**diverge**]] command causes non-termination

Example Language with Assignment: Valuation Functions

- $\mathbf{E} : \mathit{Expression} \rightarrow \mathit{Store} \rightarrow \mathit{Nat}$
  $\mathbf{E}[[E_1 + E_2]] = ?$
  $\mathbf{E}[[I]] = ?$
  $\mathbf{E}[[N]] = ?$
- $\mathbf{B} : \mathit{Boolean\_expr} \rightarrow \mathit{Store} \rightarrow \mathit{Tr}$
  $\mathbf{B}[[E_1 = E_2]] = ?$
  $\mathbf{B}[[\neg B]] = ?$
- $\mathbf{N} : \mathit{Numeral} \rightarrow \mathit{Nat}$ (maps numeral $\mathcal{N}$ to corresponding $n \in \mathit{Nat}$)

Example Language with Assignment: Valuation Functions

- The **E** function also needs a store argument, but the store is used in a *read only* mode
- **E**'s functionality shows that an expression produces a number, not a new version of store; the store is not updated by an expression
- The equation for addition is stated so that the order of evaluation of $[[E_1]]$ and $[[E_2]]$ is not important to the final answer. Indeed, the two expressions might even be evaluated in parallel
- A strictness check of the store is not needed, because **C** has already verified that the store is proper prior to passing it to **E**

Example Language with Assignment: Valuation Functions

Valuation Functions:

- $\mathbf{P} : Program \to Nat \to Nat_\perp$

  $\mathbf{P}[[C.]] = \lambda n.let\ s = (update\ [[A]]\ n\ newstore)\ in$
  $\qquad let\ s' = \mathbf{C}[[C]]s\ in\ (access\ [[Z]]\ s')$

  where the input number n is associated with identifier $[[A]]$ in a new store. As the program body is evaluated, and the answer is extracted from the store at $[[Z]]$

- $\mathbf{C} : Command \to Store_\perp \to Store_\perp$

  $\mathbf{C}[[C_1;\ C_2]] = \underline{\lambda} s.\mathbf{C}[[C_2]](\mathbf{C}[[C_1]]s)$
  $\mathbf{C}[[\mathbf{if}\ B\ \mathbf{then}\ C]] = \underline{\lambda} s.\mathbf{B}[[B]]s \to \mathbf{C}[[C]]s\ []\ s$
  $\mathbf{C}[[\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]] =$
  $\qquad \underline{\lambda} s.\mathbf{B}[[B]]s \to \mathbf{C}[[C_1]]s\ []\ \mathbf{C}[[C_2]]s$
  $\mathbf{C}[[I := E]] = \underline{\lambda} s.update[[I]]\ (\mathbf{E}[[E]]s)\ s$
  $\mathbf{C}[[\mathbf{diverge}]] = \underline{\lambda} s.\perp$

Example Language with Assignment: Valuation Functions

- $\mathbf{E} : Expression \rightarrow Store \rightarrow Nat$
  $\mathbf{E}[[E_1 + E_2]] = \lambda s.\mathbf{E}[[E_1]]s\ plus\ \mathbf{E}[[E_2]]s$
  $\mathbf{E}[[I]] = \lambda s.access\ [[I]]\ s$
  $\mathbf{E}[[N]] = \lambda s.\mathbf{N}[[N]]$
- $\mathbf{B} : Boolean\_expr \rightarrow Store \rightarrow Tr$
  $\mathbf{B}[[E_1 = E_2]] = \lambda s.\mathbf{E}[[E_1]]s\ equals\ \mathbf{E}[[E_2]]s$
  $\mathbf{B}[[\neg B]] = \lambda s.not(\mathbf{B}[[B]]s)$
- $\mathbf{N} : Numeral \rightarrow Nat\ (omitted)$

Example Language with Assignment: Example Program Workout 1

- $\mathbf{P}[[Z := 1; \textbf{if } A = 0 \textbf{ then diverge}; \; Z := 3.]](\mathit{two})$

Example Language with Assignment: Example Program Workout 1

- $\mathbf{P}[[Z := 1; \textbf{if } A = 0 \textbf{ then diverge}; \; Z := 3.]](two)$
  $= let\ s = (update\ [[A]]\ two\ newstore)\ in$
  $let\ s' = \mathbf{C}[[Z := 1; \textbf{if } A = 0 \textbf{ then diverge}; \; Z := 3]]s$
  $in\ (access\ [[Z]]\ s')$

  $let\ s' = \mathbf{C}[[Z := 1; \textbf{if } A = 0 \textbf{ then diverge}; Z := 3]]([\ [[A]] \mapsto two]\ newstore)$
  $in\ access\ [[Z]]\ s'$

Example Language with Assignment: Example Program Workout 1

- $\mathbf{P}[[Z := 1; \mathbf{if}\ A = 0\ \mathbf{then\ diverge};\ Z := 3.]](two)$
  $= let\ s = (update\ [[A]]\ two\ newstore)\ in$
  $\quad let\ s' = \mathbf{C}[[Z := 1; \mathbf{if}\ A = 0\ \mathbf{then\ diverge};\ Z := 3]]s$
  $\quad in\ (access\ [[Z]]\ s')$

  $let\ s' = \mathbf{C}[[Z := 1; \mathbf{if}\ A = 0\ \mathbf{then\ diverge}; Z := 3]]([\ [[A]] \mapsto two]\ newstore)$
  $in\ access\ [[Z]]\ s'$

Example Language with Assignment: Example Program Workout 1

- $\mathbf{C}[[Z := 1; \mathbf{if}\ A = 0\ \mathbf{then\ diverge}; Z := 3]]s_1,\ s_1 = [\ [[A]] \mapsto two]\ newstore$
  $= (\underline{\lambda} s.\mathbf{C}[[\mathbf{if}\ A = 0\ \mathbf{then\ diverge}; Z := 3]](\mathbf{C}[[Z := 1]]s))s_1$
  $\mathbf{C}[[\mathbf{if}\ A = 0\ \mathbf{then\ diverge}; Z := 3]](\mathbf{C}[[Z := 1]]s_1)$

  $\mathbf{C}[[Z := 1]]s_1 = (\underline{\lambda} s.update\ [[Z]]\ (E[[1]]s)\ s)s_1$
  $= update\ [[Z]]\ (\mathbf{E}[[1]]s_1)s_1 = update\ [[Z]]\ (\mathbf{N}[[1]])s_1 = update\ [[Z]]\ one\ s_1$
  $= [\ [[Z]] \mapsto one][\ [[A]] \mapsto two]\ newstore = s_2$

  $\mathbf{C}[[\mathbf{if}\ A = 0\ \mathbf{then\ diverge}; Z := 3]]s_2$
  $= (\underline{\lambda} s.\mathbf{C}[[Z := 3]]((\underline{\lambda} s.\mathbf{B}[[A = 0]]s \rightarrow \mathbf{C}[[\mathbf{diverge}]]s\ []\ s)s))s_2$
  $= \mathbf{C}[[Z := 3]]((\underline{\lambda} s.\mathbf{B}[[A = 0]]s \rightarrow \mathbf{C}[[\mathbf{diverge}]]s\ []\ s)s_2$
  $= \mathbf{C}[[Z := 3]](\mathbf{B}[[A = 0]]s_2 \rightarrow \mathbf{C}[[\mathbf{diverge}]]s_2\ []\ s_2)$

Example Language with Assignment: Example Program Workout 1

- $\mathbf{B}[[A = 0]]s_2$
  $= (\lambda s.\mathbf{E}[[A]]s \; \mathit{equals} \; \mathbf{E}[[0]]s)s_2 = \mathbf{E}[[A]]s_2 \; \mathit{equals} \; \mathbf{E}[[0]]s_2$
  $= (\mathit{access} \; [[A]] \; s_2) \; \mathit{equals} \; \mathit{zero}$

  $\mathit{access} \; [[A]] \; s_2$
  $= s_2[[A]] = ([\; [[Z]] \mapsto \mathit{one}][\; [[A]] \mapsto \mathit{two}] \; \mathit{newstore})[[A]]$
  $= ([\; [[A]] \mapsto \mathit{two}] \; \mathit{newstore})[[A]] \; (\mathit{why}?)$
  $= \mathit{two}$

- Thus, $\mathbf{B}[[A = 0]]s_2 = \mathit{false}$, implying that $\mathbf{C}[[\mathbf{if} \; A = 0 \; \mathbf{then} \; \mathbf{diverge}]]s_2 = s_2$. Now:
  $\mathbf{C}[[Z := 3]]s_2 = [\; [[Z]] \mapsto \mathit{three}]s_2$
  $\mathit{let} \; s' = [\; [[Z]] \mapsto \mathit{three}]s_2 \; \mathit{in} \; \mathit{access} \; [[Z]]s'$
  $= \mathit{access} \; [[Z]][\; [[Z]] \mapsto \mathit{three}]s_2 = ([\; [[Z]] \mapsto \mathit{three}]s_2)[[Z]]$
  $= \mathit{three}$

Example Language with Assignment: Example Program Workout 2

- $\mathbf{P}[[Z := 1; \mathbf{if}\ A = 0\ \mathbf{then}\ \mathbf{diverge};\ Z := 3.]](zero)$
  $= let\ s' = \mathbf{C}[[Z := 1; \mathbf{if}\ A = 0\ \mathbf{then}\ \mathbf{diverge};\ Z := 3]]s_3\ in\ access\ [[Z]]\ s'$
  where $s_3\ = [\ [[A]] \mapsto zero]\ newstore$

- $\mathbf{C}[[Z := 1;\ \mathbf{if}\ A = 0\ \mathbf{then}\ \mathbf{diverge};\ Z := 3]]s_3$
  $= \mathbf{C}[[\mathbf{if}\ A = 0\ \mathbf{then}\ \mathbf{diverge};\ Z := 3]]s_4$
  where $s_4 = [\ [[Z]] \mapsto one]s_3$

- $\mathbf{B}[[A = 0]]s_4 \to \mathbf{C}[[\mathbf{diverge}]]s_4\ []\ s_4$
  $= true \to \mathbf{C}[[\mathbf{diverge}]]s_4\ []\ s_4$
  $= \mathbf{C}[[\mathbf{diverge}]]s_4$
  $= (\underline{\lambda} s.\bot)s_4$
  $= \bot$

- $\mathbf{P} = let\ s' = \bot\ in\ access\ [[Z]]\ s'$
  $= \bot$

Example Language with Assignment: Equivalence of Stores

Prove that:

$$\mathbf{C}[[X := 0; Y := X + 1]]s = \mathbf{C}[[Y := 1; X := 0]]s$$

That is, these programs are equivalent.

Example Language with Assignment

Prove that:

$$\mathbf{C}[[X := 0; Y := X + 1]]s = \mathbf{C}[[Y := 1; X := 0]]s$$

That is, these programs are equivalent.

- $\mathbf{C}[[X := 0; Y := X + 1]]s$
  $= \mathbf{C}[[Y := X + 1]](\mathbf{C}[[X := 0]]s)$
  $= \mathbf{C}[[Y := X + 1]]([\ [[X]] \mapsto \mathit{zero}]s)$
  $= \mathit{update}\ [[Y]]\ (\mathbf{E}[[X + 1]]\ ([\ [[X]] \mapsto \mathit{zero}]s))([\ [[X]] \mapsto \mathit{zero}]s)$
  $= \mathit{update}\ [[Y]]\ \mathit{one}\ [\ [[X]] \mapsto \mathit{zero}]s$
  $= [\ [[Y]] \mapsto \mathit{one}]\ [\ [[X]] \mapsto \mathit{zero}]s = s_1$
- $\mathbf{C}[[Y := 1; X := 0]]s$
  $= \mathbf{C}[[X := 0]]\ (\mathbf{C}[[Y := 1]]s)$
  $= \mathbf{C}[[X := 0]]\ ([\ [[Y]] \mapsto \mathit{one}]s)$
  $= [\ [[X]] \mapsto \mathit{zero}]\ [\ [[Y]] \mapsto \mathit{one}]s = s_2$
- Are they the s_1 and s_2 the same store?

Example Language with Assignment

- Are they the $s_1 = [\ [[Y]] \mapsto one]\ [\ [[X]] \mapsto zero]s$ and $s_2 = [\ [[X]] \mapsto zero]\ [\ [[Y]] \mapsto one]s$ the same store?
- The argument is $[[X]]$: then
  $s_1[[X]] = ([[[Y]] \mapsto one][[[X]] \mapsto zero]s)[[X]]$
  $= ([[[X]] \mapsto zero]s)[[X]] = zero$; and
  $s_2[[X]] = ([[[X]] \mapsto zero][[[Y]] \mapsto one]s)[[X]] = zero$
- The argument is $[[Y]]$: then
  $s_1[[Y]] = ([[[Y]] \mapsto one][[[X]] \mapsto zero]s)[[Y]] = one$; and
  $s_2[[Y]] = ([[[X]] \mapsto zero][[[Y]] \mapsto one]s)[[Y]]$
  $= ([[[Y]] \mapsto one]s)[[Y]] = one$
- The argument is some identifier $[[I]]$ other than $[[X]]$ or $[[Y]]$:
  $s_1[[I]] = s[[I]]$ and $s_2[[I]] = s[[I]]$

Programs Are Functions

Programs Are Functions

Consider again the example:

$$[[Z := 1;\ \ if\ \ A = 0\ \ then\ diverge;\ \ Z := 3]]$$

What is its meaning?

- It is a function: $Nat \rightarrow Nat_{\perp}$
- Its meaning is:

$$\lambda n.n\ equals\ zero \rightarrow \perp\ []\ three$$

Prove.

Programs Are Functions

Consider again the example: $[[Z := 1;\ \mathit{if}\ A = 0\ \mathit{then}\ \mathit{diverge};\ Z := 3]]$. What is its meaning? It is a function: $Nat \rightarrow Nat_{\perp}$

- $\mathbf{P}[[Z := 1; \mathbf{if}\ A = 0\ \mathbf{then\ diverge};\ Z := 3.]]$
  $= \lambda n.\ let\ s = update\ [[A]]\ n\ newstore\ in$
  $let\ s' = \mathbf{C}[[Z := 1; \mathbf{if}\ A = 0\ \mathbf{then\ diverge};\ Z := 3]]s\ in\ access[[Z]]\ s'$
  $= \lambda n.let\ s = update\ [[A]]\ n\ newstore\ in$
  $let\ s' = (\underline{\lambda} s.(\underline{\lambda} s.C[[Z := 3]](C[[\mathbf{if}\ A = 0\ \mathbf{then\ diverge}]]s))s)(C[[Z := 1]]s)$
  $in\ access[[Z]]\ s'$
  $= \lambda n.let\ s = update\ [[A]]\ n\ newstore\ in$
  $let\ s' = (\underline{\lambda} s.(\underline{\lambda} s.update\ [[Z]]\ three\ s)$
  $((\underline{\lambda} s.(access\ [[A]]s)\ equals\ zero \rightarrow (\underline{\lambda} s.\perp)s[]s)s))$
  $((\underline{\lambda} s.update\ [[Z]]\ one\ s)s)\ in\ access\ [[Z]]s'$
  which can be restated as: $\lambda n.let\ s = update\ [[A]]\ n\ newstore\ in$
  $let\ s' = (lets'_1 = update\ [[Z]]\ one\ s\ in$
  $let\ s'_2 = (access\ [[A]]\ s'_1)\ equals\ zero \rightarrow (\underline{\lambda} s.\perp)s'1\ []\ s'_1$
  $in\ update[[Z]]\ three\ s'_2)\ in\ access\ [[Z]]s'$

Programs Are Functions

- $\lambda n.let\ s' = (let\ s'_1\ = update[[Z]]\ one\ s_0\ in$
  $let\ s'_2 = (access\ [[A]]\ s'_1)\ equals\ zero \rightarrow (\underline{\lambda} s.\bot)s'_1\ []\ s'1\ in\ update\ [[Z]]\ three\ s'_2)$
  $in\ access\ [[Z]]\ s'$

  $(access\ [[A]]\ s_1)\ equals\ zero \rightarrow \bot\ []\ s_1 = n\ equals\ zero \rightarrow \bot\ []\ s_1$

  The conditional can be simplified no further. We can make use of the following property; "for $e_2\ \in Store_\bot$ s.t. $e_2 \neq \bot$, $let\ s\ = (e_1\ \rightarrow \bot[]e_2)\ in\ e_3\ equals\ e_1 \rightarrow \bot[][e_2/s]e_3$"

  $let\ s'_2 = (n\ equals\ zero\ \rightarrow \bot\ []\ s_1)\ in\ update\ [[Z]]\ three\ s'_2$
  $= n\ equals\ zero \rightarrow \bot\ []\ update\ [[Z]]\ three\ s_1$

  $\lambda n.let\ s' = (n\ equals\ zero \rightarrow \bot\ []\ update\ [[Z]]\ three\ s_1)\ in\ access\ [[Z]]s'$

  $\lambda n.n\ equals\ zero \rightarrow \bot\ []\ access[[Z]]\ (update\ [[Z]]\ three\ s_1)$

  $\lambda n.n\ equals\ zero \rightarrow \bot\ []\ three$

Interactive File Editor

Interactive File Editor

Interactive File Editor

- The language is an interactive file editor
- A file is a list of records, where the domain of records is taken as primitive
- The file editor makes use of two levels of store
  - the primary store is a component holding the file being edited upon by the user (has a current record marker), and
  - the secondary store is a system of text files indexed by their names
- The edit process:
  - **Load** a file (identified by name) from secondary store to primary store. This initializes the current record to the first record of the file. This is skipped for new files
  - **Edit** the file in the primary store (*forward* / *rewind* – move current record marker forward or reverse, *insert* / *delete* record). Alternately, the editor may *Create* a new file and start editing
  - **Save** the file from primary store to secondary store

Interactive File Editor: Abstract Syntax

Abstract Syntax:

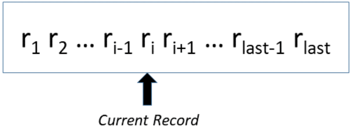
- Consider the entities as:

$$P \in \mathit{Program\_session}$$
$$S \in \mathit{Command\_sequence}$$
$$C \in \mathit{Command}$$
$$R \in \mathit{Record}$$
$$B \in \mathit{Boolean\_expr}$$
$$I \in \mathit{Identifier}$$

$$P ::= \textbf{edit}\ I\ \textbf{cr}\ S$$
$$S ::= C\ \textbf{cr}\ S \mid \textbf{quit}$$
$$C ::= \textbf{newfile} \mid \textbf{moveforward} \mid \textbf{moveback} \mid \textbf{insert}\ R \mid \textbf{delete}$$

An Interactive File Editor: Openfile Representation

- The edited files are values from the *Openfile* domain
- An opened file $r_1, r_2, \cdots, r_{last}$ is represented by two lists of text records; the lists break the file open in the middle:



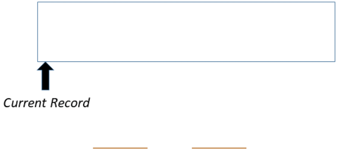
$\underline{r_{i-1} \dots r_2 r_1}$ $\quad$ $\underline{r_i r_{i+1} \dots r_{last}}$

where r_i is the *current* record of the opened file

Note how the first list is written in the reverse order

An Interactive File Editor: newfile

- *newfile* represents a file with no records



$newfile : Openfile$

$newfile = (nil, nil)$

An Interactive File Editor: copyin

- *copyin* takes a file from the file system and organizes it as:

$$\boxed{r_1\ r_2\ \ldots\ r_{i\text{-}1}\ r_i\ r_{i+1}\ \ldots\ r_{last\text{-}1}\ r_{last}}$$

Current Record

$$\underline{\qquad}\qquad \underline{r_1 r_2 \ldots r_{last}}$$

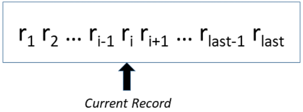
where r_1 is the *current* record of the opened file

$copyin : File \rightarrow Openfile$

$copyin = \lambda f.(nil, f)$

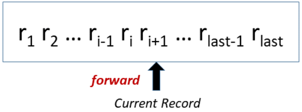
An Interactive File Editor: forward

- The *forwards* operation makes the record *following* the current record the new current record. Pictorially, for:



$\underline{r_{i-1} \ldots r_2 r_1}$ $\underline{r_i r_{i+1} \ldots r_{last}}$

a forwards move produces:



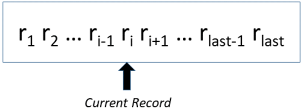
$\underline{r_i r_{i-1} \ldots r_2 r_1}$ $\underline{r_{i+1} \ldots r_{last}}$

$forwards : Openfile \rightarrow Openfile$

$forwards = \lambda(front, back).null\ back \rightarrow (front, back)\ []\ ((hd\ back)\ cons\ front, (tl\ back))$

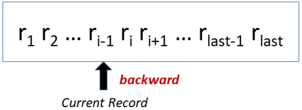
An Interactive File Editor: backward

- The *backwards* operation makes the record *preceding* the current record the new current record. Pictorially, for:



$\underline{r_{i-1} r_{i-2} \dots r_2 r_1}$ $\quad$ $\underline{r_i r_{i+1} \dots r_{last}}$

a backward move produces:



$\underline{r_{i-2} \dots r_2 r_1}$ $\quad$ $\underline{r_{i-1} r_i r_{i+1} \dots r_{last}}$

$backwards : Openfile \to Openfile$

$backwards = \lambda(front, back).null\ front \to (front, back)\ []\ (tl\ front, (hd\ front)\ cons\ back)$

An Interactive File Editor: insert R

- *insert* places a record R *after the current record*. Pictorially, for:

$$\boxed{r_1\ r_2\ \ldots\ r_{i-1}\ r_i\ r_{i+1}\ \ldots\ r_{last-1}\ r_{last}}$$

Current Record

$$\underline{r_{i-1} \ldots r_2 r_1} \qquad \underline{r_i r_{i+1} \ldots r_{last}}$$

an insertion of record R produces:

$$\boxed{r_1\ r_2\ \ldots\ r_{i-1}\ r_i\ R\ r_{i+1}\ \ldots\ r_{last-1}\ r_{last}}$$

Insert R

Current Record

$$\underline{r_i \ldots r_2 r_1} \qquad \underline{R\ \ r_{i+1} \ldots r_{last}}$$

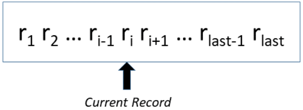
The newly inserted record becomes *current*

$insert : Record \times Openfile \rightarrow Openfile$

$insert = \lambda(r, (front, back)).null\ back \rightarrow (front, r\ cons\ back)$
$[]\ ((hd\ back)\ cons\ front), r\ cons\ (tl\ back))$

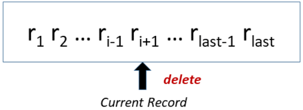
An Interactive File Editor: delete

- *delete* removes the *current record*. Pictorially, for:



$\underline{r_{i-1} \ldots r_2 r_1}$ $\underline{r_i r_{i+1} \ldots r_{last}}$

deletion produces:



$\underline{r_{i-1} \ldots r_2 r_1}$ $\underline{r_{i+1} \ldots r_{last}}$

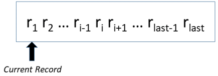
The record following the deleted record becomes *current*

$delete : Openfile \rightarrow Openfile$

$delete = \lambda(front, back).(front, (null\ back \rightarrow back\ []\ tl\ back))$

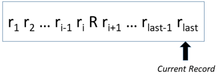
An Interactive File Editor: Test Operations

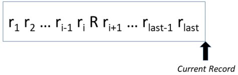
- Check if the first record in the file is current (*at_first_record*),



$$at\_first\_record : Openfile \to Tr$$
$$at\_first\_record = \lambda(front, back).null\ front$$

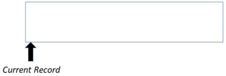
- Check if the last record in the file is current (*at_last_record*)





$$at\_last\_record : Openfile \to Tr$$
$$at\_last\_record = \lambda(front, back).null\ back \to true\ [\!]\ (null\ (tl\ back) \to true\ [\!]\ false)$$

- Check if the file is empty (*isempty*)



$$isempty : Openfile \to Tr$$
$$isempty = \lambda(front, back).(null\ front)\ and\ (null\ back)$$

An Interactive File Editor: copyout

- An open file $r_1, r_2, \cdots, r_{last}$ in the *Openfile* domain:

  $r_1\ r_2 \ldots r_{i-1}\ r_i\ r_{i+1} \ldots r_{last-1}\ r_{last}$

  Current Record

  $\underline{r_{i-1} \ldots r_2 r_1}$ $\underline{r_i r_{i+1} \ldots r_{last}}$

  needs to be written back to File System. *copyout* is the operation for it which should convert it to:

  $r_1\ r_2 \ldots r_{i-1}\ r_i\ r_{i+1} \ldots r_{last-1}\ r_{last}$

  Current Record

  $\underline{\quad\quad}$ $\underline{r_1 r_2 \ldots r_{last}}$

  and then write back:

  $copyout : Openfile \rightarrow File$

  $copyout = \lambda p.$ "appends $fst(p)$ to $snd(p)$ – defined later" // Recursive

  $copyout = \lambda(front, back).null\ front \rightarrow back\ []\ copyout((tl\ front), ((hd\ front)\ cons\ back))$

Interactive File Editor: Semantic Algebra

- *Truth Values*
  Domain: $t \in Tr$
  Operations:
  $true, false : Tr = \mathcal{B}$
  $and : Tr \times Tr \to Tr$
- *Identifiers*: Domain: $i \in Id = Identifier$
- *Text records*: Domain: $r \in Record$
- *Text file*: Domain: $f \in File = Record^*$
- *File System*
  Domain: $s \in File\_system = Id \to File$
  Operations:
  $access : Id \times File\_system \to File$
  $access = \lambda(i, s).s(i)$
  $update : Id \times File \times File\_system \to File\_system$
  $update = \lambda(i, f, s).[i \mapsto f]s$

Interactive File Editor: Semantic Algebra

- *Open file*

  Domain: $p \in Openfile = Record^* \times Record^*$

  Operations:

  $newfile : Openfile$

  $newfile = (nil, nil)$

  $copyin : File \rightarrow Openfile$

  $copyin = \lambda f.(nil, f)$

  $copyout : Openfile \rightarrow File$

  $copyout = \lambda p.$ "appends $fst(p)$ to $snd(p)$ – defined later" // Recursive

  $copyout = \lambda(front, back).null\ front \rightarrow back\ []\ copyout((tl\ front),\ ((hd\ front)\ cons\ back))$

  $forwards : Openfile \rightarrow Openfile$

  $forwards = \lambda(front, back).null\ back \rightarrow (front, back)\ []\ ((hd\ back)\ cons\ front, (tl\ back))$

  $backwards : Openfile \rightarrow Openfile$

  $backwards = \lambda(front, back).null\ front \rightarrow (front, back)\ []\ (tl\ front, (hd\ front)\ cons\ back)$

  $insert : Record \times Openfile \rightarrow Openfile$

  $insert = \lambda(r, (front, back)).null\ back \rightarrow (front, r\ cons\ back)$
  $[]\ ((hd\ back)\ cons\ front), r\ cons\ (tl\ back))$

  $delete : Openfile \rightarrow Openfile$

  $delete = \lambda(front, back).(front, (null\ back \rightarrow back\ []\ tl\ back))$

Interactive File Editor: Semantic Algebra

- *Open file* (Contd.)
  Operations:
  $at\_first\_record : Openfile \to Tr$
  $at\_first\_record = \lambda(front, back).null\ front$
  $at\_last\_record : Openfile \to Tr$
  $at\_last\_record = \lambda(front, back).null\ back \to true\ [\!]\ (null\ (tl\ back) \to true\ [\!]\ false)$
  $isempty : Openfile \to Tr$
  $isempty = \lambda(front, back).(null\ front)\ and\ (null\ back)$
- *Character String*
  Domain: $String$ = the strings formed from the elements of $\mathcal{C}$ (including an *error* string)
  Operations:
  $A, B, C, ..., Z : String$
  $empty : String$
  $error : String$
  $concat : String \times String \to String$
  $length : String \to Nat$
  $substr : String \times Nat \times Nat \to String$
- *Output terminal log*: Domain: $l \in Log = String^*$

Interactive File Editor: Valuation Functions

Valuation Functions:

- $\mathbf{P} : \mathit{Program\_session} \rightarrow \mathit{File\_system} \rightarrow (\mathit{Log} \times \mathit{File\_system})$
  $\mathbf{P}[[\mathbf{edit}\ \mathit{I}\ \mathbf{cr}\ \mathit{S}]] =?$
- $\mathbf{S} : \mathit{Command\_sequence} \rightarrow \mathit{Openfile} \rightarrow (\mathit{Log} \times \mathit{Openfile})$
  $\mathbf{S}[[\mathit{C}\ \mathbf{cr}\ \mathit{S}]] =?$
  $\mathbf{S}[[\mathbf{quit}]] =?$
- $\mathbf{C} : \mathit{Command} \rightarrow \mathit{Openfile} \rightarrow (\mathit{String} \times \mathit{Openfile})$
  $\mathbf{C}[[\mathbf{newfile}]] =?$
  $\mathbf{C}[[\mathbf{moveforward}]] =?$
  $\mathbf{C}[[\mathbf{moveback}]] =?$
  $\mathbf{C}[[\mathbf{insert}\ \mathit{R}]] =?$
  $\mathbf{C}[[\mathbf{delete}]] =?$

Interactive File Editor: Valuation Functions

Valuation Functions:

- $\mathbf{P} : \mathit{Program\_session} \to \mathit{File\_system} \to (\mathit{Log} \times \mathit{File\_system})$
  $\mathbf{P}[[\mathbf{edit}\ I\ \mathbf{cr}\ S]]$
  $= \lambda s.let\ p = copyin(access([[I]], s))\ in$
  $(''edit\ I''\ cons\ fst(\mathbf{S}[[S]]p),$
  $update([[I]], copyout(snd(\mathbf{S}[[S]]p)), s))$
- $\mathbf{S} : \mathit{Command\_sequence} \to \mathit{Openfile} \to (\mathit{Log} \times \mathit{Openfile})$
  $\mathbf{S}[[C\ \mathbf{cr}\ S]]$
  $= \lambda p.let\ (I', p') = \mathbf{C}[[C]]p\ in$
  $((I'cons\ fst(\mathbf{S}[[S]]p')), snd(\mathbf{S}[[S]]p'))$
  $\mathbf{S}[[\mathbf{quit}]] = \lambda p.(''quit''\ cons\ nil, p)$

Interactive File Editor

- The $\mathbf{S}$ function collects the log messages into a list
- $\mathbf{S}[[\mathbf{quit}]]$ builds the very end of this list
- The equation for $\mathbf{S}[[C\ \mathbf{cr}\ S]]$ deserves a bit of study. It says to:
  - Evaluate $\mathbf{C}[[C]]p$ to obtain the next log entry l' plus the updated open file p'
  - Cons l' to the log list and pass p' onto $\mathbf{S}[[S]]$
  - Evaluate $\mathbf{S}[[S]]p'$ to obtain the meaning of the remainder of the program, which is the rest of the log output plus the final version of the updated open file

Interactive File Editor: Valuation Functions

- $\mathbf{C} : \mathit{Command} \rightarrow \mathit{Openfile} \rightarrow (\mathit{String} \times \mathit{Openfile})$

$$\mathbf{C}[[\textbf{newfile}]] = \lambda p.(\text{"}\mathit{newfile}\text{"}, \mathit{newfile})$$

$$\begin{array}{l}\mathbf{C}[[\textbf{moveforward}]] \\ = \lambda p.\mathit{let}\ (k', p') = \mathit{isempty}(p) \rightarrow (\text{"}\mathit{error : file\ is\ empty}\text{"}, p) \\ \quad [\,]\ (\mathit{at\_last\_record}(p) \rightarrow (\text{"}\mathit{error :\ at\ back\ already}\text{"}, p)\ [\,]\ (\text{""}, \mathit{forwards}(p))) \\ \mathit{in}\ (\text{"}\mathit{moveforward}\text{"}\ \ \mathit{concat}\ k', p'))\end{array}$$

$$\begin{array}{l}\mathbf{C}[[\textbf{moveback}]] \\ = \lambda p.\mathit{let}\ (k', p') = \mathit{isempty}(p) \rightarrow (\text{"}\mathit{error : file\ is\ empty}\text{"}, p) \\ \quad [\,]\ (\mathit{at\_first\_record}(p) \rightarrow (\text{"}\mathit{error : at\ front\ already}\text{"}, p))\ [\,]\ (\text{""}, \mathit{backwards}(p)) \\ \mathit{in}\ (\text{"}\mathit{moveback}\text{"}\ \ \mathit{concat}\ k', p')\end{array}$$

$$\mathbf{C}[[\textbf{insert}\ R]] = \lambda p.(\text{"}\mathit{insert}\ R\text{"}, \mathit{insert}(\mathbf{R}[[R]], p))$$

$$\begin{array}{l}\mathbf{C}[[\textbf{delete}]] \\ = \lambda p.\mathit{let}\ (k', p') = \mathit{isempty}(p) \rightarrow (\text{"}\mathit{error : file\ is\ empty}\text{"}, p)\ [\,]\ (\text{""}, \mathit{delete}(p)) \\ \mathit{in}\ (\text{"}\mathit{delete}\text{"}\ \ \mathit{concat}\ k', p')\end{array}$$

Interactive File Editor: Example Program Workout 1

- $\mathbf{C}[[\mathbf{delete}]](newfile)$

$$
\begin{aligned}
&= let\ (k', p') = isempty(newfile) \rightarrow (''error: file\ is\ empty'', newfile) \\
&\qquad [] \ ('''', delete(newfile)) \\
&in\ (''delete''\ \ concat\ k', p')) \\
&= let\ (k', p') = (''error: file\ is\ empty'', newfile) \\
&in\ (''delete''\ \ concat\ k', p') \\
&= (''delete''\ \ concat\ ''error: file\ is\ empty'', newfile) \\
&= (''delete\ error: \ \ file\ is\ empty'', newfile)
\end{aligned}
$$

Interactive File Editor: Example Program Workout 2

Let $[[A]]$ be the name of a nonempty file in the file system s_0

- $\mathbf{P}[[\textbf{edit } A \textbf{ cr moveback cr delete cr quit}]]s_0$
  $= (\text{"}edit\ A\text{"}\ cons\ fst(\mathbf{S}[[\textbf{moveback cr delete cr quit}]]p_0),$
  $update([[A]],\ copyout(snd(\mathbf{S}[[\textbf{moveback cr delete cr quit}]]p_0), s_0))$
  where $p_0 = copyin(access([[A]], s_0))$

  $= (\text{"}edit\ A\text{"}\ cons\ \text{"}moveback\ error:\ at\ front\ already\text{"}$
  $cons\ fst(\mathbf{S}[[\textbf{delete cr quit}]]p_0)),$
  $update([[A]], copyout(snd(\mathbf{S}[[\textbf{delete cr quit}]]p_0), s_0)))$

Interactive File Editor: Example Program Workout 2

Let $[[A]]$ be the name of a nonempty file in the file system s_0

- $\mathbf{P}[[\textbf{edit } A \textbf{ cr moveback cr delete cr quit}]]s_0$

$$= (\text{"}edit\ A\text{"}\ cons\ fst(\mathbf{S}[[\textbf{moveback cr delete cr quit}]]p_0),$$
$$update([[A]],\ copyout(snd(\mathbf{S}[[\textbf{moveback cr delete cr quit}]]p_0), s_0))$$
$$\text{where } p_0 = copyin(access([[A]], s_0))$$

$$= (\text{"}edit\ A\text{"}\ cons\ \text{"}moveback\ error:\ at\ front\ already\text{"}$$
$$cons\ fst(\mathbf{S}[[\textbf{delete cr quit}]]p_0)),$$
$$update([[A]], copyout(snd(\mathbf{S}[[\textbf{delete cr quit}]]p_0))))$$

$\mathbf{S}[[\textbf{delete cr quit}]]p_0$ simplifies to a pair $(\text{"}delete\ quit\text{"}, p_1)$, for $p_1 = delete(p_0)$, and the final result is:
$(\text{"}edit\ A\ moveback\ error:\ at\ front\ already\ delete\ quit,\text{"}\ update([[A]], copyout(p_1), s_0))$

Dynamically Typed Language

Dynamically Typed Language with IO

- An extension of the Language with assignment
- Languages like Python, SNOBOL allow variables to take on values from different data types during the course of evaluation
- This provides flexibility to the user but requires that type checking be performed at run-time
- The semantics of the language gives us insight into the type checking
- We use:
  - $Storable\_value = Tr + Nat$
  - $Store = Id \rightarrow Storable\_value$
- Since storable values are used in arithmetic and logical expressions, type errors are possible, as in an attempt to add a truth value to a number

Dynamically Typed Language with IO: Abstract Syntax

Abstract Syntax:

- Consider the entities as:

$$P \in Program\_session$$
$$C \in Command$$
$$E \in Expression$$
$$N \in Numeral$$
$$I \in Id$$

$$P ::= C.$$
$$C ::= C_1;\ C_2 \mid I := E \mid$$
$$\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \mid \textbf{read } I \mid \textbf{write } E \mid \textbf{diverge}$$
$$E ::= E_1 + E_2 \mid E_1 = E_2 \mid \neg E \mid (E) \mid I \mid N \mid \textbf{true}$$

Dynamically Typed Language with IO: Semantic Algebras

Semantic Algebras:

- *Truth values*
  Domain: $t \in Tr = B$
  Operations:
  $true, false : Tr$
  $not : Tr \to Tr$
- *Natural Numbers*
  Domain: $n \in Nat = \mathcal{N}$
  Operations:
  $zero, one, \ldots : Nat$
  $plus : Nat \times Nat \to Nat$
  $equals : Nat \times Nat \to Tr$
- *Identifiers*
  Domain: $i \in Id = Identifier$

Dynamically Typed Language with IO: Semantic Algebras

Semantic Algebras:

- *Character String*
  Domain: $String$ = the strings formed from the elements of $\mathcal{C}$ (including an $error$ string)
  Operations:
  $A, B, C, ..., Z : String$
  $empty : String$
  $error : String$
  $concat : String \times String \rightarrow String$
  $length : String \rightarrow Nat$
  $substr : String \times Nat \times Nat \rightarrow String$
- *Values that may be stored*
  Domain: $v \in Storable\_value = Tr + Nat$

Dynamically Typed Language with IO: Semantic Algebras

- *Values that expressions may denote*
  Domain: $x \in Expressible\_value = Storable\_value + Errvalue$
  where $Errvalue = Unit$
  Operations:
  $check\_expr : (Store \rightarrow Expressible\_value) \times$
  $(Storable\_value \rightarrow Store \rightarrow Expressible\_value) \rightarrow (Store \rightarrow Expressible\_value)$
  $f_1\ check\_expr\ f_2 = \lambda s.let\ z = (f_1\ s)\ in\ cases\ z\ of$
  $\quad isStorable\_value(v) \rightarrow (f_2\ v\ s)\ []\ isErrvalue() \rightarrow inErrvalue()\ end$
- Note:
  - $check\_expr$ performs error trapping at the expression level when two expressions are used to build a bigger expression with an operator. For example, consider: $E_1 + E_2$
  - f_1 is the valuation function $[[E_1]]$ of E_1 that takes the store s to produce $v \in Storable\_value$ (or an $Errvalue$ in which case we cannot proceed)
  - If v is not a type error, f_2, the valuation function $[[E_2]]$ of E_2, takes the store s to produce $v' \in Storable\_value$ (or an $Errvalue$)
  - Finally, the operator ($plus$) is applied onto v and v' (if there is not type error) to produce the final result

Dynamically Typed Language with IO: Semantic Algebras

- *Input buffer*
  Domain: $i \in Input = Expressible\_value^*$
  Operations:
  $get\_value : Input \rightarrow (Expressible\_value \times Input)$
  $\quad get\_value = \lambda i.null\ i \rightarrow (inErrvalue(), i)\ []\ (hd\ i, tl\ i)$
- *Output buffer*
  Domain: $o \in Output = (Storable\_value + String)^*$
  Operations:
  $empty : Output$
  $\quad empty = nil$
  $put\_value : Storable\_value \times Output \rightarrow Output$
  $\quad put\_value = \lambda(v, o).inStorable\_value(v)\ cons\ o$
  $put\_message : String \times Output \rightarrow Output$
  $\quad put\_message = \lambda(t, o).inString(t)\ cons\ o$

Dynamically Typed Language with IO: Semantic Algebras

- *Store*
  Domain: $s \in Store = Id \rightarrow Storable\_value$
  Operations:
  $newstore : Store$
  $access : Id \rightarrow Store \rightarrow Storable\_value$
  $access = \lambda(i, s).s(i)$
  $update : Id \rightarrow Storable\_value \rightarrow Store \rightarrow Store$
  $update = \lambda(i, v, s).[i \mapsto v]s$
- *Program State*
  Domain: $a \in State = Store \times Input \times Output$

Dynamically Typed Language with IO: Semantic Algebras

- *Post program state*
  Domain: $z \in Post\_state = OK + Err$
  where $OK = State$ and $Err = State$
  Operations:
  $check\_result : (Store \rightarrow Expressible\_value) \times (Storable\_value \rightarrow State \rightarrow Post\_state_\perp)$
  $\quad \rightarrow (State \rightarrow Post\_state_\perp)$
  $f\ \ check\_result\ \ g = \lambda(s, i, o).let\ z = (f\ \ s)\ in\ cases\ z\ of$
  $\quad isStorable\_value(v) \rightarrow (g\ \ v\ \ (s, i, o))$
  $\quad [\,]\ isErrvalue() \rightarrow$
  $\quad inErr(s, i, put\_message(''type\ \ error'', o))\ end$
  $check\_cmd : (State \rightarrow Post\_state_\perp) \times$
  $\quad (State \rightarrow Post\_state_\perp) \rightarrow (State \rightarrow Post\_state_\perp)$
  $\quad h_1\ \ check\_cmd\ \ h_2 = \lambda a.let\ z = (h_1\ \ a)\ in\ cases\ z\ of$
  $\quad isOK(s, i, o) \rightarrow h_2\ (s, i, o)$
  $\quad [\,]\ isErr(s, i, o) \rightarrow z\ end$

Dynamically Typed Language with IO

- Program state (*State*) is a triple of store and input and output buffers
- The *Post_state* domain is used to signal when an evaluation is completed successfully and when a type error occurs
- The tag attached to the state is utilized by the *check_cmd* operation

The expression ($\mathbf{C}[[C_1]]$ *check_cmd* $\mathbf{C}[[C_2]]$) denotes the sequencing operation for the language and does the following:

[1] It gives the current state a to $\mathbf{C}[[C_1]]$, producing a *Post_state* $z = \mathbf{C}[[C_1]]a$

[2] If z is a proper state a', and then, if the state component is OK, it produces $\mathbf{C}[[C_2]]a'$

[3] If z is erroneous, $\mathbf{C}[[C_2]]$ is ignored (it is *branched over*), and z is the result

check_result, sequences an expression with a command:

[1] For example, for an assignment $[[I := E]]$, $[[E]]$'s value must be determined before a store update can occur

[2] If Since $[[E]]$'s evaluation may cause a type error, the error must be detected before the update is attempted

[3] Operation *check_result* performs this action

check_expr performs error trapping at the expression level

Dynamically Typed Language with IO: Valuation Functions

Valuation Functions:

- $\mathbf{P} : \mathit{Program} \rightarrow \mathit{Store} \rightarrow \mathit{Input} \rightarrow \mathit{Post\_state}_\perp$
  $\mathbf{P}[[C.]] = \lambda s.\lambda i.\mathbf{C}[[C]]\ (s, i, \mathit{empty})$
- $\mathbf{C} : \mathit{Command} \rightarrow \mathit{State} \rightarrow \mathit{Post\_state}_\perp$
  $\mathbf{C}[[C_1;\ C_2]] = \mathbf{C}[[C_1]]\ \mathit{check\_cmd}\ \mathbf{C}[[C_2]]$

Recall:

$\mathit{check\_cmd} : (\mathit{State} \rightarrow \mathit{Post\_state}_\perp)\times$
$(\mathit{State} \rightarrow \mathit{Post\_state}_\perp) \rightarrow (\mathit{State} \rightarrow \mathit{Post\_state}_\perp)$
$h_1\ \mathit{check\_cmd}\ h_2 = \lambda a.\mathit{let}\ z = (h_1\ a)\ \mathit{in\ cases}\ z\ \mathit{of}$
$\mathit{isOK}(s, i, o) \rightarrow h_2\ (s, i, o)$
$[]\ \mathit{isErr}(s, i, o) \rightarrow z\ \mathit{end}$

Dynamically Typed Language with IO: Valuation Functions

- $\mathbf{C}[[I := E]] = \mathbf{E}[[E]]\ check\_result\ (\lambda v.\lambda(s,i,o).inOK((update[[I]]\ v\ s),i,o))$
  $\mathbf{C}[[\mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]] = \mathbf{E}[[E]]\ check\_result$
  $(\lambda v.\lambda(s,i,o).cases\ v\ of\ isTr(t) \rightarrow (t \rightarrow \mathbf{C}[[C_1]]\ []\ \mathbf{C}[[C_2]])(s,i,o)$
  $[]\ isNat(n) \rightarrow inErr(s,i,put\_message(''bad\ test'',o))\ end)$
  $\mathbf{C}[[\mathbf{read}\ I]] = \lambda(s,i,o).let\ (x,i') = get\_value(i)\ in$
  $cases\ x\ of\ isStorable\_value(v) \rightarrow inOK((update[[I]]\ v\ s),i',o)$
  $[]\ isErrvalue() \rightarrow inErr(s,i',put\_message(''bad\ input'',o))\ end$
  $\mathbf{C}[[\mathbf{write}\ E]] = \mathbf{E}[[E]]\ check\_result\ (\lambda v.\lambda(s,i,o).inOK(s,i,put\_value(v,o)))$
  $\mathbf{C}[[\mathbf{diverge}]] = \lambda a.\bot$

Recall:

$check\_result : (Store \rightarrow Expressible\_value) \times (Storable\_value \rightarrow State \rightarrow Post\_state_\bot)$
$\rightarrow (State \rightarrow Post\_state_\bot)$

$f\ check\_result\ g = \lambda(s,i,o).let\ z = (f\ s)\ in\ cases\ z\ of$
$isStorable\_value(v) \rightarrow (g\ v\ (s,i,o))\ []\ isErrvalue() \rightarrow$
$inErr(s,i,put\_message(''type\ error'',o))\ end$

Dynamically Typed Language with IO: Valuation Functions

- $\mathbf{E} : \mathit{Expression} \rightarrow \mathit{Store} \rightarrow \mathit{Expressible\_value}$

$$
\begin{array}{l}
\mathbf{E}[[E_1 + E_2]] = \mathbf{E}[[E_1]]\ \mathit{check\_expr} \\
\quad (\lambda v.\ \mathit{cases}\ v\ \mathit{of} \\
\quad\quad \mathit{isTr}(t) \rightarrow \lambda s.\mathit{inErrvalue}() \\
\quad\quad [\,]\ \mathit{isNat}(n) \rightarrow \mathbf{E}[[E_2]]\ \mathit{check\_expr} \\
\quad\quad\quad (\lambda v'.\lambda s.\ \mathit{cases}\ v'\ \mathit{of} \\
\quad\quad\quad\quad \mathit{isTr}(t') \rightarrow \mathit{inErrvalue}() \\
\quad\quad\quad\quad [\,]\ \mathit{isNat}(n') \rightarrow \mathit{inStorable\_value}(\mathit{inNat}(n\ \mathit{plus}\ n'))\ \mathit{end}) \\
\quad\quad \mathit{end})
\end{array}
$$

Recall:

$$
\begin{array}{l}
\mathit{check\_expr} : (\mathit{Store} \rightarrow \mathit{Expressible\_value}) \times \\
(\mathit{Storable\_value} \rightarrow \mathit{Store} \rightarrow \mathit{Expressible\_value}) \rightarrow (\mathit{Store} \rightarrow \mathit{Expressible\_value}) \\
f_1\ \mathit{check\_expr}\ f_2 = \lambda s.\mathit{let}\ z = (f_1\ s)\ \mathit{in}\ \mathit{cases}\ z\ \mathit{of} \\
\quad\quad \mathit{isStorable\_value}(v) \rightarrow (f_2\ v\ s)\ [\,]\ \mathit{isErrvalue}() \rightarrow \mathit{inErrvalue}() \\
\quad\quad \mathit{end}
\end{array}
$$

Dynamically Typed Language with IO: Valuation Functions

- $\mathbf{E}[[E1 = E2]] = \textit{similar to above equation}$
  $\mathbf{E}[[\neg E]] = \mathbf{E}[[E]] check\_expr$
  $(\lambda v.\lambda s.\ cases\ v\ of$
  $isTr(t) \rightarrow inStorable\_value(inTr(not\ t))\ []\ isNat(n) \rightarrow inErrvalue()\ end)$

  $\mathbf{E}[[(E)]] = \mathbf{E}[[E]]$
  $\mathbf{E}[[I]] = \lambda s.inStorable\_value(access\ [[I]]\ s)$
  $\mathbf{E}[[N]] = \lambda s.inStorable\_value(inNat(\mathbf{N}[[N]]))$
  $\mathbf{E}[[true]] = \lambda s.inStorable\_value(inTr(true))$
  $\mathbf{N} : Numeral \rightarrow Nat(omitted)$

  Recall:
  $check\_expr : (Store \rightarrow Expressible\_value) \times$
  $(Storable\_value \rightarrow Store \rightarrow Expressible\_value) \rightarrow (Store \rightarrow Expressible\_value)$
  $f_1\ check\_expr\ f_2 = \lambda s.let\ z = (f_1\ s)\ in\ cases\ z\ of$
  $isStorable\_value(v) \rightarrow (f_2\ v\ s)\ []\ isErrvalue() \rightarrow inErrvalue()$
  end

Dynamically Typed Language with IO: Example Programs

Compute the semantics (post-state) for the following:

$$
\begin{aligned}
&\mathbf{P}[[\mathbf{read}\ X; X := X+1; \mathbf{write}\ X.]](\lambda i.\ zero,\ (3),\ ())\\
&\quad = \mathbf{C}[[\mathbf{read}\ X; X := X+1; \mathbf{write}\ X]]\ (\lambda i.\ zero,\ (3),\ ())\\
&\quad = (\mathbf{C}[[\mathbf{read}\ X]]\ check\_cmd\ \mathbf{C}[[X := X+1; \mathbf{write}\ X]])\ (\lambda i.\ zero,\ (3),\ ())\\
&\quad = (\lambda a. let\ z = (\mathbf{C}[[\mathbf{read}\ X;]]\ a)\ in\ cases\ z\ of\\
&\quad\quad isOK(s,i,o) \rightarrow \mathbf{C}[[X := X+1; \mathbf{write}\ X]]\ (s,i,o)\\
&\quad\quad []\ isErr(s,i,o) \rightarrow z\ end)\ (\lambda i.\ zero,\ (3),\ ())\\
&\quad = \mathbf{C}[[X := X+1; \mathbf{write}\ X]]\ inOK((\lambda i.i\ equals\ X \rightarrow three\ []\ zero, (), ()))\\
&\quad = (\mathbf{C}[[X := X+1]]\ check\_cmd\ \mathbf{C}[[\mathbf{write}\ X]])\ inOK((\lambda i.i\ equals\ X \rightarrow three\ []\ zero, (), ()))\\
&\quad = (\lambda a. let\ z = (\mathbf{C}[[X := X+1]]\ a)\ in\ cases\ z\ of\\
&\quad\quad isOK(s,i,o) \rightarrow \mathbf{C}[[\mathbf{write}\ X]]\ (s,i,o)\\
&\quad\quad []\ isErr(s,i,o) \rightarrow z\ end)\ inOK((\lambda i.i\ equals\ X \rightarrow three\ []\ zero, (), ()))\\
&\quad = \mathbf{C}[[\mathbf{write}\ X]]\ inOK((\lambda i.i\ equals\ X \rightarrow four\ []\ zero, (), ()))\\
&\quad = inOK((\lambda i.i\ equals\ X \rightarrow four\ []\ zero), (), (4))
\end{aligned}
$$

Dynamically Typed Language with IO: Example Programs

$$
\begin{aligned}
&\mathbf{C}[[\mathbf{read}\ X]]\ (\lambda i.\ zero,\ (3),\ ())\\
&\quad = (\lambda(s, i, o).let\ (x, i') = get\_value(i)\ in\\
&\qquad\quad cases\ x\ of\ \ isStorable\_value(v) \to inOK((update[[X]]\ v\ s), i', o)\\
&\qquad\qquad [] \ isErrvalue() \to inErr(s, i', put\_message(''bad\ \ input'', o))\ end)\ (\lambda i.\ zero,\ (3),\ ())\\
&\quad = let\ (x, i') = get\_value((3))\ in\\
&\qquad\quad cases\ x\ of\ \ isStorable\_value(v) \to inOK((update[[X]]\ v\ (\lambda i.\ zero)), i', ())\\
&\qquad\qquad [] \ isErrvalue() \to inErr(s, i', put\_message(''bad\ \ input'', ()))\ end\\
&\quad = inOK((update[[X]]\ inStorable\_value(three)\ (\lambda i.\ zero)), (), ())\\
&\quad = inOK(\lambda i.i\ \ equals\ X \to three\ []\ zero, (), ())
\end{aligned}
$$

Dynamically Typed Language with IO: Example Programs

$\mathbf{C}[[X := X+1]]\ inOK(\lambda i.i\ equals\ X \rightarrow three\ []\ zero, (), ()))$

$= \mathbf{E}[[X+1]]\ check\_result\ (\lambda v.inOK((update[[X]]\ v\ inOK((\lambda i.i\ equals\ X \rightarrow$
$three\ []\ zero)), (), ())))$

$= let\ z = (\mathbf{E}[[X+1]]\ s_1)\ in\ cases\ z\ of\ isStorable\_value(v) \rightarrow (g\ v\ (s_1, (), ()))$
$[]\ isErrvalue() \rightarrow inErr(s_1, (), put\_message(''type\ error'', ()))\ end$
where $s_1 = (\lambda i.i\ equals\ X \rightarrow three\ []\ zero), (), ())$ and
$g = \lambda v.inOK((update[[X]]\ v\ (\lambda i.i\ equals\ X \rightarrow three\ []\ zero)), (), ())$

$= g\ inNat(four)\ (s_1, (), ())$

$= inOK((update[[X]]\ inNat(four)\ (\lambda i.i\ equals\ X \rightarrow three\ []\ zero)), (), ())$

$= inOK(\lambda i.i\ equals\ X \rightarrow four\ []\ zero), (), ())$

Dynamically Typed Language with IO: Example Programs

$$
\begin{aligned}
&\mathbf{E}[[X+1]]\ s_1 = \mathbf{E}[[X+1]]\ inOK((\lambda i.i\ equals\ X \rightarrow three\ []\ zero, (), ()))\\
&\quad = \mathbf{E}[[X]]\ check\_expr\\
&\qquad (\lambda v.\ cases\ v\ of\ isTr(t) \rightarrow \lambda s.inErrvalue()\\
&\qquad\quad []\ isNat(n) \rightarrow \mathbf{E}[[1]]\ check\_expr\\
&\qquad\qquad (\lambda v'.\lambda s.\ cases\ v'\ of\ isTr(t') \rightarrow inErrvalue()\\
&\qquad\qquad\quad []\ isNat(n') \rightarrow inStorable\_value(inNat(n\ plus\ n'))\ end)\\
&\qquad\quad end)\ inOK((\lambda i.i\ equals\ X \rightarrow three\ []\ zero, (), ()))\\
&\quad = inStorable\_value(inNat(three\ plus\ one)) = inStorable\_value(inNat(four))
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{C}[[\mathbf{write}\ X]] = \mathbf{E}[[X]]\ check\_result\ (\lambda v.\lambda(s,i,o).inOK(s,i,put\_value(v,o)))\\
&\quad = inOK((\lambda i.i\ equals\ X \rightarrow four\ []\ zero), (), (4))
\end{aligned}
$$

Dynamically Typed Language with IO: Example Programs

Compute the semantics (post-state) for the following:

- $\mathbf{P}[[\mathbf{read}\ X; X := X + 1; \mathbf{write}\ X.]](\lambda i.\ zero,\ (3),\ ())$
- $\mathbf{P}[[\mathbf{read}\ X; X := X + 1; \mathbf{write}\ X.]](\lambda i.\ zero,\ (),\ ())$
- $\mathbf{P}[[\mathbf{read}\ X; \mathbf{read}\ Y;$
  $\mathbf{if}\ (X = Y)\ \mathbf{then}\ M := \mathbf{true}\ \mathbf{else}\ M := \neg\mathbf{true}; \mathbf{write}\ M.]]\ (\lambda i.\ zero,\ (3\ 3),\ ())$
- $\mathbf{P}[[\mathbf{read}\ X; \mathbf{read}\ Y;$
  $\mathbf{if}\ (X = Y)\ \mathbf{then}\ M := \mathbf{true}\ \mathbf{else}\ M := \neg\mathbf{true}; \mathbf{write}\ M.]]\ (\lambda i.\ zero,\ (3\ 4),\ ())$
- $\mathbf{P}[[\mathbf{read}\ X; \mathbf{read}\ Y;$
  $\mathbf{if}\ (X = Y)\ \mathbf{then}\ M := \mathbf{true}\ \mathbf{else}\ M := \neg\mathbf{true}; \mathbf{write}\ M.]]\ (\lambda i.\ zero,\ (3),\ ())$
- $\mathbf{P}[[\mathbf{read}\ X; \mathbf{read}\ Y;$
  $\mathbf{if}\ (X = Y)\ \mathbf{then}\ M := X = Y\ \mathbf{else}\ M := \neg(X = Y);\ \mathbf{write}\ M.]]\ (\lambda i.\ zero,\ (3\ 4),\ ())$
- $\mathbf{P}[[\mathbf{if}\ (X = Y)\ \mathbf{then}\ M := \mathbf{true}\ \mathbf{else}\ M := \neg\mathbf{true}; \mathbf{write}\ M.]]\ (\lambda i.\ two,\ (),\ ())$
- $\mathbf{P}[[\mathbf{read}\ X; \mathbf{read}\ Y; X := X + Y; \mathbf{write}\ X.]]\ (\lambda i.\ zero,\ (3\ false),\ ())$

Dynamically Typed Language with IO: Example Programs with denotations

Denotations (post-state) for the programs:

- $\mathbf{P}[[\mathbf{read}\ X; X := X + 1; \mathbf{write}\ X.]](\lambda i.\ zero,\ (3),\ ()) =$
  $inOK((\lambda i.i\ equals\ X \rightarrow four\ []\ zero,\ (),\ (4)))$
- $\mathbf{P}[[\mathbf{read}\ X; X := X + 1; \mathbf{write}\ X.]](\lambda i.\ zero,\ (),\ ()) =$
  $inErr((\lambda i.\ zero,\ (),\ (''bad\ input'')))$
- $\mathbf{P}[[\mathbf{read}\ X; \mathbf{read}\ Y;\ \mathbf{if}\ (X = Y)\ \mathbf{then}\ M := \mathbf{true}\ \mathbf{else}\ M := \neg\mathbf{true}; \mathbf{write}\ M.]]\ (\lambda i.\ zero,\ (3\ 3),\ ()) =$
  $inOK((\lambda i.i\ equals\ X \rightarrow three\ []\ i\ equals\ Y \rightarrow three\ []\ i\ equals\ M \rightarrow true\ []\ zero,\ (),\ (true)))$
- $\mathbf{P}[[\mathbf{read}\ X; \mathbf{read}\ Y;\ \mathbf{if}\ (X = Y)\ \mathbf{then}\ M := \mathbf{true}\ \mathbf{else}\ M := \neg\mathbf{true}; \mathbf{write}\ M.]]\ (\lambda i.\ zero,\ (3\ 4),\ ()) =$
  $inOK((\lambda i.i\ equals\ X \rightarrow three\ []\ i\ equals\ Y \rightarrow four\ []\ i\ equals\ M \rightarrow false\ []\ zero,\ (),\ (false)))$
- $\mathbf{P}[[\mathbf{read}\ X; \mathbf{read}\ Y;\ \mathbf{if}\ (X = Y)\ \mathbf{then}\ M := \mathbf{true}\ \mathbf{else}\ M := \neg\mathbf{true}; \mathbf{write}\ M.]]\ (\lambda i.\ zero,\ (3),\ ()) =$
  $inErr((\lambda i.i\ equals\ X \rightarrow three\ []\ zero,\ (),\ (''bad\ input'')))$
- $\mathbf{P}[[\mathbf{read}\ X; \mathbf{read}\ Y;$
  $\mathbf{if}\ (X = Y)\ \mathbf{then}\ M := X = Y\ \mathbf{else}\ M := \neg(X = Y);\ \mathbf{write}\ M.]]\ (\lambda i.\ zero,\ (3\ 4),\ ()) =$
  $inOK((\lambda i.i\ equals\ X \rightarrow three\ []\ i\ equals\ Y \rightarrow four\ []\ i\ equals\ M \rightarrow true\ []\ zero,\ (),\ (true)))$
- $\mathbf{P}[[\mathbf{if}\ (X = Y)\ \mathbf{then}\ M := \mathbf{true}\ \mathbf{else}\ M := \neg\mathbf{true}; \mathbf{write}\ M.]]\ (\lambda i.\ two,\ (),\ ()) =$
  $inOK((\lambda i.i\ equals\ M \rightarrow true\ []\ two,\ (),\ (true)))$
- $\mathbf{P}[[\mathbf{read}\ X; \mathbf{read}\ Y; X := X + Y; \mathbf{write}\ X.]]\ (\lambda i.\ zero,\ (3\ false),\ ()) =$
  $inErr((\lambda i.i\ equals\ X \rightarrow three\ []\ i\ equals\ Y \rightarrow false\ []\ zero,\ (),\ (''type\ error'')))$

Recursive Definitions

Recursively Defined Functions

- The *copyout* function of File Editor, which concatenates two lists, could not be given. We can specify *copyout* using an iterative or recursive specification, but at this point neither is allowed in the function notation.

$$copyout : Openfile \rightarrow File$$

$$copyout = \lambda p.\ \text{“appends } fst(p) \text{ to } snd(p)\text{”} \quad // \text{ Recursive}$$

$$copyout = \lambda(front, back).null\ front \rightarrow back$$
$$[]\ copyout((tl\ front),\ ((hd\ front)\ cons\ back))$$

Recursively Defined Functions

- A similar situation arises with the semantics of a Pascal-like while-loop:

  B : $Boolean\_expression$
  C : $Command$
  $C ::= \cdots \mid \textbf{while}\ B\ \textbf{do}\ C \mid \cdots$

- Here is a recursive definition of its semantics: for

  $\mathbf{B} : Boolean\_expression \rightarrow Store \rightarrow Tr$, and $\mathbf{C} : Command \rightarrow Store_{\perp} \rightarrow Store_{\perp}$:

$$\mathbf{C}[[\textbf{while}\ B\ \textbf{do}\ C]] = \underline{\lambda} s.\mathbf{B}[[B]]s \rightarrow \mathbf{C}[[\textbf{while}\ B\ \textbf{do}\ C]](\mathbf{C}[[C]]s)\ []\ s$$

Recursively Defined Functions

- Unfortunately, the clause violates the rule that the meaning of a syntax phrase may be defined only in terms of the meanings of its proper sub-parts
- We avoid this problem by stating:

$$\mathbf{C}[[\mathbf{while}\ B\ \mathbf{do}\ C]] = w$$

where $w : Store_{\perp} \rightarrow Store_{\perp}$ is

$$w = \underline{\lambda} s.\mathbf{B}[[B]]s \rightarrow w(\mathbf{C}[[C]]s)\ []\ s$$

- But the recursion remains, for the new version exchanges the recursion in the syntax for recursion in the function notation

Recursive Functions Definitions

A recursive definition may not uniquely define a function. Consider

$$q(x) = x \ equals \ zero \rightarrow one \ [] \ q(x \ plus \ one)$$

which apparently is: $\mathcal{N} \rightarrow \mathcal{N}_\perp$. The following functions all satisfy q's definition in the sense that they have exactly the behavior required by the equation:

- $f_1(x) = one$, if $x = zero$
  $= \perp$, otherwise. OR
  $f_1(x) = \lambda x.(x \ equals \ zero \rightarrow one \ [] \ \perp) = \{(zero, \ one)\}, \ (one, \ \perp), \ (two, \ \perp), \cdots\}$
- $f_2(x) = one$, if $x = zero$
  $= two$, otherwise. OR
  $f_2(x) = \lambda x.(x \ equals \ zero \rightarrow one \ [] \ two) = \{(zero, \ one), \ (one, \ two), \ (two, \ two), \cdots\}$
- $f_3(x) = \lambda x.(one) = \{(zero, \ one), \ (one, \ one), \ (two, \ one), \cdots\}$
- $g_k(x) = \{(zero, \ one), \ (one, \ k), \ (two, \ k), \cdots\}, \ k \in Nat$
- and there are infinitely many others

Recursive Functions Definitions

Given

$$q(x) = x\ equals\ zero \rightarrow one\ []\ q(x\ plus\ one)$$

Prove that $\forall n \in Nat$

[1] $n\ equals\ zero \rightarrow one\ []\ f_1(n\ plus\ one) = f_1(n) = q(n)$
where $f_1(x) = \lambda x.(x\ equals\ zero \rightarrow one\ []\ \bot)$

[2] $n\ equals\ zero \rightarrow one\ []\ f_2(n\ plus\ one) = f_2(n) = q(n)$
where $f_2(x) = \lambda x.(x\ equals\ zero \rightarrow one\ []\ two)$

[3] $n\ equals\ zero \rightarrow one\ []\ f_3(n\ plus\ one) = f_3(n) = q(n)$
where $f_3(x) = \lambda x.(one)$

[4] $n\ equals\ zero \rightarrow one\ []\ g_k(n\ plus\ one) = g_k(n) = q(n)$
where $g_k(x) = \lambda x.(x\ equals\ zero \rightarrow one\ []\ k)$ and $k \in Nat$

Recursive Functions Definitions

[1] $n\ equals\ zero \rightarrow one\ []\ f_1(n\ plus\ one)$

$$= n\ equals\ zero \rightarrow one\ []\ (\lambda x.(x\ equals\ zero \rightarrow one\ []\ \perp))(n\ plus\ one)$$
$$= n\ equals\ zero \rightarrow one\ []\ ((n\ plus\ one)\ equals\ zero \rightarrow one\ []\ \perp)$$
$$= n\ equals\ zero \rightarrow one\ []\ \perp$$
$$= f_1(n) = \lambda x.(x\ equals\ zero \rightarrow one\ []\ \perp)$$

[2] $n\ equals\ zero \rightarrow one\ []\ f_2(n\ plus\ one)$

$$= n\ equals\ zero \rightarrow one\ []\ (\lambda x.(x\ equals\ zero \rightarrow one\ []\ two))(n\ plus\ one)$$
$$= n\ equals\ zero \rightarrow one\ []\ ((n\ plus\ one)\ equals\ zero \rightarrow one\ []\ two)$$
$$= n\ equals\ zero \rightarrow one\ []\ two$$
$$= f_2(n) = \lambda x.(x\ equals\ zero \rightarrow one\ []\ two)$$

[3] $n\ equals\ zero \rightarrow one\ []\ f_3(n\ plus\ one)$

$$= n\ equals\ zero \rightarrow one\ []\ (\lambda x.(one))(n\ plus\ one)$$
$$= n\ equals\ zero \rightarrow one\ []\ one$$
$$= one$$
$$= f_3(n) = \lambda x.(one)$$

Recursive Functions Definitions

- A large number of functions satisfy the recursive definition of q:
$$q(x) = x \ equals \ zero \to one \ [] \ q(x \ plus \ one)$$
- One choice is the function that maps *zero* to *one* and all other arguments to $\perp$. We write this function's graph as $\{(zero, one)\}$ (rather than $\{(zero, one), (one, \perp), (two, \perp), \cdots\}$, treating the $(n, \perp)$ pairs as *ghost members*)
- This choice is a natural one for programming, for it corresponds to what happens when the definition is run as a routine on a machine.

```
// Natural Operational Semantics in C
// q(n) = n equals zero -> one [] q(n plus one)

unsigned int q(unsigned int n) {
    if (n == 0) return 1;
    else return q(n + 1); // Never terminates -- bottom
}
```

Recursive Functions Definitions

- But the graph $\{(zero, one), (one, four), (two, four), (three, four), \cdots\}$ also denotes a function that also has the behavior specified by q: *zero* maps to *one* and all other arguments map to the same answer as their successors

```
// Unnatural Operational Semantics in C
// q(n) = n equals zero -> one [] four

unsigned int q(unsigned int x) {
    if (x == 0) return 1;
    else return 4;
}
```

So we have multiple choices. Not all of them may be operationally acceptable.

- In general, any graph $\{(zero, one), (one, k), (two, k), \cdots\}$, for some $k \in Nat_{\perp}$, represents a function that satisfies the specification.
- For a programmer, the last graph is an unnatural choice for the meaning of q, but a mathematician might like a function with the largest possible graph instead, the claim being that a *fuller* function gives more insight

Recursive Functions Definitions

- As a recursive specification may not define a unique function, so which one should be selected as the meaning of the specification?
- Choose the one that suits operational intuitions
- Theory of *least fixed point semantics* establishes the meaning of recursive specifications. The theory:
  [1] Guarantees that the specification has *at least one function* satisfying it.
  [2] Provides a means for choosing a *best* function out of the set of all functions satisfying the specification.
  [3] Ensures that the function selected has a graph that *corresponds to the operational treatment* of recursion:

  > The function maps an argument *a* to a defined answer *b iff* the operational evaluation of the specification with the representation of argument *a* produces the representation of *b* in a **finite number of recursive invocations**

Recursive Functions Definitions

Consider: $fac : Nat \rightarrow Nat_{\perp}$

$$fac(n) = n \ equals \ zero \rightarrow one \ [] \ n \ times \ (fac(n \ minus \ one))$$
$$= \{(zero, \ one), \ (one, \ one), \ (two, \ two), \ (three, \ six), \cdots, \ (i, \ i!), \cdots\}$$

Recursive Functions Definitions

Finite Unfolding:

[1] *Zero unfolding* (fac_0): no argument $n \in Nat$ can produce an answer, for no form $fac(n)$ can simplify to an answer without the initial unfolding. Hence $graph(fac_0) = \{\ \}$

[2] *One unfolding* (fac_1): This allows fac to be replaced by its body only once. Thus, $fac(zero) \Rightarrow zero\ equals\ zero \rightarrow one\ []\ \cdots = one$, but all other nonzero arguments require further unfoldings to simplify to answers. Hence $graph(fac_1) = \{(zero,\ one)\}$

[3] *Two unfolding's* (fac_2): Since only one unfolding is needed for mapping argument $zero$ to one, $(zero, one)$ appears in the graph. The extra unfolding allows argument one to evaluate to one, for
$fac(one) \Rightarrow one\ equals\ zero \rightarrow one\ []\ one\ times\ (fac(one\ minus\ one)) =$
$one\ times\ fac(zero) \Rightarrow one\ times\ (zero\ equals\ zero \rightarrow one\ []\ \cdots) = one\ times\ one =$
one. All other arguments require further unfoldings and do not produce answers at this stage. Hence $graph(fac_2) = \{(zero,\ one),\ (one,\ one)\}$

[4] $(i+1)$ *unfolding's* (fac_{i+1}), for $i \geq 0$: All arguments with values of i or less will simplify to answers $i!$, giving
$graph(fac_{i+1}) = \{(zero,\ one),\ (one,\ one),\ (two,\ two),\ (three,\ six), \cdots,\ (i,\ i!)\}$.

Unfolding of fac by Simulation

// Project: PoPL 2019. fac_unfold.cpp, $\perp$ shown as -1

$fac = \lambda n.n\ equals\ zero \rightarrow one\ []\ n\ times\ (fac(n\ minus\ one))$

```
COMPUTING LFP of fac(n) =
n equals zero -> one[] n times (fac(n minus one))

fac(0) = 1 in 1 unfolds
fac(1) = 1 in 2 unfolds
fac(2) = 2 in 3 unfolds
fac(3) = 6 in 4 unfolds
fac(4) = 24 in 5 unfolds
fac(5) = 120 in 6 unfolds
fac(6) = 720 in 7 unfolds
fac(7) = 5040 in 8 unfolds
fac(8) = 40320 in 9 unfolds
fac(9) = 362880 in 10 unfolds
```

Unfolding of fac by Simulation

// Project: PoPL 2019. fac_unfold.cpp, $\perp$ shown as -1

```
// fac(n) = n equals zero -> one[] n times (fac(n minus one))
#include <iostream>
using namespace std;
static unsigned int facCount = 0, maxUnfoldingLevel;
unsigned int fac(unsigned int x) { ++facCount;
    if (x == 0) return 1;
    else if (facCount == maxUnfoldingLevel) throw 1;
         else return x * fac(x - 1);
}
int fac_unfold(unsigned int maxUnfoldingLevel = 100, unsigned int maxParam = 10) {
    bool bottom = false; unsigned int result; ::maxUnfoldingLevel = maxUnfoldingLevel;
    cout<<"COMPUTING LFP of fac(n) = n equals zero -> one [] n times (fac(n minus one))\n";
    for (unsigned int n = 0; n < maxParam; ++n) {
        try { bottom = false; facCount = 0; result = fac(n); }
        catch (int) { bottom = true; }
        cout << "fac(" << n << ") = "
             << (int)((bottom) ? -1 : result) << " in " << facCount << " unfolds" << endl;
    }
    cout << endl << endl;
    return 0;
}
```

Recursive Functions Definitions

Prove: $\cup_{i=0}^{\infty} graph(fac_i) = graph(factorial)$

Recursive Functions Definitions

Prove: $\cup_{i=0}^{\infty}\ graph(fac_i) = graph(factorial)$

Forward direction:

- We get: $\forall i \geq 0, graph(fac_i) \subseteq graph(fac_{i+1})$
- Clearly, $\forall i \geq 0, graph(fac_i) \subseteq graph(factorial)$
- Hence,

$$\cup_{i=0}^{\infty}\ graph(fac_i) \subseteq graph(factorial)$$

Backward direction:

- If some pair (a, b) is in $graph(factorial)$, then there must be some finite $i > 0$ such that (a, b) is in $graph(fac_i)$ also. Thus:

$$graph(factorial) \subseteq \cup_{i=0}^{\infty}\ graph(fac_i)$$

- Hence,

$$\cup_{i=0}^{\infty}\ graph(fac_i) = graph(factorial)$$

Recursive Functions Definitions

The fundamental principle of least fixed point semantics:

- The meaning of any recursively defined function is exactly the union of the meanings of its finite sub-functions
- It is easy to produce a non-recursive representation of each sub-function

For example: Define each $fac_i : Nat \to Nat_\perp$, for $i \geq 0$, as:

- $fac_0 = \lambda n.\perp$
- $fac_{i+1} = \lambda n.n\ equals\ zero \to one\ []$
  $n\ times\ fac_i(n\ minus\ one)$, for all $i \geq 0$

The graph of each fac_i is the one produced at stage i of the fac unfolding

Recursive Functions Definitions

This has two advantages:

[1] Each fac_i is a non-recursive definition, which suggests that a recursive specification can be understood in terms of a family of non-recursive ones; and

[2] A format common to all the fac_i's can be extracted. Let:

$$
\begin{aligned}
F &= \lambda f.\lambda n.n \; equals \; zero \rightarrow one \; [] \\
&\qquad\quad n \; times \; f(n \; minus \; one) \\
&= \lambda f.\lambda n.n \; equals \; zero \rightarrow one \; [] \\
&\qquad\quad let \; n' \; = \; f(n \; minus \; one) \; in \; n \; times \; n'
\end{aligned}
$$

Each $fac_{i+1} = F(fac_i)$, $\forall i \geq 0$

Recursive Functions Definitions: factorial

- The non-recursive $F : (Nat \to Nat_\perp) \to (Nat \to Nat_\perp)$ is called a *functional*, because it takes a function as an argument and produces one as a result. Thus:

$$graph(factorial) = \cup_{i=0}^{\infty} graph(F^i(\Phi))$$

  where $F^i = F \circ F \circ \cdots \circ F$, i times, and $\Phi = (\lambda n.\perp)$
- Also, $graph(F(factorial)) = graph(factorial)$, which implies $F(factorial) = factorial$, by the *extensionality principle*
- The *factorial* function is a fixed point of F, as the answer F produces from argument factorial is exactly factorial again

$$factorial = fix\ F$$

Recursive Functions Definitions

Consider:

$q : Nat \rightarrow Nat_{\perp},\ q(x) = x\ equals\ zero \rightarrow one\ []\ q(x\ plus\ one)$

Then, $Q : (Nat \rightarrow Nat_{\perp}) \rightarrow (Nat \rightarrow Nat_{\perp})$

$Q = \lambda g.\lambda n.n\ equals\ zero \rightarrow one\ []\ g(n\ plus\ one)$

Compute the fixed point of Q

Recursive Functions Definitions: q Function

Consider:
$q : \mathcal{N} \to \mathcal{N}_\perp$, $q(x) = x \textit{ equals zero} \to \textit{one} \; [] \; q(x \textit{ plus one})$

Then, $Q : (Nat \to Nat_\perp) \to (Nat \to Nat_\perp)$
$Q = \lambda g.\lambda n.n \textit{ equals zero} \to \textit{one} \; [] \; g(n \textit{ plus one})$

We get:
$Q^0(\Phi) = (\lambda n.\perp)$, where $\Phi = (\lambda n.\perp)$
$graph(Q^0(\Phi)) = \{ \}$

$Q^1(\Phi) = Q(Q^0(\Phi)) = \lambda n.n \textit{ equals zero} \to \textit{one} \; [] \; (\lambda n.\perp)(n \textit{ plus one})$
$\qquad = \lambda n.n \textit{ equals zero} \to \textit{one} \; [] \perp$
$graph(Q^1(\Phi)) = \{(zero, \; one)\}$

$Q^2(\Phi) = Q(Q^1(\Phi)) = \lambda n.n \textit{ equals zero} \to \textit{one} \; [] \; ((n \textit{ plus one}) \textit{ equals zero} \to \textit{one} \; [] \perp)$
$\qquad = \lambda n.n \textit{ equals zero} \to \textit{one} \; [] \perp \; = \; Q^1(\Phi)$
$graph(Q^2(\Phi)) = \{(zero, \; one)\}$

Hence, $\forall i \geq 1, graph(Q^i(\Phi)) = \{(zero, one)\}$. It follows that:

$$\cup_{i=0}^{\infty} graph(Q^i(\Phi)) = \{(zero, \; one)\}$$

Recursive Functions Definitions: q Function

Let $qlimit$ denote the function that has this graph. It is easy to show that $Q(qlimit) = qlimit$, that is, $qlimit$ is a fixed point of Q. Or, $qlimit = fix\ Q$. Unlike the specification fac, q has many possible solutions. Recall that each one must have a graph of the form $\{(zero, one), (one, k), \cdots, (i, k), \cdots\}$ for some $k \in Nat_{\perp}$.

Let qk be one of these solutions. We can show that:

[1] qk is a fixed point of Q, that is, $Q(qk) = qk$

[2] $graph(qlimit) \subseteq graph(qk)$

- Fact 1 says that the act of satisfying a specification is formalized by the fixed point property – only fixed points of the associated functional are possible meanings of the specification
- Fact 2 states that the solution obtained using the stages of unfolding method is the smallest of all the possible solutions

So, we call it the *least fixed point* of the *functional*

Try to prove Fact 1 and Fact 2 above

Unfolding of q by Simulation

// Project: PoPL 2019. q_unfold.cpp, $\perp$ shown as -1

$q = \lambda x.x \ equals \ zero \rightarrow one \ [] \ q(x \ plus \ one)$

```
COMPUTING LFP of q(n) =
n equals zero -> one [] q(n plus one)

q(0) = 1 in 1 unfolds
q(1) = -1 in 100 unfolds
q(2) = -1 in 100 unfolds
q(3) = -1 in 100 unfolds
q(4) = -1 in 100 unfolds
q(5) = -1 in 100 unfolds
q(6) = -1 in 100 unfolds
q(7) = -1 in 100 unfolds
q(8) = -1 in 100 unfolds
q(9) = -1 in 100 unfolds
```

Unfolding of q by Simulation

// Project: PoPL 2019. q_unfold.cpp, $\perp$ shown as -1

```
// q(n) = n equals zero -> one [] q(n plus one)
#include <iostream>
using namespace std;
static unsigned int qCount = 0, maxUnfoldingLevel;
unsigned int q(unsigned int x) { ++qCount;
    if (x == 0) return 1;
    else if (qCount == maxUnfoldingLevel) throw 1; else return q(x + 1);
}
int q_unfold(unsigned int maxUnfoldingLevel = 100, unsigned int maxParam = 10) {
    bool bottom = false; unsigned int result; ::maxUnfoldingLevel = maxUnfoldingLevel;
    cout << "COMPUTING LFP of q(n) = n equals zero -> one [] q(n plus one)" << endl;
    for (unsigned int n = 0; n < maxParam; ++n) {
        try { bottom = false; qCount = 0; result = q(n); }
        catch (int) { bottom = true; }
        cout << "q(" << n << ") = " << (int)((bottom) ? -1 : result)
             << " in " << qCount << " unfolds" << endl;
    }
    cout << endl << endl;
    return 0;
}
```

Recursive Functions Definitions

Let a recursive specification $f = F(f)$ denote the least fixed point of functional F, that is, the function associated with $\cup_{i=0}^{\infty} graph(F^i(\Phi))$, as obtained by the stages of unfolding.

The three desired properties follow:

- A solution to the specification exists;
- The criterion of least-ness is used to select from the possible solutions; and,
- Since the method for constructing the function exactly follows the usual operational treatment of recursive definitions, the solution corresponds to the one determined computationally

Recursive Functions Definitions: *copyout*

copyout function:

- The domains:
  $File = Record^*$ and $Openfile = Record^* \times Record^*$
- Function *copyout* converts an open file into a file by appending the two record lists
- A specification of $copyout$: $Openfile \rightarrow File_\perp$ is:
  $$copyout = \lambda(front, back).null\ front \rightarrow back\ []\ copyout((tl\ front), ((hd\ front)\ cons\ back))$$
- Construct functional F such that $copyout = (fix\ F)$
- Prove that the function $F^i(\perp)$ is capable of appending list pairs whose first component has length $i-1$ or less
- This implies that the *lub* of the $F^i(\perp)$ functions, $(fix\ F)$, is capable of concatenating all pairs of lists whose first component has finite length

Recursive Functions Definitions: *copyout*

copyout function:

- $copyout^0 = \lambda(front, back).\perp$
- $copyout^1 = \lambda(front, back).null\ front \rightarrow back\ []\ copyout^0((tl\ front), ((hd\ front)\ cons\ back))$
  $= \lambda(front, back).null\ front \rightarrow back\ []\ \perp$
- $copyout^2 = \lambda(front, back).null\ front \rightarrow back\ []\ copyout^1((tl\ front), ((hd\ front)\ cons\ back))$
  $= \lambda(front, back).null\ front \rightarrow back\ []$
  $(\lambda(front, back).null\ front \rightarrow back\ []\ \perp)\ ((tl\ front), ((hd\ front)\ cons\ back))$
  $= \lambda(front, back).null\ front \rightarrow back\ []\ (null\ (tl\ front) \rightarrow ((hd\ front)\ cons\ back)\ []\ \perp)$
- $copyout^{i+1} = \lambda(front, back).null\ front \rightarrow back\ []\ copyout^i((tl\ front), ((hd\ front)\ cons\ back))$
- Functional $F : (Openfile \rightarrow File_\perp) \rightarrow (Openfile \rightarrow File_\perp)$:
  $F = \lambda f.\lambda(front, back).null\ front \rightarrow back\ []\ f((tl\ front), ((hd\ front)\ cons\ back))$
- $copyout = fix\ F = F(copyout)$

Recursive Functions Definitions: Double Recursion

Consider the specification for $g : Nat \to Nat_{\perp}$:

$g = \lambda n.n\ equals\ zero \to one\ []$
$(g(n\ minus\ one)\ plus\ g(n\ minus\ one))\ minus\ one$

What is g?

Hint: Construct F and compute the graphs of $F^i(\perp)$

Recursive Functions Definitions: Double Recursion

Consider the specification for $g : Nat \to Nat_{\perp}$:

$$g = \lambda n.n \; equals \; zero \to one \; [] \; (g(n \; minus \; one) \; plus \; g(n \; minus \; one)) \; minus \; one$$

$$F = \lambda f.\lambda n.n \; equals \; zero \to one \; [] \; (f(n \; minus \; one) \; plus \; f(n \; minus \; one)) \; minus \; one$$

Using $\perp$ for $(\lambda n.\perp)$

$$graph(F^0(\perp)) = \{\}$$
$$graph(F^1(\perp)) = \{(zero, one)\}$$
$$graph(F^2(\perp)) = \{(zero, one)\}$$
$$graph(F^3(\perp)) = \{(zero, one), (one, one)\}$$
$$graph(F^4(\perp)) = \{(zero, one), (one, one)\}$$
$$graph(F^5(\perp)) = \{(zero, one), (one, one)\}$$
$$graph(F^6(\perp)) = \{(zero, one), (one, one)\}$$
$$graph(F^7(\perp)) = \{(zero, one), (one, one), (two, one)\}$$

Recursive Functions Definitions: Double Recursion

$graph(F^0(\bot)) = \{\}$
$graph(F^1(\bot)) = \{(zero, one)\}$
$graph(F^2(\bot)) = \{(zero, one)\}$
$graph(F^3(\bot)) = \{(zero, one), (one, one)\}$
$graph(F^4(\bot)) = \{(zero, one), (one, one)\}$
$graph(F^5(\bot)) = \{(zero, one), (one, one)\}$
$graph(F^6(\bot)) = \{(zero, one), (one, one)\}$
$graph(F^7(\bot)) = \{(zero, one), (one, one), (two, one)\}$
$graph(F^8(\bot)) = \{(zero, one), (one, one), (two, one)\}$
$graph(F^9(\bot)) = \{(zero, one), (one, one), (two, one)\}$
$graph(F^{10}(\bot)) = \{(zero, one), (one, one), (two, one)\}$
$graph(F^{11}(\bot)) = \{(zero, one), (one, one), (two, one)\}$
$graph(F^{12}(\bot)) = \{(zero, one), (one, one), (two, one)\}$
$graph(F^{13}(\bot)) = \{(zero, one), (one, one), (two, one)\}$
$graph(F^{14}(\bot)) = \{(zero, one), (one, one), (two, one)\}$
$graph(F^{15}(\bot)) = \{(zero, one), (one, one), (two, one), (three, one)\}$

$\forall i, i \geq 0, graph(F^i(\bot)) = \{(zero, one), (one, one), (two, one), \cdots, (i, one)\}$

Hence: $(fix\ F) = \lambda n.one$

Prove: $graph(F^i(\bot)) = \cup_{k=0}^{j}\{(k, one)\}$, $2^{j+1} - 1 \leq i \leq 2^{j+2} - 2$, $j \geq 0$

Recursive Functions Definitions: Double Recursion

$F = \lambda f.\lambda n.n \ equals \ zero \rightarrow one \ []$
$(f(n \ minus \ one) \ plus \ f(n \ minus \ one)) \ minus \ one$

Prove:

- Exponential number of unfoldings are required for this graph:

$$\begin{aligned} graph(F^0(\perp)) &= \{\ \} \\ graph(F^i(\perp)) &= \cup_{k=0}^{j}\{(k, one)\},\ 2^{j+1} - 1 \leq i \leq 2^{j+2} - 2,\ j \geq 0 \end{aligned}$$

- $(fix \ F) = \lambda n.one$

Unfolding of g (Double) by Simulation

// Project: PoPL 2019. g_double_unfold.cpp, $\perp$ shown as -1

$g = \lambda n.n \; equals \; zero \rightarrow one \; [] \; (g(n \; minus \; one) \; plus \; g(n \; minus \; one)) \; minus \; one$

```
COMPUTING LFP of g(n) = n equals zero -> one []
          (g(n minus one) plus g(n minus one)) minus one

g_double(0) = 1 in 1 unfolds
g_double(1) = 1 in 3 unfolds
g_double(2) = 1 in 7 unfolds
g_double(3) = 1 in 15 unfolds
g_double(4) = 1 in 31 unfolds
g_double(5) = 1 in 63 unfolds
g_double(6) = 1 in 127 unfolds
g_double(7) = 1 in 255 unfolds
g_double(8) = 1 in 511 unfolds
g_double(9) = -1 in 1000 unfolds
```

This is till 1000 unfolds. g_double(9) gets 1 in 1023 unfolds

Unfolding of g (Double) by Simulation

// Project: PoPL 2019. g_double_unfold.cpp, $\perp$ shown as -1

```
// g(n) = n equals zero -> one [] (g(n minus one) plus g(n minus one)) minus one
#include <iostream>
using namespace std;
static unsigned int g_doubleCount = 0, maxUnfoldingLevel;
unsigned int g_double(unsigned int x) { ++g_doubleCount;
    if (g_doubleCount == maxUnfoldingLevel) throw 1;
    if (x == 0) return 1;
    else try { return g_double(x - 1) + g_double(x - 1) - 1;} catch (int) { throw; }
}
int g_double_unfold(unsigned int maxUnfoldingLevel = 100, unsigned int maxParam = 10) {
    bool bottom = false; unsigned int result; ::maxUnfoldingLevel = maxUnfoldingLevel;
    cout << "COMPUTING LFP of g(n)";
    cout << " = n equals zero -> one [] (g(n minus one) plus g(n minus one)) minus one\n";
    for (unsigned int n = 0; n < maxParam; ++n) {
        try { bottom = false; g_doubleCount = 0; result = g_double(n); }
        catch (int) { bottom = true; }
        cout << "g_double(" << n << ") = " << (int)((bottom) ? -1 : result)
             << " in " << g_doubleCount << " unfolds" << endl;
    }
    cout << endl << endl;
    return 0;
}
```

Recursive Functions Definitions: Simultaneous Definition

Consider the specifications $f : Nat \rightarrow Nat_\perp$ and $g : Nat \rightarrow Nat_\perp$:

$$f = \lambda x.x \ equals \ zero \rightarrow g(zero) \ [] \ f(g(x \ minus \ one)) \ plus \ two$$

$$g = \lambda y.y \ equals \ zero \rightarrow zero \ [] \ y \ times \ f(y \ minus \ one)$$

Build a functional for function pairs as:
$F : ((Nat \rightarrow Nat_\perp) \times (Nat \rightarrow Nat_\perp)) \rightarrow ((Nat \rightarrow Nat_\perp) \times (Nat \rightarrow Nat_\perp))$

$$F = \lambda(f, g).(\lambda x.x \ equals \ zero \rightarrow g(zero) \ [] \ f(g(x \ minus \ one)) \ plus \ two, \\ \lambda y.y \ equals \ zero \rightarrow zero \ [] \ y \ times \ f(y \ minus \ one))$$

Find a pair of functions (α, β) such that $F(\alpha, \beta) = (\alpha, \beta)$

Recursive Functions Definitions: Simultaneous Definition

Consider the specifications $f : Nat \rightarrow Nat_{\perp}$ and $g : Nat \rightarrow Nat_{\perp}$:

$f = \lambda x.x\ equals\ zero \rightarrow g(zero)\ []\ f(g(x\ minus\ one))\ plus\ two$

$g = \lambda y.y\ equals\ zero \rightarrow zero\ []\ y\ times\ f(y\ minus\ one)$

$$F = \lambda(f, g).(\lambda x.x\ equals\ zero \rightarrow g(zero)\ []\ f(g(x\ minus\ one))\ plus\ two, \\ \lambda y.y\ equals\ zero \rightarrow zero\ []\ y\ times\ f(y\ minus\ one))$$

Using $\perp$ for $((\lambda n.\perp), (\lambda n.\perp))$

$F^0(\perp) = (\{\ \}, \{\ \})$
$F^1(\perp) = (\{\ \}, \{(zero, zero)\})$
$F^2(\perp) = (\{(zero, zero)\}, \{(zero, zero)\})$
$F^3(\perp) = (\{(zero, zero)\}, \{(zero, zero), (one, zero)\})$
$F^4(\perp) = (\{(zero, zero), (one, two)\},\ \{(zero, zero), (one, zero)\})$
$F^5(\perp) = (\{(zero, zero), (one, two)\},\ \{(zero, zero), (one, zero), (two, four)\})$
$F^6(\perp) = (\{(zero, zero), (one, two), (two, two)\},\ \{(zero, zero), (one, zero), (two, four)\})$
$F^7(\perp) = (\{(zero, zero), (one, two), (two, two)\},\ \{(zero, zero), (one, zero), (two, four), (three, six)\})$

$\forall i, i > 7, F^i(\perp) = F^7(\perp)$

$f = fst(fix\ F),\ g = snd(fix\ F)$

Unfolding of f & g (Simultaneous) by Simulation

// Project: PoPL 2019. f_g_unfold.cpp, $\perp$ shown as -1

$f = \lambda x.x \textit{ equals zero} \rightarrow g(\textit{zero})\ [\!]\ f(g(x \textit{ minus one})) \textit{ plus two}$
$g = \lambda y.y \textit{ equals zero} \rightarrow \textit{zero}\ [\!]\ y \textit{ times } f(y \textit{ minus one})$

```
COMPUTING LFP of f_g_simul
f(x) = x equals zero->g(zero) [] f(g(x minus one)) plus two
g(y) = y equals zero -> zero[] y times f(y minus one)

g(0) = 0 in 1 unfolds
g(1) = 0 in 3 unfolds
g(2) = 4 in 5 unfolds
g(3) = 6 in 7 unfolds
g(4) = -1 in 1001 unfolds
g(5) = -1 in 1001 unfolds
g(6) = -1 in 1001 unfolds
g(7) = -1 in 1001 unfolds
g(8) = -1 in 1001 unfolds
g(9) = -1 in 1001 unfolds

f(0) = 0 in 2 unfolds
f(1) = 2 in 4 unfolds
f(2) = 2 in 6 unfolds
f(3) = -1 in 1001 unfolds
f(4) = -1 in 1001 unfolds
f(5) = -1 in 1001 unfolds
f(6) = -1 in 1001 unfolds
f(7) = -1 in 1001 unfolds
f(8) = -1 in 1001 unfolds
f(9) = -1 in 1001 unfolds
```

Unfolding of f & g (Simultaneous) by Simulation

// Project: PoPL 2019. f_g_unfold.cpp, $\perp$ shown as -1

```
//f(x) = x equals zero -> g(zero)[] f(g(x minus one)) plus two
//g(y) = y equals zero -> zero[] y times f(y minus one)

#include <iostream>
using namespace std;
static unsigned int fCount = 0, gCount = 0, maxUnfoldingLevel;
unsigned int g(unsigned int x);
//f(x) = x equals zero -> g(zero)[] f(g(x minus one)) plus two
unsigned int f(unsigned int x) {
    ++fCount;
    if (fCount + gCount > maxUnfoldingLevel) throw 1;
    if (x == 0) { // return g(0);
        try { int t = g(0); return t; } catch (int) { throw; }
    }
    else try { int t = g(x - 1); t = f(t); return t + 2; } catch (int) { throw; }
}
//g(y) = y equals zero -> zero[] y times f(y minus one)
unsigned int g(unsigned int x) { ++gCount;
    if (fCount + gCount > maxUnfoldingLevel) throw 2;
    if (x == 0) return 0;
    else try { int t = f(x - 1); return x * t; } catch (int) { throw; }
}
```

Unfolding of f & g (Simultaneous) by Simulation

Module M09

Partha Pratim Das

Imperative Languages

Language + Assignment

Programs Are Functions

Interactive File Editor

Dynamically Typed Language

Recursive Definitions

copyout

Unfolding Examples

While Loop

Languages with Context

Block Structured

Applicative

Summary

// Project: PoPL 2019. f_g_unfold.cpp, $\perp$ shown as -1

```
//f(x) = x equals zero -> g(zero)[] f(g(x minus one)) plus two
//g(y) = y equals zero -> zero[] y times f(y minus one)

int f_g_unfold(unsigned int maxUnfoldingLevel = 100, unsigned int maxParam = 10) {
    bool bottom = false; unsigned int gResult, fResult; ::maxUnfoldingLevel = maxUnfoldingLevel;
    cout << "COMPUTING LFP of f_g_simul" << endl;
    cout << "f(x) = x equals zero->g(zero) [] f(g(x minus one)) plus two" << endl;
    cout << "g(y) = y equals zero -> zero[] y times f(y minus one)" << endl;
    for (unsigned int n = 0; n < maxParam; ++n) {
        try { bottom = false; fCount = gCount = 0; gResult = g(n); }
        catch (int) { bottom = true; }
        cout << "g(" << n << ") = " << (int)((bottom) ? -1 : gResult) << " in "
             << fCount + gCount << " unfolds" << endl;
    }
    cout << endl;
    for (unsigned int n = 0; n < maxParam; ++n) {
        try { bottom = false; fCount = gCount = 0; fResult = f(n); }
        catch (int) { bottom = true; }
        cout << "f(" << n << ") = " << (int)((bottom) ? -1 : fResult) << " in "
             << fCount + gCount << " unfolds" << endl;
    }
    cout << endl << endl;
    return 0;
}
```

Recursive Functions Definitions: Simultaneous Definition

This is the solution given in the book. This is wrong.

Consider the specifications $f : Nat \to Nat_\perp$ and $g : Nat \to Nat_\perp$:
$f = \lambda x.x\ equals\ zero \to g(zero)\ []\ f(g(x\ minus\ one))\ plus\ two$
$g = \lambda y.y\ equals\ zero \to zero\ []\ y\ times\ f(y\ minus\ one)$

$$
\begin{aligned}
F = \lambda(f, g).(&\lambda x.x\ equals\ zero \to g(zero)\ []\ f(g(x\ minus\ one))\ plus\ two,\\
&\lambda y.y\ equals\ zero \to zero\ []\ y\ times\ f(y\ minus\ one))
\end{aligned}
$$

Using $\perp$ for $((\lambda n.\perp), (\lambda n.\perp))$

$$
\begin{aligned}
F^0(\perp) &= (\{\ \}, \{\ \})\\
F^1(\perp) &= (\{\ \}, \{(zero, zero)\})\\
F^2(\perp) &= (\{(zero, zero)\}, \{(zero, zero)\})\\
F^3(\perp) &= (\{(zero, zero), (one, two)\}, \{(zero, zero), (one, zero)\})\\
F^4(\perp) &= (\{(zero, zero), (one, two), (two, two)\},\\
&\quad \{(zero, zero), (one, zero), (two, four)\})\\
F^5(\perp) &= (\{(zero, zero), (one, two), (two, two)\},\\
&\quad \{(zero, zero), (one, zero), (two, four), (three, six)\})
\end{aligned}
$$

$\forall i, i > 5, F^i(\perp) = F^5(\perp)$

$f = fst(fix\ F),\ g = snd(fix\ F)$

Recursive Functions Definitions: Simultaneous Definition

- Any finite set of mutually recursive function definitions can be handled in this manner
- Thus, the least fixed point method is powerful enough to model the most general forms of computation, such as general recursive equation sets and flowcharts

Unfolding of odd-even by Simulation

// Project: PoPL 2019. odd_even_unfold.cpp, $\perp$ shown as -1

$oe = \lambda x.x \; equals \; zero \rightarrow one \; [] \; (x \; equals \; one \rightarrow one \; [] \; (odd \; x \rightarrow oe(x \; plus \; one) \; [] \; oe(x \; div \; two)))$

```
COMPUTING LFP of oe(x) =
x equals zero -> one [] (x equals one -> one [] (odd x -> oe(x plus one) [] oe(x div two)))

odd_even(0) = 1 in 1 unfolds
odd_even(1) = 1 in 1 unfolds
odd_even(2) = 1 in 2 unfolds
odd_even(3) = 1 in 4 unfolds
odd_even(4) = 1 in 3 unfolds
odd_even(5) = 1 in 6 unfolds
odd_even(6) = 1 in 5 unfolds
odd_even(7) = 1 in 5 unfolds
odd_even(8) = 1 in 4 unfolds
odd_even(9) = 1 in 8 unfolds
```

Hence, $oe = \lambda x.one$

Unfolding of odd-even by Simulation

// Project: PoPL 2019. odd_even_unfold.cpp, $\perp$ shown as -1

```
// oe(x) = x equals zero -> one [] (x equals one -> one [] (odd x -> oe(x plus one) [] oe(x div two)))
#include <iostream>
using namespace std;
static unsigned int odd_evenCount = 0,  maxUnfoldingLevel;
unsigned int odd_even(unsigned int x) { ++odd_evenCount;
    if (odd_evenCount == maxUnfoldingLevel) throw 1;
    if (x == 0) return 1;
    else if (x == 1) return 1;
    else if (x % 2) return odd_even(x + 1);
    else return odd_even(x / 2);
}
int odd_even_unfold(unsigned int maxUnfoldingLevel = 100, unsigned int maxParam = 10) {
    bool bottom = false; unsigned int result; ::maxUnfoldingLevel = maxUnfoldingLevel;
    cout << "COMPUTING LFP of oe(x) = x equals zero -> one [] (x equals one -> one"
         << "[] (odd x -> oe(x plus one) [] oe(x div two)))" << endl;
    for (unsigned int n = 0; n < maxParam; ++n) {
        try { bottom = false; odd_evenCount = 0; result = odd_even(n); }
        catch (int) { bottom = true; }
        cout << "odd_even(" << n << ") = " << (int)((bottom) ? -1 : result) << " in "
             << odd_evenCount << " unfolds" << endl;
    }
    cout << endl << endl;
    return 0;
}
```

Module M09

Partha Pratim Das

Unfolding of odd-even (bottom) by Simulation

// Project: PoPL 2019. odd_even_bot_unfold.cpp, $\perp$ shown as -1

$oe = \lambda x.x \; equals \; zero \rightarrow one \; [] \; (odd \; x \rightarrow oe(x \; plus \; one) \; [] \; oe(x \; div \; two))$

```
COMPUTING LFP of oe(x) =
x equals zero -> one [] (odd x -> oe(x plus one) [] oe(x div two))

odd_even_bot(0) = 1 in 1 unfolds
odd_even_bot(1) = -1 in 1000 unfolds
odd_even_bot(2) = -1 in 1000 unfolds
odd_even_bot(3) = -1 in 1000 unfolds
odd_even_bot(4) = -1 in 1000 unfolds
odd_even_bot(5) = -1 in 1000 unfolds
odd_even_bot(6) = -1 in 1000 unfolds
odd_even_bot(7) = -1 in 1000 unfolds
odd_even_bot(8) = -1 in 1000 unfolds
odd_even_bot(9) = -1 in 1000 unfolds
```

Hence, $oe = \lambda x.x \; equals \; zero \rightarrow one \; [] \; \perp$

Unfolding of odd-even (bottom) by Simulation

// Project: PoPL 2019. odd_even_bot_unfold.cpp, $\perp$ shown as -1

```
// oe(x) = x equals zero -> one [] (odd x -> oe(x plus one) [] oe(x div two))
#include <iostream>
using namespace std;
static unsigned int odd_even_botCount = 0, maxUnfoldingLevel;
unsigned int odd_even_bot(unsigned int x) {
    ++odd_even_botCount;
    if (odd_even_botCount == maxUnfoldingLevel) throw 1;
    if (x == 0) return 1;
    else if (x % 2) return odd_even_bot(x + 1); else return odd_even_bot(x / 2);
}
int odd_even_bot_unfold(unsigned int maxUnfoldingLevel = 100, unsigned int maxParam = 10) {
    bool bottom = false; unsigned int result; ::maxUnfoldingLevel = maxUnfoldingLevel;
    cout << "COMPUTING LFP of oe(x) = x equals zero -> one "
         << "[] (odd x -> oe(x plus one) [] oe(x div two))" << endl;
    for (unsigned int n = 0; n < maxParam; ++n) {
        try { bottom = false; odd_even_botCount = 0; result = odd_even_bot(n); }
        catch (int) { bottom = true; }
        cout << "odd_even_bot(" << n << ") = " << (int)((bottom) ? -1 : result) << " in "
             << odd_even_botCount << " unfolds" << endl;
    }
    cout << endl << endl;
    return 0;
}
```

Unfolding of Simple Simultaneous by Simulation

// Project: PoPL 2019. f_g_simple_unfold.cpp, $\perp$ shown as -1

$f = \lambda x.x \textit{ equals zero} \rightarrow g(\textit{zero}) \; [] \; f(g(x)) \textit{ plus two}$

$g = \lambda y.y \textit{ equals zero} \rightarrow \textit{zero} \; [] \; y \textit{ times } f(y)$

```
COMPUTING LFP of f_g_simple
f(x) = x equals zero->g(zero)[] f(g(x)) plus two
g(y) = y equals zero -> zero [] y times f(y)
g_s(0) = 0 in 1 unfolds
g_s(1) = -1 in 1001 unfolds
g_s(2) = -1 in 1001 unfolds
g_s(3) = -1 in 1001 unfolds
g_s(4) = -1 in 1001 unfolds
g_s(5) = -1 in 1001 unfolds
...
g_s(9) = -1 in 1001 unfolds

f_s(0) = 0 in 2 unfolds
f_s(1) = -1 in 1001 unfolds
f_s(2) = -1 in 1001 unfolds
f_s(3) = -1 in 1001 unfolds
f_s(4) = -1 in 1001 unfolds
f_s(5) = -1 in 1001 unfolds
...
f_s(9) = -1 in 1001 unfolds
```

Hence, $f = g = \lambda x.x \textit{ equals zero} \rightarrow \textit{zero} \; [] \; \perp$

Unfolding of Simple Simultaneous by Simulation

// Project: PoPL 2019. f_g_simple_unfold.cpp, $\perp$ shown as -1

```
//f(x) = x equals zero -> g(zero) [] f(g(x)) plus two
//g(y) = y equals zero -> zero [] y times f(y)

#include <iostream>
using namespace std;
static unsigned int fCount = 0, gCount = 0, maxUnfoldingLevel;
unsigned int g_s(unsigned int x);
//f = x:x equals zero -> g(zero) [] f(g(x)) plus two
unsigned int f_s(unsigned int x) {
    ++fCount;
    if (fCount + gCount > maxUnfoldingLevel) throw 1;
    if (x == 0) { // return g(0);
        try { int t = g_s(0); return t; } catch (int) { throw; }
    }
    else try { int t = g_s(x); t = f_s(t); return t + 2; } catch (int) { throw; }
}
//g = y : y equals zero -> zero [] y times f(y)
unsigned int g_s(unsigned int x) { ++gCount;
    if (fCount + gCount > maxUnfoldingLevel) throw 2;
    if (x == 0) return 0;
    else try { int t = f_s(x); return x * t; } catch (int) { throw; }
}
```

Unfolding of Simple Simultaneous by Simulation

// Project: PoPL 2019. f_g_simple_unfold.cpp, $\perp$ shown as -1

```
//f(x) = x equals zero -> g(zero) [] f(g(x)) plus two
//g(y) = y equals zero -> zero [] y times f(y)

int f_g_simple_unfold(unsigned int maxUnfoldingLevel = 100, unsigned int maxParam = 10) {
    bool bottom = false; unsigned int gResult, fResult; ::maxUnfoldingLevel = maxUnfoldingLevel;
    cout << "COMPUTING LFP of f_g_simple" << endl;
    cout << "f(x) = x equals zero->g(zero)[] f(g(x)) plus two" << endl;
    cout << "g(y) = y equals zero -> zero [] y times f(y)" << endl;
    for (unsigned int n = 0; n < maxParam; ++n) {
        try { bottom = false; fCount = gCount = 0; gResult = g_s(n); } catch (int) { bottom = true; }
        cout << "g_s(" << n << ") = " << (int)((bottom) ? -1 : gResult) << " in "
             << fCount + gCount << " unfolds" << endl;
    }
    cout << endl;
    for (unsigned int n = 0; n < maxParam; ++n) {
        try { bottom = false; fCount = gCount = 0; fResult = f_s(n); } catch (int) { bottom = true; }
        cout << "f_s(" << n << ") = " << (int)((bottom) ? -1 : fResult) << " in "
             << fCount + gCount << " unfolds" << endl;
    }
    cout << endl << endl;
    return 0;
}
```

Recursive Functions Definitions: **The While Loop**

Specification of the semantics of **while** loop:

$\mathbf{C}[[\mathbf{while}\ B\ \mathbf{do}\ C]] = \underline{\lambda} s.\mathbf{B}[[B]]s \rightarrow \mathbf{C}[[\mathbf{while}\ B\ \mathbf{do}\ C]](\mathbf{C}[[C]]s)\ []\ s$

In terms of *fix* operations:
$\mathbf{C}[[\mathbf{while}\ B\ \mathbf{do}\ C]] = fix(\lambda f.\underline{\lambda} s.\mathbf{B}[[B]]s \rightarrow f(\mathbf{C}[[C]]s)\ []\ s)$

The functional is $Store_{\perp} \rightarrow Store\perp$, where $Store = Id \rightarrow Nat$

Recursive Functions Definitions: **The While Loop**

- Let us unfold the loop $\mathbf{C}[[\textbf{while } A > 0 \textbf{ do } (A := A - 1; B := B + 1)]]$ and capture the transformation to the $Store_{\perp}$ at every stage. The functional is:
  $F = \lambda f.\underline{\lambda} s.test\ s \rightarrow f(adjust\ s)\ []\ s$ where $test = \mathbf{B}[[A > 0]]$ and
  $adjust = \mathbf{C}[[A := A - 1; B := B + 1]]$
- To unfold the functional of the while loop, we need to compute on the store s
- Let us assume that initially the store is: $s_0 = \lambda i.zero$ That is, all identifiers are initialized to 0
- Additionally, we may assume that before entry to the loop, the store may have been changed (for A, B, as well as other identifiers) to s_{loop_start}, where A and B may have any pair of Nat values that will impact the computation of F^i
- Now only identifiers A and B are involved in the computation of F^i. Hence, at some stage of the loop if A has value a and B has value b, the store is:
  $s = \lambda i.i\ equals\ A \rightarrow a\ []\ i\ equals\ B \rightarrow b\ []\ s_{loop\_start}$
- For the purpose of computation of F^i, we can represent this s as a pair (a, b) that actually stands for an infinite class of mappings for s as given by all possible mappings for s_{loop_start}
- We enumerate all such pairs (a, b) at every loop entry and loop exit during unfolding

Recursive Functions Definitions: **The While Loop**

Let us unfold the loop $\mathbf{C}[[\mathbf{while}\ A > 0\ \mathbf{do}\ (A := A - 1; B := B + 1)]]$ and capture the transformation to the $Store_\perp$ at every stage. The functional is: $F = \lambda f.\underline{\lambda} s.test\ s \to f(adjust\ s)\ []\ s$, where $test = \mathbf{B}[[A > 0]]$ and $adjust = \mathbf{C}[[A := A - 1; B := B + 1]]$

On loop Entry (A, B) → **On loop Exit** (A, B)

$\to_0$

Entry:

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

Exit:

$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$\to_1$

Entry:

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

Exit:

0,0	0,1	0,2	0,3	0,4
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$\to_2$

Entry:

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

Exit:

0,0	0,1	0,2	0,3	0,4
0,1	0,2	0,3	0,4	0,5
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$\to_3$

Entry:

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

Exit:

0,0	0,1	0,2	0,3	0,4
0,1	0,2	0,3	0,4	0,5
0,2	0,3	0,4	0,5	0,6
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Recursive Functions Definitions: **The While Loop**

Consider: $\mathbf{C}[[\mathbf{while}\ A > 0\ \mathbf{do}\ (A := A - 1; B := B + 1)]]$
Let $test = \mathbf{B}[[A > 0]]$ and $adjust = \mathbf{C}[[A := A - 1; B := B + 1]]$

The functional is: $F = \lambda f.\underline{\lambda} s.test\ s \rightarrow f(adjust\ s)\ []\ s$

$graph(F^0(\perp)) = \{\}$
$graph(F^1(\perp)) = \{$
$(\{([[A]], zero), ([[B]], zero), \cdots\}, \{([[A]], zero), ([[B]], zero), \cdots\}), \cdots,$
$(\{([[A]], zero), ([[B]], four), \cdots\}, \{([[A]], zero), ([[B]], four), \cdots\}), \cdots\}$

Since the result is a member of $Store_\perp \rightarrow Store_\perp$, $graph(F^1(\perp))$ contains pairs of function graphs. Each pair shows a store prior to its *loop entry* and the store after *loop exit*. The members shown in the graph at this step are those stores whose $[[A]]$ value equals zero. Thus, those stores that already map $[[A]]$ to zero fail the test upon loop entry and exit immediately. The store is left unchanged. Those stores that require loop processing are mapped to $\perp$.

$graph(F^2(\perp)) = \{$
$(\{([[A]], zero), ([[B]], zero), \cdots\}, \{([[A]], zero), ([[B]], zero), \cdots\}), \cdots,$
$(\{([[A]], zero), ([[B]], four), \cdots\}, \{([[A]], zero), ([[B]], four), \cdots\}), \cdots,$
$(\{([[A]], one), ([[B]], zero), \cdots\}, \{([[A]], zero), ([[B]], one), \cdots\}), \cdots,$
$(\{([[A]], one), ([[B]], four), \cdots\}, \{([[A]], zero), ([[B]], five), \cdots\}), \cdots\}$

Recursive Functions Definitions: **The While Loop**

$graph(F^2(\perp)) = \{$
$(\{([[A]], zero), ([[B]], zero), \cdots\}, \{([[A]], zero), ([[B]], zero), \cdots\}), \cdots,$
$(\{([[A]], zero), ([[B]], four), \cdots\}, \{([[A]], zero), ([[B]], four), \cdots\}), \cdots,$
$(\{([[A]], one), ([[B]], zero), \cdots\}, \{([[A]], zero), ([[B]], one), \cdots\}), \cdots,$
$(\{([[A]], one), ([[B]], four), \cdots\}, \{([[A]], zero), ([[B]], five), \cdots\}), \cdots\}$

Those input stores that require one or fewer iterations to process appear in the graph. For example, the fourth illustrated pair denotes a store that has $[[A]]$ set to one and $[[B]]$ set to $four$ upon loop entry. Only one iteration is needed to reduce $[[A]]$ down to $zero$, the condition for loop exit. In the process $[[B]]$ is incremented to $five$:

$graph(F^3(\perp)) = \{$
$(\{([[A]], zero), ([[B]], zero), \cdots\}, \{([[A]], zero), ([[B]], zero), \cdots\}), \cdots,$
$(\{([[A]], zero), ([[B]], four), \cdots\}, \{([[A]], zero), ([[B]], four), \cdots\}), \cdots,$
$(\{([[A]], one), ([[B]], zero), \cdots\}, \{([[A]], zero), ([[B]], one), \cdots\}), \cdots,$
$(\{([[A]], one), ([[B]], four), \cdots\}, \{([[A]], zero), ([[B]], five), \cdots\}), \cdots,$
$(\{([[A]], two), ([[B]], zero), \cdots\}, \{([[A]], zero), ([[B]], two), \cdots\}), \cdots,$
$(\{([[A]], two), ([[B]], four), \cdots\}, \{([[A]], zero), ([[B]], six), \cdots\}), \cdots\}$

Recursive Functions Definitions: **The While Loop**

$$
\begin{aligned}
&graph(F^3(\bot)) = \{ \\
&(\{([[A]], zero), ([[B]], zero), \cdots\}, \{([[A]], zero), ([[B]], zero), \cdots\}), \cdots, \\
&(\{([[A]], zero), ([[B]], four), \cdots\}, \{([[A]], zero), ([[B]], four), \cdots\}), \cdots, \\
&(\{([[A]], one), ([[B]], zero), \cdots\}, \{([[A]], zero), ([[B]], one), \cdots\}), \cdots, \\
&(\{([[A]], one), ([[B]], four), \cdots\}, \{([[A]], zero), ([[B]], five), \cdots\}), \cdots, \\
&(\{([[A]], two), ([[B]], zero), \cdots\}, \{([[A]], zero), ([[B]], two), \cdots\}), \cdots, \\
&(\{([[A]], two), ([[B]], four), \cdots\}, \{([[A]], zero), ([[B]], six), \cdots\}), \cdots\}
\end{aligned}
$$

All stores that require two iterations or less for processing are included in the graph. The $graph(F^{i+1}(\bot))$ contains those pairs whose input stores finish processing in i iterations or less. The least fixed point of the functional contains mappings for those stores that conclude their loop processing in a finite number of iterations.

Recursive Functions Definitions: **The While Loop**

The while-loop's semantics makes a good example for restating the important principle of least fixed point semantics:

> *The meaning of a recursive specification is totally determined by the meanings of its finite subfunctions. Each subfunction can be represented non-recursively in the function notation.*

In this case:

$$
\begin{aligned}
&\mathbf{C}[[\mathbf{while}\ B\ \mathbf{do}\ C]] = \sqcup\ \{\underline{\lambda} s.\bot,\\
&\quad \underline{\lambda} s.\mathbf{B}[[B]]s \rightarrow \bot\ []\ s,\\
&\quad \underline{\lambda} s.\mathbf{B}[[B]]s \rightarrow (\mathbf{B}[[B]](\mathbf{C}[[C]]s) \rightarrow \bot\ []\ \mathbf{C}[[C]]s)\ []\ s,\\
&\quad \underline{\lambda} s.\mathbf{B}[[B]]s \rightarrow (\mathbf{B}[[B]](\mathbf{C}[[C]]s) \rightarrow\\
&\qquad\qquad (\mathbf{B}[[B]](\mathbf{C}[[C]](\mathbf{C}[[C]]s))\ \rightarrow \bot\ []\ \mathbf{C}[[C]](\mathbf{C}[[C]]s))\ []\ \mathbf{C}[[C]]s),\\
&\quad \cdots, \}
\end{aligned}
$$

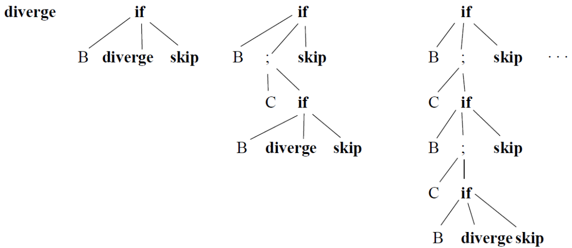
The family of expressions makes apparent that iteration is an unwinding of a loop body; this corresponds to the operational view

Recursive Functions Definitions: **The While Loop**

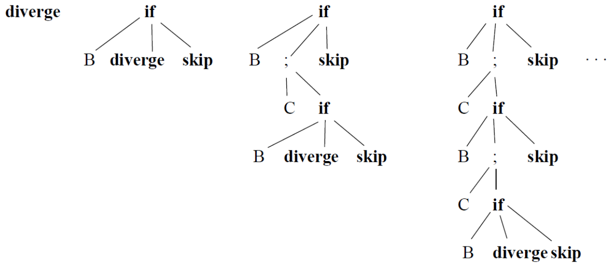
Can we restate this idea even more directly? Recall that $\mathbf{C}[[\textbf{diverge}]] = \lambda s.\bot$. Substituting the commands into the set just constructed gives us:

$$
\begin{aligned}
\mathbf{C}[[\textbf{while } B \textbf{ do } C]] = \sqcup \{ & \mathbf{C}[[\textbf{diverge}]], \\
& \mathbf{C}[[\textbf{if } B \textbf{ then diverge else skip}]], \\
& \mathbf{C}[[\textbf{if } B \textbf{ then } (C;\ \textbf{if } B \textbf{ then diverge else skip}) \textbf{ else skip}]], \\
& \mathbf{C}[[\textbf{if } B \textbf{ then } (C; \textbf{if } B \textbf{ then} \\
& \qquad (C;\ \textbf{if } B \textbf{ then diverge else skip}) \textbf{ else skip}) \textbf{ else skip}]], \cdots \}
\end{aligned}
$$

A family of finite non-iterative programs represents the loop. It is easier to see what is happening by drawing the abstract syntax trees:



Recursive Functions Definitions: **The While Loop**



At each stage, the finite tree becomes larger and better defined. The obvious thing to do is to place a partial ordering upon the trees: for all commands C, **diverge** $\sqsubseteq C$, and for commands C_1 and C_2, $C_1 \sqsubseteq C_2$ iff C_1 and C_2 are the same command type (have the same root node) and all subtrees in C_1 are less defined than the corresponding trees in C_2. This makes families of trees like the one above into chains. What is the lub of such a chain? It is the infinite tree corresponding to:

if B **then** $(C;$ **if** B **then** $(C;$ **if** B **then** $(\mathbf{C}; \cdots)$ **else skip**$)$ **else skip**$)$ **else skip**

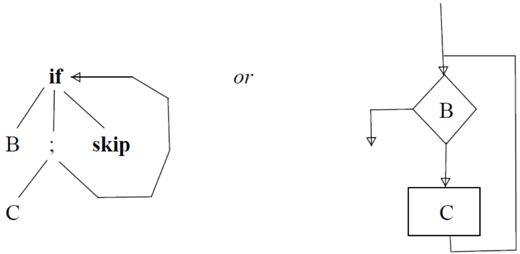
Recursive Functions Definitions: **The While Loop**

Draw this tree, and define $L = \textbf{if } B \textbf{ then } (C; L) \textbf{ else skip}$.

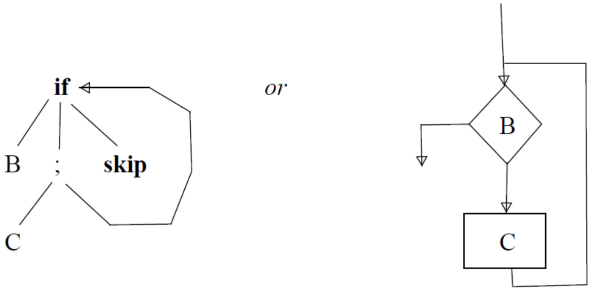
The while-loop example has led researchers to study languages that contain infinite programs that are represented by recursive definitions, such as L. The goal of such studies is to determine the semantics of recursive and iterative constructs by studying their circularity at the syntax level. The fundamental discovery of this research is that, whether the recursion is handled at the syntax level or at the semantics level, the result is the same:

$$\mathbf{C}[[\textbf{while } B \textbf{ do } C]] = \mathbf{C}[[L]]$$

Finally, the infinite tree L is abbreviated:



Recursive Functions Definitions: **The While Loop**



Every flowchart loop can be read as an abbreviation for an infinite program. This brings us back to representations of functions again, for the use of finite loops to represent infinite flowcharts parallels the use of finite function expressions to denote infinite objects – functions. The central issue of computability theory might be stated as the search for finite representations of infinite objects.

Languages with Context

Language with Contexts

- Languages rely on notions of context which influences the meanings of phrases – attributes meanings to identifiers
- Programming language contexts can have different notions

Store as Context

Store establishes the context for a phrase – but it does suggest that the context within the block is constantly changing which is counter-intuitive. Surely the declarations of the identifiers X and Y establish the context of the block, and the commands within the block operate within that context.

```
begin
    integer X; integer Y;
    Y:=0;  // X = bot, Y = 0
    X:=Y;  // X = 0, Y = 0
    Y:=1;  // X = 0, Y = 1
    X:=Y+1 // X = 2, Y = 1
end
```

Block as Context

The meaning of an identifier is not just its storeable value. There are two definitions of X – outer (inner) is an integer (real) object. Any ambiguity in using X is handled by the scope rules. These are actually computer storage locations, and the primary meaning of an identifier is the location bound to it.

```
begin integer X;
    X:=0;          // integer X
    begin real X;
        X:=1.5     // real X
    end;
    X:=X+1         // integer X
end
```

Language with Contexts: Environment

The context we choose to use is

- the set of identifier and storage location pairs (typically referred to as *binding*) that are
- *accessible* at a textual position

Each position in the program

- resides within a unique context, and
- the context can be determined without running the program

Language with Contexts: Environment

In denotational semantics, the context of a phrase is modeled by a value called an **environment**. Environments possess several distinctive properties:

[1] An environment establishes a context for a syntactic phrase, resolving any ambiguities concerning the meaning of identifiers.

[2] There are as many environment values as there are distinct contexts in a program. Multiple environments may be maintained during program evaluation.

[3] An environment is (usually) a static object. A phrase uses the same environment each time it is evaluated with the store.

Language with Contexts: Environment & Store

- An environment argument was not needed for the languages so far, because the programs in the languages used exactly one environment
- The single *Environment* was *pasted onto* the *Store*, giving a map from *Identifiers* to *Storable Values*
- Now, that simple model is split apart into two separate components:
  - the *Environment* and
  - the *Store*)
- *Identifiers* map to *Locations* (in *Environment*) and *Locations* map to *Storeable Values* (in *Store*)

Language with Contexts: Symbol Table

- The primary real-life example of an environment is a compiler's symbol table used to translate a source program into compiled code
- The symbol table contains an entry for each identifier in the program, listing:
  - the identifier's data type,
  - its mode of usage (variable, constant, parameter, . . .), and
  - its relative location in the run-time computer store
- Since a block-structured language allows multiple uses of the same identifier, the symbol table is responsible for resolving naming conflicts.

Language with Contexts: Symbol Table

- The schemes for implementation are many:
  - one is to keep a different symbol table for each block of the program (the portions in common between blocks may be shared);
  - another is to build the table as a single stack, which is incremented and decremented upon respective entry and exit for a block.
- Symbol tables may be
  - compile-time objects, as in ALGOL68, standard Pascal, C, C++, or
  - run-time objects, as in SNOBOL4, LISP or
  - used in both phases, as in ALGOL60, Java, Python

Language with Contexts: Static & Dynamic Semantics

- Those portions of a semantics definition that use an environment to resolve context questions are sometimes called the **Static Semantics**
  - The term traditionally describes compile-time actions such as type-checking, scope resolution, and storage calculation
- Static semantics may be contrasted with the *real* production of meaning, which takes the name **Dynamic Semantics**
  - Code generation and execution comprise the implementation-oriented version of dynamic semantics
- In general, the separation of static from dynamic semantics is rarely clear cut, and will be skipped here

Language with Contexts: Commands

- Environments are used as arguments by the valuation functions. The meaning of a command is now determined by the function:

$$\mathbf{C} : \text{Command} \rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow \mathit{Store}_\perp$$

instead of the earlier:

$$\mathbf{C} : \text{Command} \rightarrow \mathit{Store} \rightarrow \mathit{Store}_\perp$$

- The meaning of a command as a $\mathit{Store} \rightarrow \mathit{Store}_\perp$ function is determined once an environment establishes the context for the command.
- An environment belongs to the domain:

$$\mathit{Environment} = \mathit{Identifier} \rightarrow \mathit{Denotable\_value}$$

- The *Denotable_value* domain contains all the values that identifiers may represent.
- This domain varies widely from language to language and its structure largely determines the character of the language

Language with Contexts: Block-structured and Applicative languages

- We study language features whose semantics are understood in terms of environments
- These features include:
  - declarations,
  - block structure,
  - scoping mechanisms,
  - recursive bindings, and
  - compound data structures
- The concepts are covered within the framework of two languages:
  - an imperative block-structured language and
  - an applicative language

Languages with Context: Block Structured Languages

Block Structured Language: Abstract Syntax

Abstract Syntax:

- $P \in Program$
  $K \in Block$
  $D \in Declaration$
  $C \in Command$
  $E \in Expression$
  $B \in Boolean\_expr$
  $I \in Identifier$
  $N \in Numeral$

  $P ::= K.$
  $K ::= \textbf{begin}\ D;\ C\ \textbf{end}$
  $D ::= D_1;\ D_2 \mid \textbf{const}\ I = N \mid \textbf{var}\ I$
  $C ::= C_1;\ C_2 \mid I := E \mid \textbf{while}\ B\ \textbf{do}\ C \mid K$
  $E ::= E_1 + E_2 \mid I \mid N$
  $B ::= E_1 = E_2 \mid \neg B$

Block Structured Language: Semantic Algebras

Semantic Algebras:

- *Truth values*
  Domain: $t \in Tr = B$
  Operations:
  $true, false : Tr$
  $not : Tr \rightarrow Tr$
- *Natural Numbers*
  Domain: $n \in Nat = \mathcal{N}$
  Operations:
  $zero, one, \ldots : Nat$
  $plus : Nat \times Nat \rightarrow Nat$
  $equals : Nat \times Nat \rightarrow Tr$
- *Identifiers*
  Domain: $i \in Id = Identifier$

Block Structured Language: Semantic Algebras

- *Expressible value*
  Domain: $x \in Expressible\_value = Nat + Errvalue$
  where $Errvalue = Unit$
  - Expressible value errors occur when an expressible value is inappropriately used
  - For example, a truth value is added to a natural number expression

Block Structured Language: Semantic Algebras

- *Storage Location*
  Domain: $l \in Location$
  Operations:
  $first\_locn : Location$
  $next\_locn : Location \rightarrow Location$
  $equal\_locn : Location \rightarrow Location \rightarrow Tr$
  $lessthan\_locn : Location \rightarrow Location \rightarrow Tr$

  - *first_locn* is a constant, marking the first usable location in a store
  - *next_locn* maps a location to its immediate successor in a store
  - *equal_locn* checks for equality of two values, and
  - *lessthan_locn* compares two locations and returns a truth value based on the locations' relative values

Block Structured Language: Semantic Algebras

- *Denotable values*
  Domain: $d \in Denotable\_value = Location + Nat + Errvalue$
  where $Errvalue = Unit$

  - Of the three components of the *Denotable_value* domain:
    - *Location* holds the denotations of variable identifiers,
    - *Nat* holds the meanings of constant identifiers, and
    - *Errvalue* holds the meaning for undeclared identifiers
  - Since the *Denotable_value* domain contains both natural numbers and locations, denotable value errors may occur in a program; for example, an identifier with a number denotation might be used where an identifier with a location denotation is required
  - An identifier with an erroneous denotable value always induces an error

Block Structured Language: Semantic Algebras

- *Environment*: a map to denotable values and the maximum store location

  Domain: $e \in Environment = (Id \rightarrow Denotable\_value) \times Location$

  Operations:

  $emptyenv : Location \rightarrow Environment$

  $\quad emptyenv = \lambda l.((\lambda l.inErrvalue()), l)$

  $accessenv : Id \rightarrow Environment \rightarrow Denotable\_value$

  $\quad accessenv = \lambda i.\lambda(map, l).map(i)$

  $updateenv : Id \rightarrow Denotable\_value \rightarrow$

  $\qquad\qquad Environment \rightarrow Environment$

  $\quad updateenv = \lambda i.\lambda d.\lambda(map, l).([i \mapsto d]map, l)$

  $reserve\_locn : Environment \rightarrow (Location \times Environment)$

  $\quad reserve\_locn = \lambda(map, l).(l, (map, next\_locn(l)))$

Block Structured Language: Semantic Algebras

- An environment is a pair
  - The first component is the function that maps identifiers to their denotable values
  - The second component is a location value, which marks the extent of the store reserved for declared variables
- The environment takes the responsibility for assigning locations to variables. This is done by the *reserve_locn* operation, which returns the next usable location
- *Although it is not made clear by the algebra, the structure of the language will cause the locations to be used in a* **stack-like fashion**
- The *emptyenv* must be given the location marking the beginning of usable space in the store so that it can build the initial environment

Block Structured Language: Semantic Algebras

- *Storable values*
  Domain: $v \in Storable\_value = Nat$
- *Store*
  Domain: $s \in Store = Location \to Storable\_value$
  Operations:
  $access : Location \to Store \to Storable\_value$
  $access = \lambda(l, s).s(l)$
  $update : Location \to Storable\_value \to Store \to Store$
  $update = \lambda(l, v, s).[l \mapsto v]s$

  - The store is a map from storage locations to storable values, and the operations are the obvious ones
  - Errors during evaluation are possible, so the store will be labeled with the status of the evaluation
  - The *check* operation uses the tags to determine if evaluation should continue

Block Structured Language: Semantic Algebras

- *Run-time store, labeled with status of computation*

  Domain: $p \in Poststore = OK + Err$

  where $OK = Err = Store$

  Operations:

  $return : Store \rightarrow Poststore$

  $\quad return = \lambda s.inOK(s)$

  $signalerr : Store \rightarrow Poststore$

  $\quad signalerr = \lambda s.inErr(s)$

  $check : (Store \rightarrow Poststore_\perp) \rightarrow$

  $\qquad (Poststore_\perp \rightarrow Poststore_\perp)$

  $\quad check\ f = \underline{\lambda} p.\ cases\ p\ of$

  $\qquad isOK(s) \rightarrow (f\ \ s)\ [\!]\ isErr(s) \rightarrow p\ end$

Block Structured Language: Valuation Functions

Valuation Functions:

- **P**: Program$\rightarrow$ *Location* $\rightarrow$ *Store* $\rightarrow$ *Poststore*$_\perp$

  $$\mathbf{P}[[K.]] = \lambda l.\mathbf{K}[[K]]\ (emptyenv\ \ l)$$

  - The **P** valuation function requires a store and a location value, the latter marking the beginning of the store's free space
- **K**: Block$\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ *Poststore*$_\perp$

  $$\mathbf{K}[[\textbf{begin}\ D; C\ \textbf{end}]] = \lambda e.\mathbf{C}[[C]](\mathbf{D}[[D]]e)$$

  - The **K** function establishes the context for a block

Block Structured Language: Valuation Functions

Valuation Functions:

- **D**: Declaration→ *Environment* → *Environment*

$$\mathbf{D}[[D_1; D_2]] = \mathbf{D}[[D_2]] \circ \mathbf{D}[[D_1]]$$

  - The **D** function augments an environment
  - The composition of declarations parallels the composition of commands

$$\mathbf{D}[\mathbf{const}\ I = N]] = \mathit{updateenv}\ [[I]]\ \mathit{inNat}(\mathbf{N}[[N]])\ e$$

  - A constant identifier declaration causes an environment update, where the identifier is mapped to its numeral value in the environment

Block Structured Language: Valuation Functions

Valuation Functions:

- **D**: Declaration→ *Environment* → *Environment*

$$\mathbf{D}[\mathbf{var}\ I]] = \lambda e.let(l', e') = \\ (reserve\_locn\ e)\ in\ (updateenv\ [[I]]\ inLocation(l')\ e')$$

  - The denotation of a variable declaration is more involved: a new location is reserved for the variable
  - This location, l', plus the current environment, e', are used to create the environment in which the variable $[[I]]$ binds to $inLocation(l')$
  - What happens on duplicate declaration of the same identifier in the same block?

Block Structured Language: Valuation Functions

Valuation Functions:

- **C**: Command→ *Environment* → *Store* → *Poststore*$_\perp$

$$\mathbf{C}[[C_1;\ C_2]] = \lambda e.(check(\mathbf{C}[[C_2]]e)) \circ (\mathbf{C}[[C_1]]e)$$

  - First, consider the *check* operation. If command $\mathbf{C}[[C_1]]e$ maps a store into an erroneous *Poststore*, then *check* traps the error and prevents $\mathbf{C}[[C_2]]e$ from altering the store
  - Note that the commands $[[C_1]]$ and $[[C_2]]$ are both evaluated in the context represented by e. This is important, for $[[C_1]]$ could be a block with local declarations that would need its own local environment to process its commands while $\mathbf{C}[[C_2]]$ retains its own copy of e. (Of course, whatever alterations $\mathbf{C}[[C_1]]e$ makes upon the store are passed to $\mathbf{C}[[C_2]]e$.)
  - This language feature is called **static scoping**. The context for a phrase in a statically scoped language is determined solely by the textual position of the phrase and any identifier declared within a block may be referenced only by the commands within that block
  - **Dynamically scoped** languages, whose contexts are not totally determined by textual position, will be discussed later.

Block Structured Language: Valuation Functions

Valuation Functions:

- **C**: Command$\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ *Poststore*$_\perp$

$$
\begin{aligned}
&\mathbf{C}[[I := E]] = \lambda e.\lambda s.cases\ (accessenv\ [[I]]\ e)\ of\\
&\quad isLocation(l) \rightarrow (cases(\mathbf{E}[[E]]e\ s)\ of\\
&\qquad\quad isNat(n) \rightarrow (return(update\ l\ n\ s))\\
&\qquad\quad [\,]\ isErrValue() \rightarrow (signalerr\ s)\ end)\\
&\quad [\,]\ isNat(n) \rightarrow (signalerr\ s)\\
&\quad [\,]\ isErrValue() \rightarrow (signalerr\ s)\ end
\end{aligned}
$$

  - Note that, if identifier *I* has not been declared in this environment, then *accessenv* will return a *Errvalue* as the *Denotational_value*. On this, a *signalerr* is rightly done putting the store *s* as *Err*

Block Structured Language: Valuation Functions

Valuation Functions:

- **C**: Command$\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ *Poststore*$_\perp$

$$\mathbf{C}[[\mathbf{while}\ B\ \mathbf{do}\ C]] = \lambda e.fix(\lambda f.\lambda s.\ cases\ (\mathbf{B}[[B]]e\ s)\ of \\ isTr(t) \rightarrow (t \rightarrow (check\ f) \circ (\mathbf{C}[[C]]e)\ []\ return)(s) \\ []\ isErrValue(l) \rightarrow (signalerr\ s)\ end)$$

$$\mathbf{C}[[K]] = \mathbf{K}[[K]]$$

Block Structured Language: Valuation Functions

Valuation Functions:

- **E**: Expression$\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ *Expressible_value*

$$
\begin{aligned}
&\mathbf{E}[[E_1 + E_2]] = \lambda e.\lambda s.cases\ (\mathbf{E}[[E_1]]e\ s)\ of \\
&\qquad [\,]\ isNat(n_1) \rightarrow (cases\ (\mathbf{E}[[E_2]]e\ s)\ of \\
&\qquad\quad isNat(n_2) \rightarrow inNat(n_1\ plus\ n_2) \\
&\qquad\quad [\,]\ isErrvalue() \rightarrow inErrvalue()\ end) \\
&\qquad [\,]\ isErrvalue() \rightarrow inErrvalue()\ end
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{E}[[I]] = \lambda e.\lambda s.cases\ (accessenv\ [[I]]\ e)\ of \\
&\qquad isLocation(l) \rightarrow inNat(access\ l\ s) \\
&\qquad [\,]\ isNat(n) \rightarrow inNat(n) \\
&\qquad [\,]\ isErrValue() \rightarrow inErrvalue()\ end
\end{aligned}
$$

$$
\mathbf{E}[[N]] = \lambda e.\lambda s.inNat(\mathbf{N}[[N]])
$$

Block Structured Language: Valuation Functions

Valuation Functions:

- $\mathbf{B} : \mathit{Boolean\_expr} \rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow (\mathit{Tr} + \mathit{Errvalue})$

$$\begin{array}{l} \mathbf{B}[[E_1 = E_2]] = \lambda e.\lambda s.\mathit{cases}\ (\mathbf{E}[[E_1]]e\ s)\ \mathit{of} \\ \quad [\,]\ \mathit{isNat}(n_1) \rightarrow (\mathit{cases}\ (\mathbf{E}[[E_2]]e\ s)\ \mathit{of} \\ \quad\quad \mathit{isNat}(n_2) \rightarrow \mathit{inTr}(n_1\ \mathit{equals}\ n_2) \\ \quad\quad [\,]\ \mathit{isErrvalue}() \rightarrow \mathit{inErrvalue}()\ \mathit{end}) \\ \quad [\,]\ \mathit{isErrvalue}() \rightarrow \mathit{inErrvalue}()\ \mathit{end} \end{array}$$

$$\begin{array}{l} \mathbf{B}[[\neg B]] = \lambda e.\lambda s.\mathit{cases}\ (\mathbf{B}[[B]]e\ s)\ \mathit{of} \\ \quad [\,]\ \mathit{isTr}(t) \rightarrow \mathit{inTr}(\mathit{not}\ t) \\ \quad [\,]\ \mathit{isErrvalue}() \rightarrow \mathit{inErrvalue}()\ \mathit{end} \end{array}$$

- $\mathbf{N} : \mathit{Numeral} \rightarrow \mathit{Nat}\ (\mathit{omitted})$

Block Structured Language: Example

Perform the valuation for:

$\mathbf{P}[[\mathbf{begin}\ D_0;\ \ D_1;\ \ C_0\ \mathbf{end}]]$ where

$$
\begin{aligned}
D_0 &= \mathbf{const}\ A = 1 \\
D_1 &= \mathbf{var}\ X \\
C_0 &= C1;\ \ C2;\ \ C3 \\
C_1 &= X := A + 2 \\
C_2 &= \mathbf{begin\ var}\ A;\ \ C4\ \mathbf{end} \\
C_3 &= X := A \\
C_4 &= \mathbf{while}\ X = 0\ \mathbf{do}\ A := X
\end{aligned}
$$

```
begin
    const A = 1;
    var X;
    X := A + 2;
    begin
        var A;
        while X = 0 do A := X
    end
    X := A
end
```

Block Structured Language: Example

$\mathbf{P}[[\mathbf{begin}\ D_0;\ D_1;\ C_0\ \mathbf{end}]] =$

$\lambda I.\mathbf{K}[[\mathbf{begin}\ D_0;\ D_1;\ C_0\ \mathbf{end}]](emptyenv\ I) =$
where $e_0 = emptyenv\ I$

$\mathbf{C}[[C_0]](\mathbf{D}[[D_0;\ D_1]]e_0) =$

$\mathbf{D}[[D_0;\ D_1]]e_0 = \mathbf{D}[[D_1]](\mathbf{D}[[\mathbf{const}\ A = 1]]e_0)$

$\mathbf{D}[[D_0]] = \mathbf{D}[[\mathbf{const}\ A = 1]]e_0 = (updateenv\ [[A]]\ inNat(one)\ e_0) = e_1$
where $e_1 = [A \mapsto inNat(one)]$

$\mathbf{D}[[D_1]] = \mathbf{D}[[\mathbf{var}\ X]]e_1 = (updateenv\ [[X]]\ inLocation(l)\ e_2) = e_3$
where $let(l', e') = (reserve\_locn\ e_1)\ in\ e_2 = (l,\ (map, (next\_locn\ l)))$
$e_2 = [l.inErrvalue(),\ A \mapsto inNat(one)]$,
$e_3 = [X \mapsto inLocation(l),\ A \mapsto inNat(one)]$

Block Structured Language: Example

$(check(\mathbf{C}[[C_2;\ C_3]]e_3)) \circ (\mathbf{C}[[C_1]]e_3) =$

$(check(\mathbf{C}[[C_2;\ C_3]]e_3)) \circ (\mathbf{C}[[X := A + 2]]e_3) =$

$$\begin{aligned}\mathbf{C}[[X := A+2]] &= \lambda s.cases\ (accessenv\ [[X]]\ e_3)\ of\ isLocation(l) \rightarrow \\ &\quad (cases(\mathbf{E}[[A+2]]e_3\ s)\ of\ isNat(n) \rightarrow (return(update\ l\ n\ s))\ \ldots end)\ \ldots end = \\ &\quad inOK([l \mapsto three]s), \text{ where } e_3 = [X \mapsto inLocation(l),\ A \mapsto inNat(one)]\end{aligned}$$

$$\begin{aligned}\mathbf{E}[[A+2]]e_3 &= \lambda s.cases\ (\mathbf{E}[[A]]e_3\ s)\ of\ []\ isNat(n_1) \rightarrow (cases\ (\mathbf{E}[[2]]e_3\ s)\ of \\ &\quad isNat(n_2) \rightarrow inNat(n_1\ plus\ n_2)\ \ldots end)\ \ldots end = \\ &\quad inNat(one\ plus\ two) = inNat(three)\end{aligned}$$

$$\begin{aligned}\mathbf{E}[[A]]e_3 &= \lambda s.cases\ (accessenv\ [[A]]\ e_3)\ of\ isLocation(l) \rightarrow \\ &\quad inNat(access\ l\ s)\ []\ isNat(n) \rightarrow inNat(n)\ \ldots end = inNat(one)\end{aligned}$$

$$\mathbf{E}[[2]]e_3 = inNat(\mathbf{N}[[2]]) = inNat(two)$$

Block Structured Language: Example

$(check(\mathbf{C}[[C_2;\ C_3]]e_3)) =$

$(check(\mathbf{C}[[C_3]]e_3)) \circ (\mathbf{C}[[\mathbf{while}\ X = 0\ \mathbf{do}\ A := X]]e_3) =$

$\mathbf{C}[[\mathbf{while}\ X = 0\ \mathbf{do}\ A := X]]e_3) = \mathbf{C}[[C_4]](\mathbf{D}[[\mathbf{var}\ A]]e_3) =$

$\mathbf{C}[[\mathbf{while}\ X = 0\ \mathbf{do}\ A := X]]e_5 =$
$fix(\lambda f.\ \lambda s.\ cases\ (\mathbf{B}[[X = 0]]e_5\ s)\ of\ \ldots end$
$((access\ l\ s)\ equals\ zero \rightarrow (check\ f) \circ (\mathbf{C}[[A := X]]e_5)\ []\ return)\ s$
$\lambda s.\ return(update(next\_locn\ l)\ (access\ l\ s)\ s)$

$\mathbf{B}[[X = 0]]e_5\ s = inTr((access\ l\ s)\ equals\ zero) = false$

$\mathbf{D}[[\mathbf{var}\ A]]e_3 = (updateenv\ [[A]]\ inLocation(l)\ e_4) = e_5$
where $let(l', e') = (reserve\_locn\ e_3)\ in\ e_4 = (l,\ (map, (next\_locn\ l)))$
$e_4 = [X \mapsto inLocation(l),\ A \mapsto inNat(one),\ l_{inner}.inErrvalue()],$
$e_5 = [X \mapsto inLocation(l),\ A \mapsto inNat(one),\ A_{inner} \mapsto inLocation(l_{inner})]$

$\mathbf{C}[[C_3]]e_3 = \mathbf{C}[[X := A]]e_3$
$\lambda s.\ return(update\ l\ one\ s) = inOK([l \mapsto one]s)$

Block Structured Language: Example

Code blocks annotated with environment and store

```
begin
// e0 (empty). s0 (empty)

    const A = 1;
    // e1 (A -> one). s0 (empty)

    var X;
    // e2, e3 (A -> one, X -> loc(X)). s0 (empty)

    X := A + 2;
    // e3 (A -> one, X -> loc(X)). s1 (loc(X) -> three)

    begin
    // e3 (A -> one, X -> loc(X)). s1 (loc(X) -> three)

        var A;
        // e4, e5 (A -> one, X -> loc(X), Ain -> loc(Ain)). s1 (loc(X) -> three)

        while X = 0 do A := X
        // e5 (A -> one, X -> loc(X), Ain -> loc(Ain)). s1 (loc(X) -> three)

    end
    // e3 (A -> one, X -> loc(X)). s1 (loc(X) -> three)

    X := A
    // e3 (A -> one, X -> loc(X)). s2 (loc(X) -> one)

end
// e0 (empty). s0 (empty)
```

Block Structured Language: Stack-Managed Storage

- The store of a **block-structured language** is used in a **stack-like fashion**:
  - Locations are bound to identifiers sequentially using *next_locn*, and
  - A location bound to an identifier in a local block is freed for re-use when the block is exited
  - The re-use of locations happens automatically due to the equation for $\mathbf{C}[[C_1;\ C_2]]$
  - Any locations bound to identifiers in $[[C_1]]$ are reserved by the environment built from e for $\mathbf{C}[[C_1]]$, but $\mathbf{C}[[C_2]]$ re-uses the original e (and its original location marker), effectively deallocating the locations.

Block Structured Language: Stack-Managed Storage

- Stack-based storage is a significant characteristic of block-structured programming languages, and
- The **Store Algebra** deserves to possess mechanisms for stack-based allocation and deallocation
- Next we start to move the storage calculation mechanism over to the store algebra

Block Structured Language: Stack-Managed Storage: Semantic Algebras

- *Stack-based store*

  Domain: $s \in Store = (Location \to Storable\_value) \times Location$

  - The new store domain uses the

    $$Location \to Storable\_value$$

    component as the data space of the stack, and
  - the Location component indicates the amount of storage in use: it is the
    - ▷ *stack top* marker

Block Structured Language: Stack-Managed Storage: Semantic Algebras

- *Stack-based store*

  Domain: $s \in Store = (Location \rightarrow Storable\_value) \times Location$

  Operations:

  $access : Location \rightarrow Store \rightarrow (Storable\_value + Errvalue)$

  $\quad access = \lambda(l, s).\ s(l)$

  $update : Location \rightarrow Storable\_value \rightarrow Store \rightarrow Poststore$

  $\quad update = \lambda l.\ \lambda v.\ \lambda(map, top).\ l\ lessthan\_locn\ top \rightarrow$

  $\qquad inOK([l \mapsto v]map, top)\ []\ inErr(map, tops)$

  - Operations *access* and *update* verify that any reference to a storage location is a valid one, occurring at an active location beneath the stack top

Block Structured Language: Stack-Managed Storage: Semantic Algebras

- *Stack-based store*

  Domain: $s \in Store = (Location \to Storable\_value) \times Location$

  Operations:

  $mark\_locn : Store \to Location$
  $\quad mark\_locn = \lambda(map, top).top$

  $allocate\_locn : Store \to Location \times Poststore$
  $\quad allocate\_locn =$
  $\qquad \lambda(map, top).(top, inOK(map, next\_locn(top)))$

  - The purposes of *mark_locn* and *allocate_locn* should be clear; the latter is the run-time version of the environment's *reserve_locn* operation

Block Structured Language: Stack-Managed Storage: Semantic Algebras

- *Stack-based store*

  Operations:

  $$\begin{array}{l} deallocate\_locns : Location \rightarrow Store \rightarrow Poststore \\ \quad deallocate\_locns = \lambda l.\lambda(map, top). \\ \qquad (l\ lessthan\_locn\ top)\ or\ (l\ equal\_locn\ top) \rightarrow \\ \qquad\quad inOK(map, l)\ []\ inErr(map, top) \end{array}$$

  - The *deallocate_locns* operation releases stack storage from the stack top to the value indicated by its argument. Freed from storage management, the environment domain takes the form $Environment = Id \rightarrow Denotable\_value$
  - The operations are adjusted accordingly, and the operation *reserve_locn* is dropped
  - If the environment leaves the task of storage calculation to the store operations, then processing of declarations requires the store as well as the environment

Block Structured Language: Stack-Managed Storage: Valuation Functions

The functionality of the valuation function for declarations becomes:

- $\mathbf{D}$: Declaration$\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ (*Environment* $\times$ *Poststore*)

  $\mathbf{D}[\textbf{var } I]] = \lambda e.\lambda s.let(I, p) = (allocate\_locns)$
  $in\ ((updateenv\ [[I]]\ inLocation(I)\ e),\ \ p)$

  $\mathbf{D}[[D_1; D_2]] == \lambda e.\lambda s.let(e', p) = (\mathbf{D}[[D_1]]e\ s)\ in\ (check\ \mathbf{D}[[D_2]]e')(p)$

  $check :\ (Store \rightarrow (Environment \times Poststore)) \rightarrow$
  $(Poststore \rightarrow (Environment \times Poststore))$

  Earlier it was:
  $\mathbf{D}$: Declaration$\rightarrow$ *Environment* $\rightarrow$ *Environment*
  $\mathbf{D}[\textbf{var } I]] = \lambda e.let(I', e') =$
  $(reserve\_locn\ e)\ in\ (updateenv\ [[I]]\ inLocation(I')\ e')$

- This version of declaration processing makes the environment into a run-time object, for the binding of location values to identifiers cannot be completed without the run-time store
- Contrast this with the arrangement in the last model, where location binding is computed by the environment operation *reserve_locn*, which produced a result relative to an arbitrary base address
- A solution for freeing the environment from dependence upon *allocate_locn* is to provide it information about storage management strategies, so that the necessary address calculations can be performed independently of the value of the run-time store

Block Structured Language: Stack-Managed Storage: Valuation Functions

- **K**: Block$\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ *Poststore*$_\perp$

  The **K** function manages the storage for the block:

  $$\begin{array}{l}\mathbf{K}[[\mathbf{begin}\ D; C\ \mathbf{end}]] = \lambda e.\ \lambda s.\ let\ l = mark\_locn\ s\ in \\ \quad let\ (e', p) = \mathbf{D}[[D]]e\ s\ in \\ \quad let\ p' = (check(\mathbf{c}[[C]]e'))(p)\ in\ (check(deallocate\_locns\ l))(p')\end{array}$$

  Earlier, it was:

  K: Block$\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ *Poststore*$_\perp$

  $$\mathbf{K}[[\mathbf{begin}\ D; C\ \mathbf{end}]] = \lambda e.\mathbf{C}[[C]](\mathbf{D}[[D]]e)$$

  - The *deallocate_locns* operation frees storage down to the level held by the store prior to block entry, which is (*mark_locn s*)

Block Structured Language: Stack-Managed Storage: Meanings of Identifiers

- The notion of context can be even more subtle than we first imagined. Consider the Pascal assignment statement

```
X:=X+1
```

- The meaning of `X` on the right-hand side of the assignment is decidedly different from `X`'s meaning on the left-hand side. Specifically,
  - the *left-hand side value* is a *location value*, while
  - the *right-hand side value* is the *storable value* associated with that location.
  - Apparently the context problem for identifiers is found even at the primitive command level

Block Structured Language: Stack-Managed Storage: Meanings of Identifiers

- One way out of this problem would be to introduce two environment arguments for the semantic function for commands:
  - a *left-hand side one* and
  - a *right-hand side one*

  This arrangement is hardly natural; commands are the *sentences* of a program, and sentences normally operate in a *single context*

Block Structured Language: Stack-Managed Storage: Meanings of Identifiers

- Another option is to say that any variable identifier actually denotes a pair of values:
  - a *location value*, or,
    - identifier's *L-value* which is kept in the
    - *environment* and
  - a *storable value*, or,
    - identifier's *R-value* which is kept in the
    - *store*

Block Structured Language: Stack-Managed Storage: Meanings of Identifiers

- We introduce a valuation function

$$\mathbf{I} : \mathit{Id} \rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow (\mathit{Location} \times \mathit{Storable\_value})$$

. In practice, the **I** function is split into two semantic functions

$$\mathbf{L} : \mathit{Id} \rightarrow \mathit{Environment} \rightarrow \mathit{Location}$$

and

$$\mathbf{R} : \mathit{Id} \rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow \mathit{Storable\_value}$$

such that:
  - $\mathbf{L}[[I]] = \mathit{accessenv}\ [[I]]$
  - $\mathbf{R}[[I]] = \mathit{access} \circ \mathit{accessenv}\ [[I]]$
- We restate the semantic equations using variables as:

$$\mathbf{C}[[I := E]] = \lambda e.\ \lambda s.\ \mathit{return}(\mathit{update}(\mathbf{L}[[I]]e)(\mathbf{E}[[E]]e\ s)\ s)$$
$$\mathbf{E}[[I]] = \mathbf{R}[[I]]$$

Block Structured Language: Stack-Managed Storage: Meanings of Identifiers

- The definitions are a bit simplistic because they assume that all identifiers are variables.
  - Constant identifiers can be integrated into the scheme
  - a declaration such as $[[\textbf{const}\ A = N]]$ suggests $\mathbf{L}[[A]]e = inErrvalue()$
  - What should $(\mathbf{R}[[A]]e\ s)$ be?
- Yet another view to take is that the *R-value* of a variable identifier is a function of its *Lvalue*
- The *true meaning* of a variable is its *Lvalue*, and a *coercion* occurs when a variable is used on the right-hand side of an assignment
- This coercion is called *dereferencing*

Block Structured Language: Stack-Managed Storage: Meanings of Identifiers

- We formalize this view as:

$$\mathbf{J} : Id \rightarrow Environment \rightarrow Denotable\_value$$
$$\mathbf{J}[[I]] = \lambda e.\ (accessenv\ [[I]]\ e)$$
$$\mathbf{C}[[I := E]] = \lambda e.\ \lambda s.\ return(update(\mathbf{J}[[I]]e)(\mathbf{E}[[E]]e\ s)\ s)$$
$$\mathbf{E}[[I]] = \lambda e.\ \lambda s.\ dereference(\mathbf{J}[[I]]e)\ s$$

where

$$dereference : Location \rightarrow Store \rightarrow Storable\_value$$
$$dereference = access$$

Block Structured Language: Stack-Managed Storage: Meanings of Identifiers

- An identifier's meaning is just its denotable value
- Those identifiers with locations as their meanings (the variables) are dereferenced when an expressible value is needed.
- The implicit use of dereferencing is so common in general purpose programming languages that we take it for granted, despite the somewhat unorthodox appearance of commands such as

```
X = X + 1
```

in FORTRAN

Block Structured Language: Stack-Managed Storage: Meanings of Identifiers

- Systems-oriented programming languages such as BCPL, Bliss, and C use an explicit dereferencing operator
  - For example, in BCPL expressible values include locations, and the appropriate semantic equations are:

$$
\begin{aligned}
&\mathbf{E}[[I]] = \lambda e.\ \lambda s.\ \mathit{inLocation}(\mathbf{J}[[I]]e)\\
&\mathbf{E}[[@E]] = \lambda e.\ \lambda s.\ \mathit{cases}\ (\mathbf{E}[[E]]e\ s)\ \mathit{of}\\
&\qquad \mathit{isLocation}(l) \rightarrow (\mathit{dereference}\ l\ s)\\
&\qquad [] \ldots \mathit{end}
\end{aligned}
$$

  - The @ symbol is the dereferencing operator
  - The meaning of

```
X := X + 1
```

in BCPL is decidedly different from that of

```
X := @X+1
```

Languages with Context: Applicative Languages

Applicative Language

- An applicative language contains no variables
- All identifiers are constants and can be given attributes – but once, at their point of definition
- Without variables, mechanisms such as assignment are superfluous and are dropped
- Arithmetic is an applicative language
- Another example is the minimal subset of LISP known as *pure LISP*
- The function notation that we use to define denotational definitions can also be termed an applicative language
- Since an applicative language has no variables, its semantics can be specified without a *Store domain*
- The environment holds the attributes associated with the identifiers

Applicative Language

- It is similar to pure LISP – a list processing language
- A program in the language is just an expression
- An expression can be
  - a *LET* definition;
  - a *LAMBDA* form;
    - representing a function routine with parameter *I*
  - a function application;
  - a list expression using *CONS*, *HEAD*, *TAIL*, or *NIL*;
  - an identifier; or
  - an atomic symbol

Applicative Language: Abstract Syntax

Abstract Syntax:

- $E \in Expression$
  $A \in Atomic\_symbol$
  $I \in Identifier$

$$
\begin{aligned}
E ::= {} & LET\ I = E_1\ IN\ E_2 \mid \\
& LAMBDA\ (I)\ E \mid \\
& E_1\ E_2 \mid \\
& E_1\ CONS\ E_2 \mid HEAD\ E \mid TAIL\ E \mid NIL \mid \\
& I \mid \\
& A \mid (E)
\end{aligned}
$$

Note: $(let\ x = e_1\ in\ e_2)\ for\ (\lambda x.\ e_2)e_1$

Applicative Language: Semantic Algebras

Semantic Algebras:

- *Atomic answer values*
  Domain: $a \in Atom$
  Operations: (Omitted)
  - *Atom* is a primitive answer domain and its internal structure will not be considered
- *Identifiers*
  Domain: $i \in Id = Identifier$
  Operations: (Usual)

Applicative Language: Semantic Algebras

Semantic Algebras:

- *Denotable values, functions, and lists*
  Domain: $d \in Denotable\_value = (Function + List + Atom + Error)_{\perp}$
  $f \in Function = Denotable\_value \rightarrow Denotable\_value$
  $t \in List = Denotable\_value^{*}$
  $Error = Unit$

  - The language also contains a domain of functions, which map denotable values to denotable values; a denotable value can be a function, a list, or an atom
  - For the first time, we encounter a semantic domain defined in terms of itself. By substitution, we see that:

$$Denotable\_value = ((Denotable\_value \rightarrow Denotable\_value) + Denotable\_value^{*} + Atom + Error)_{\perp}$$

Applicative Language: Semantic Algebras

Semantic Algebras:

- *Expressible value*
  Domain: $x \in Expressible\_value = Denotable\_value$
- *Environment*
  Domain: $e \in Environment = Id \rightarrow Denotable\_value$
  Operations:
  $accessenv : Id \rightarrow Environment \rightarrow Denotable\_value$
  $accessenv = \lambda i.\lambda e.\ e(i)$
  $updateenv : Id \rightarrow Denotable\_value \rightarrow Environment \rightarrow Environment$
  $updateenv = \lambda i.\lambda d.\lambda e.\ [i \mapsto d]e$

Applicative Language: Valuation Functions

Valuation Functions:

- **E**: Expression$\rightarrow$ *Environment* $\rightarrow$ *Expressible_value*

  $$\mathbf{E}[[LET\ I = E_1\ IN\ E_2]] = \\ \lambda e.\ \mathbf{E}[[E_2]](updateenv\ [[I]]\ (\mathbf{E}[[E_1]]e)\ e)$$

  Note: $(let\ x = e_1\ in\ e_2)$ *for* $(\lambda x.\ e_2)e_1$

  - **E** determines the meaning of an expression, a denotable value, with the aid of an environment
  - An atom, list, or even a function can be a legal *answer*
  - The LET expression provides a definition mechanism for augmenting the environment
  - Static scoping is used

Applicative Language: Valuation Functions

Valuation Functions:

- **E**: Expression$\rightarrow$ *Environment* $\rightarrow$ *Expressible_value*

$$\mathbf{E}[[LAMBDA\ (I)\ E]] = \\ \lambda e.\ inFunction(\lambda d.\ \mathbf{E}[[E]](updateenv\ [[I]]\ d\ e))$$

$$\mathbf{E}[[E_1\ E_2]] = \lambda e.\ let\ x = (\mathbf{E}[[E_1]]e)\ in\ cases\ x\ of \\ isFunction(f) \rightarrow f(\mathbf{E}[[E_2]]e) \\ []\ isList(t) \rightarrow inError() \\ []\ isAtom(a) \rightarrow inError() \\ []\ isError() \rightarrow inError()\ end$$

  - Functions are created by the LAMBDA construction
  - A function body is evaluated in the context that is active at the point of function definition, augmented by the binding of an actual parameter to the binding identifier
  - This definition is also statically scoped

Applicative Language: Valuation Functions

Valuation Functions:

- **E**: Expression$\rightarrow$ *Environment* $\rightarrow$ *Expressible_value*

$$
\begin{aligned}
&\mathbf{E}[[E_1\ CONS\ E_2]] = \lambda e.\ let\ x = (\mathbf{E}[[E_2]]e)\ in\ cases\ x\ of\\
&\quad isFunction(f) \rightarrow inError()\\
&\quad []\ isList(t) \rightarrow inList(\mathbf{E}[[E_1]]e\ cons\ t)\\
&\quad []\ isAtom(a) \rightarrow inError()\\
&\quad []\ isError() \rightarrow inError()\ end
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{E}[[HEAD\ E]] = \lambda e.\ let\ x = (\mathbf{E}[[E]]e)\ in\ cases\ x\ of\\
&\quad isFunction(f) \rightarrow inError()\\
&\quad []\ isList(t) \rightarrow (null\ t \rightarrow inError()\ []\ (hd\ t))\\
&\quad []\ isAtom(a) \rightarrow inError()\\
&\quad []\ isError() \rightarrow inError()\ end
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{E}[[TAIL\ E]] = \lambda.\ let\ x = (\mathbf{E}[[E]]e)\ in\ cases\ x\ of\\
&\quad isFunction(f) \rightarrow inError()\\
&\quad []\ isList(t) \rightarrow (null\ t \rightarrow inError()\ []\ inList(tl\ t))\\
&\quad []\ isAtom(a) \rightarrow inError()\\
&\quad []\ isError() \rightarrow inError()\ end
\end{aligned}
$$

$$\mathbf{E}[[NIL]] = \lambda e.\ inList(nil)$$

Applicative Language: Valuation Functions

Valuation Functions:

- **E**: Expression$\rightarrow$ *Environment* $\rightarrow$ *Expressible_value*

  $$\mathbf{E}[[I]] = accessenv\ [[I]]$$

  $$\mathbf{E}[[A]] = \lambda e.\ inAtom(\mathbf{A}[[A]])$$

  $$\mathbf{E}[[(E)]] = \mathbf{E}[[E]]$$

- **A**:Atomic-symbol$\rightarrow$ *Atom*

Applicative Language: Scoping Rule

Static Scoping

- The applicative language uses static scoping; that is, the context of a phrase is determined by its physical position in the program
- Consider (let a_0 and a_1 be atomic symbols):

  $LET\ F = a_0\ IN$
  $\quad LET\ F = LAMBDA\ (Z)\ F\ CONS\ Z\ IN$
  $\quad\quad LET\ Z = a_1\ IN$
  $\quad\quad\quad F(Z\ CONS\ NIL)$

  Note: $(let\ x = e_1\ in\ e_2)\ for\ (\lambda x.\ e_2)e_1$

- The occurrence of the first F in the body of the function bound to the second F refers to the atom a_0 – the function is not recursive. The meaning of the entire expression is the same as

$$(LAMBDA\ (Z)\ a_0\ CONS\ Z)\ (a_1\ CONS\ NIL)$$

which equals

$$(a_0\ CONS\ (a_1\ CONS\ NIL)) = (a_0\ a_1)$$

Applicative Language: Scoping Rule

Dynamic Scoping

- An alternative to static scoping is dynamic scoping, where the context of a phrase is determined by the place(s) in the program where the phrase's value is required
- The most general form of dynamic scoping is macro definition and invocation. A definition $LET\ I = E$ binds identifier I to the text E; E is not assigned a context until its value is needed
- When I's value is required, the context where I appears is used to acquire the text that is bound to it
- I is replaced by the text, and the text is evaluated in the existing context
- The version of dynamic scoping found in LISP limits dynamic scoping just to LAMBDA forms
- The semantics of $[[LAMBDA\ (I)\ E]]$ shows that the construct is evaluated within the context of its application to an argument (and not within the context of its definition)

Applicative Language: Scoping Rules

Semantic Algebras:

- *Function*
  Domain: $f \in Function = Environment \rightarrow$
  $Denotable\_value \rightarrow Denotable\_value$

Valuation Functions:

- $\mathbf{E}$: Expression$\rightarrow$ $Environment \rightarrow Expressible\_value$

$$\mathbf{E}[[LAMBDA\ (I)\ E]] =$$
$$\lambda e.\ inFunction(\lambda e'.\ \lambda d.\ \mathbf{E}[[E]](updateenv\ [[I]]\ d\ e'))$$

$$\mathbf{E}[[E_1\ \ E_2]] = \lambda e.\ let\ x = (\mathbf{E}[[E_1]]e)\ in\ cases\ x\ of$$
$$isFunction(f) \rightarrow (f\ \ e\ (\mathbf{E}[[E_2]]e))$$
$$[]\ isList(t) \rightarrow inError()$$
$$[]\ isAtom(a) \rightarrow inError()$$
$$[]\ isError() \rightarrow inError()\ end$$

Principles of Programming Languages: Summary

- λ-Calculus: Syntax
- λ-Calculus: Semantics
- λ-Calculus: Typed
- Programming Languages with λ
  - Functional: Haskell,Scheme, Lisp, ML
  - Multi-Paradigm: λ in C++
- Type Systems
- Denotational Semantics
  - Definition
  - Relationship with Operational and Axiomatic Semantics
  - Semantics of Imperative Languages

Principles of Programming Languages: Modules

[1] **Module 01: Course Information**

a) Why PoPL?
b) Prerequisites
c) Syllabus
   - Module 01
   - Module 02
   - Module 03
   - Module 04
   - Module 05
   - Module 06
   - Module 07
   - Module 08
   - Module 09
d) Course Information
   - Books
   - About the Course
   - Moodle
   - TA & Teacher

Principles of Programming Languages: Modules

[2] **Module 02: λ-Calculus: Syntax**

a) Relations
b) Functions
- Composition
- Currying

c) λ-Calculus
- Concept of λ

d) Syntax
- λ-expressions
- * Notation
- Examples
- * Simple
- * Composition
- * Boolean
- * Numerals
- * Recursion
- * Curried Functions
- * Higher Order Functions

Principles of Programming Languages: Modules

[3] **Module 03: λ-Calculus: Semantics**

a) Semantics

- Free and Bound Variables
- Substitution
- Reduction
- * α-Reduction
- * β-Reduction
- * η-Reduction
- * δ-Reduction
- Order of Evaluation
- * Normal and Applicative Order

[4] **Module 04: Typed λ-Calculus**

a) $\Lambda^{\rightarrow}$

- Type Expression
- Pre-Expression & Expression
- Type-checking Rules
- * Example
- * Practice Problems

b) $\Lambda^{\rightarrow}_{rr}$

- Types
- * Tuple Type
- * Record Type
- * Sum Type
- * Reference Type
- * Array Type
- Type Expression
- Pre-Expression
- Type-checking Rules
- * Derived Rules

[5] **Module 05: λ in C++**

a) Functors
- Callable Entities
- Function Pointers
- * Replace Switch / IF Statements
- * Late Binding
- * Virtual Function
- * Callback
- * Issues
- Basic Functors
- * Elementary Example
- * Examples from STL

b) λ in C++
- λ Expression
- Closure Object
- Examples
- * Factorial
- * Fibonacci
- * Pipeline
- Curry Function

c) More on λ in C++

[6] **Module 06: Denotational Semantics of Imperative Languages**

a) Type Systems
- Type & Type Error
- Type Safety
- Type Checking
- Type Inference

b) Type Inference
- add x = 2 + x
- apply (f, x) = f x
- Inference Algorithm
- * Unification

c) Examples
- sum
- length
- append
- Homework

d) Type Deduction in C++
- Polymorphism
- * Ad-hoc
- * Parametric
- * Subtype
- C++11,...

Principles of Programming Languages: Modules

[7] **Module 07: Denotational Semantics**

a) Semantic Styles

b) Syntax

c) Semantic Domains
- Set, Functions, and Domains
- * Product
- * Sum
- Rat

d) Semantic Algebras
- Nat, Tr
- String
- Unit
- Product Dom
- Sum Dom
- Lists
- Function
- Arrays
- Lifted Domains
- Recursive Fn

e) Denotational Definitions
- Binary
- Calculator

Principles of Programming Languages: Modules

[8] **Module 08: Denotational Semantics of Imperative Languages**

a) Imperative Languages
b) Language + Assignment
c) Programs Are Functions
d) Interactive File Editor
e) Dynamically Typed Language
f) Recursive Definitions
g) Language with Contexts
  - Block Structured Language
  - Applicative Language

[8] **Module 09: λ Calculus – Languages**

a) Overview of Functional Programming
b) Haskell
c) Scheme
d) Lisp