Vrije Universiteit Brussel

Software Languages Lab

Programming Language Engineering
Master of Computer Science

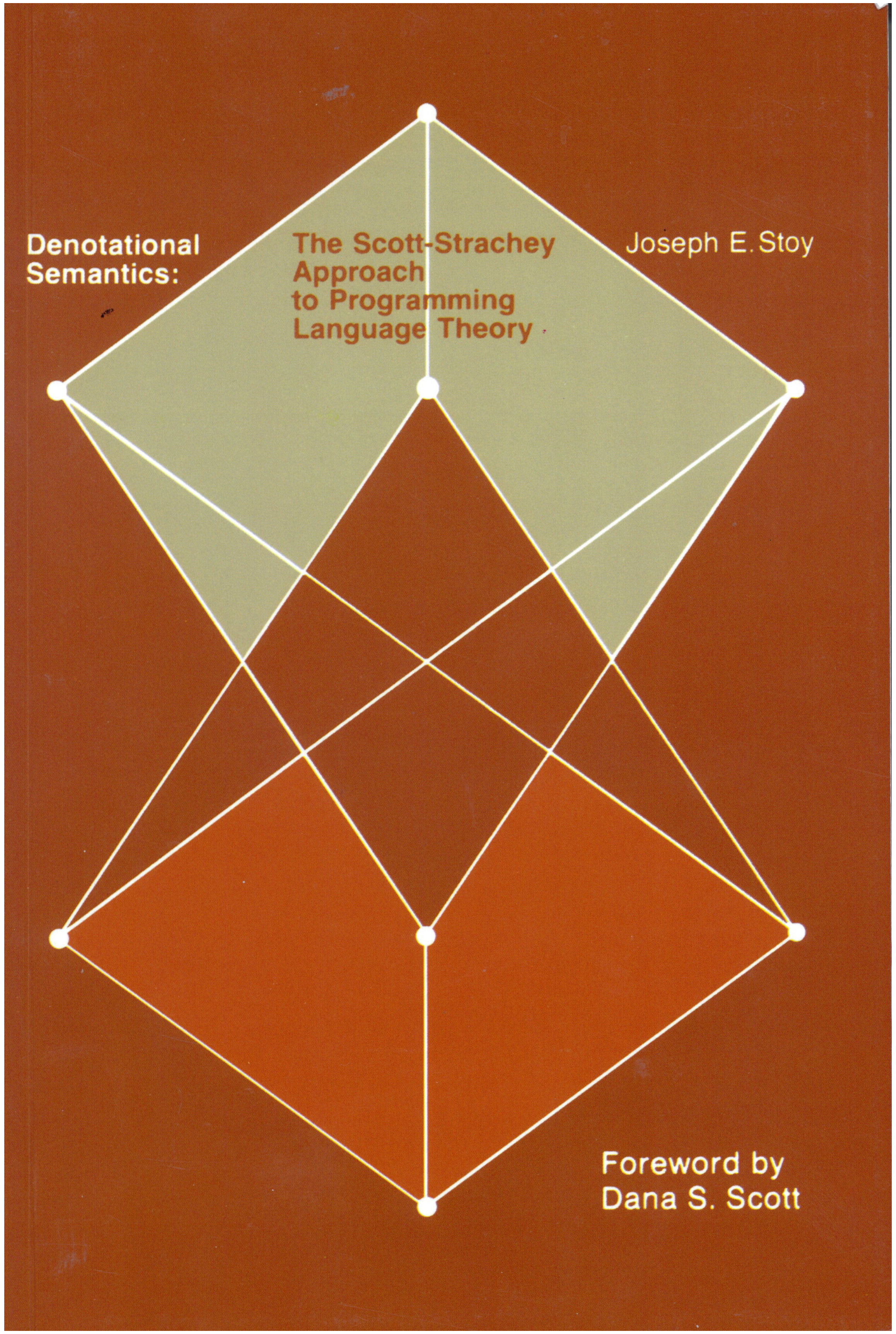
Faculty of Science and Bio-Engineering Sciences
Vrije Universiteit Brussel

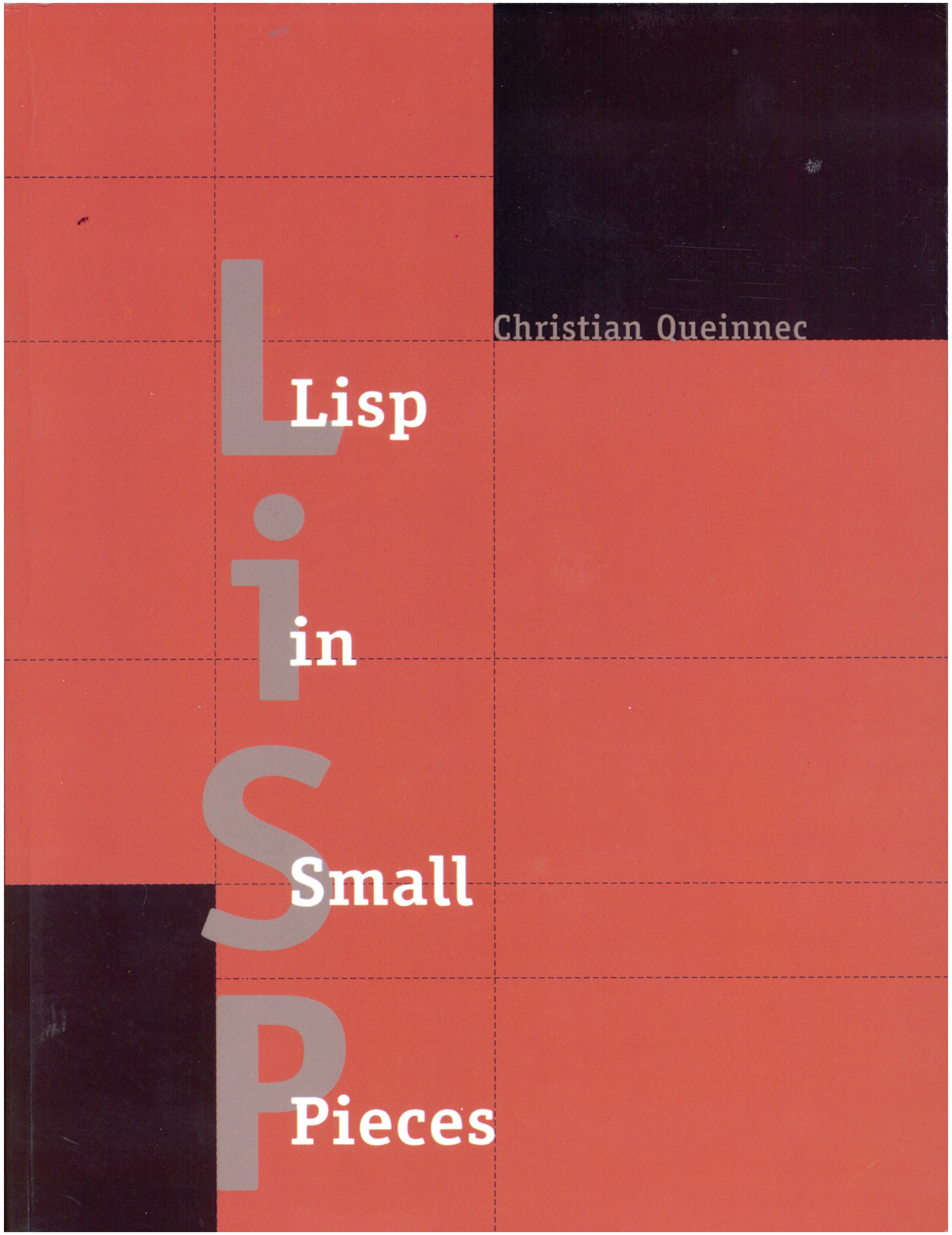
Section 4: Formal Semantics

Theo D'Hondt
Software Languages Lab

"… Strachey: Decide what you want to say before you worry how you are going to say it …"

Sources


Denotational Semantics:
The Scott-Strachey Approach to Programming Language Theory
Joseph E. Stoy
Foreword by Dana S. Scott


Christian Queinnec
Lisp
in
Small
Pieces

The Case of Scheme

Data and procedures and the values they amass,
Higher-order functions to combine and mix and match,
Objects with their local state, the messages they pass,
A property, a package, the control point for a catch—
In the Lambda Order they are all first-class.
One Thing to name them all, One Thing to define them,
One Thing to place them in environments and bind them,
In the Lambda Order they are all first-class.

from AIM-848

Scheme was originally about Actors

This work developed out of an initial attempt to understand the actorness of actors. Steele thought he understood it, but couldn't explain it; Sussman suggested the experimental approach of actually building an "ACTORS interpreter". This interpreter attempted to intermix the use of actors and LISP lambda expressions in a clean manner. When it was completed, we discovered that the "actors" and the lambda expressions were identical in implementation. Once we had discovered this, all the rest fell into place, and it was only natural to begin thinking about actors in terms of lambda calculus.

from AIM-349

A Timeline for Scheme

43pp.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 349 — December 1975

SCHEME

AN INTERPRETER FOR EXTENDED LAMBDA CALCULUS

by

Gerald Jay Sussman and Guy Lewis Steele Jr.

35pp.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 452 — January 1978

THE REVISED REPORT ON

SCHEME

A DIALECT OF LISP

by

Guy Lewis Steele Jr.* and Gerald Jay Sussman

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 848 — August 1985

The Revised Revised Report on Scheme
or
An UnCommon Lisp

by

Hal Abelson	Chris Haynes
Norman Adams	Eugene Kohlbecker
David Bartley	Don Oxley
Gary Brooks	Kent Pitman
William Clinger [editor]	Jonathan Rees
Dan Friedman	Bill Rozas
Robert Halstead	Gerald Jay Sussman
Chris Hanson	Mitchell Wand

76pp.

A Timeline for Scheme

43pp.

Revised^3 Report on the Algorithmic Language Scheme

Jonathan Rees and William Clinger (*Editors*)

H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. J. Sussman, D. H. Bartley, R. Halstead, D. Oxley, M. Wand, G. Brooks, C. Hanson, K. M. Pitman

1986

55pp.

Revised^4 Report on the Algorithmic Language Scheme

William Clinger and Jonathan Rees (*Editors*)

H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, M. Wand

1991

50pp.

Revised^5 Report on the Algorithmic Language Scheme

Richard Kelsey, William Clinger, and Jonathan Rees (*Editors*)

H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, M. Wand

1998

Revised^6 Report on the Algorithmic Language Scheme

Michael Sperber
R. Kent Dybvig, Matthew Flatt, Anton van Straaten
(*Editors*)
Richard Kelsey, William Clinger, Jonathan Rees
(*Editors, Revised^5 Report on the Algorithmic Language Scheme*)
Robert Bruce Findler, Jacob Matthews
(*Authors, formal semantics*)
26 September 2007

2007

90pp.

88pp.

Revised^7 Report on the Algorithmic Language Scheme

Alex Shinn, John Cowan, and Arthur A. Gleckler (*Editors*)

Steven Ganz, Alexey Radul, Olin Shivers, Aaron W. Hsu, Jeffrey T. Read, Alaric Snell-Pym, Bradley Lucier, David Rush, Gerald J. Sussman, Emmanuel Medernach, Benjamin L. Russel

DRAFT 2013

+ IEEE Standard 1178-1990 Standard

Semantics for Scheme

AIM-349	····>	informal lambda-calculus substitution semantics
AIM-452	····>	informal
AIM-848 (RRRS)	····>	… a formal definition of the semantics of Scheme will be included in a separate report …
R3RS	····>	denotational semantics + rewrite rules
R4RS	····>	denotational semantics + rewrite rules + macros (added support for immutables)
R5RS	····>	denotational semantics + syntactic forms
R6RS	····>	operational semantics
R7RS	····>	denotational semantics + syntactic forms (added support for dynamic-wind)

Page 33 from R3RS

$$
\begin{array}{l}
\mid \texttt{(lambda I } \Gamma^* \texttt{ E}_0\texttt{)} \\
\mid \texttt{(if E}_0 \texttt{ E}_1 \texttt{ E}_2\texttt{)} \mid \texttt{(if E}_0 \texttt{ E}_1\texttt{)} \\
\mid \texttt{(set! I E)}
\end{array}
$$

7.2.2. Domain equations

$\alpha \in \mathtt{L}$		locations
$\nu \in \mathtt{N}$		natural numbers
$\mathtt{T}$	$= \{\mathit{false},\ \mathit{true}\}$	booleans
$\mathtt{Q}$		symbols
$\mathtt{H}$		characters
$\mathtt{R}$		numbers
$\mathtt{E_p}$	$= \mathtt{L} \times \mathtt{L}$	pairs
$\mathtt{E_v}$	$= \mathtt{L}^*$	vectors
$\mathtt{E_s}$	$= \mathtt{L}^*$	strings
$\mathtt{M}$	$= \{\mathit{false},\ \mathit{true},\ \mathit{null},\ \mathit{undefined},\ \mathit{unspecified}\}$	miscellaneous
$\phi \in \mathtt{F}$	$= \mathtt{L} \times (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$	procedure values
$\epsilon \in \mathtt{E}$	$= \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E_p} + \mathtt{E_v} + \mathtt{E_s} + \mathtt{M} + \mathtt{F}$	expressed values
$\sigma \in \mathtt{S}$	$= \mathtt{L} \to (\mathtt{E} \times \mathtt{T})$	stores
$\rho \in \mathtt{U}$	$= \mathrm{Ide} \to \mathtt{L}$	environments
$\theta \in \mathtt{C}$	$= \mathtt{S} \to \mathtt{A}$	command continuations
$\kappa \in \mathtt{K}$	$= \mathtt{E}^* \to \mathtt{C}$	expression continuations
$\mathtt{A}$		answers
$\mathtt{X}$		errors

7.2.3. Semantic functions

$$
\begin{array}{r}
\mathcal{K} : \mathrm{Con} \to \mathtt{E} \\
\mathcal{E} : \mathrm{Exp} \to \mathtt{U} \to \mathtt{K} \to \mathtt{C} \\
\mathcal{E}^* : \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{K} \to \mathtt{C} \\
\mathcal{C} : \mathrm{Com}^* \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}
\end{array}
$$

Definition of $\mathcal{K}$ deliberately omitted.

$$\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho\kappa \,.\, \mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa$$

$$
\begin{array}{l}
\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa \,.\, \mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I}) \\
\qquad\qquad (\mathit{single}(\lambda\epsilon \,.\, \epsilon = \mathit{undefined} \to \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“undefined variable”}, \\
\qquad\qquad\qquad \mathit{send}\,\epsilon\,\kappa))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(E}_0\ \texttt{E}^*\texttt{)}]\!] = \\
\quad \lambda\rho\kappa \,.\, \mathcal{E}^*(\mathit{permute}(\langle \mathrm{E}_0\rangle \,\S\, \mathrm{E}^*)) \\
\qquad \rho \\
\qquad (\lambda\epsilon^* \,.\, ((\lambda\epsilon^* \,.\, \mathit{applicate}\,(\epsilon^* \downarrow 1)\,(\epsilon^* \dagger 1)\,\kappa) \\
\qquad\quad (\mathit{unpermute}\,\epsilon^*)))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda (I}^*\texttt{) } \Gamma^* \texttt{ E}_0\texttt{)}]\!] = \\
\quad \lambda\rho\kappa \,.\, \lambda\sigma \,. \\
\qquad \mathit{new}\,\sigma \in \mathtt{L} \to \\
\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma \mid \mathtt{L}, \\
\qquad\qquad \lambda\epsilon^*\kappa' \,.\, \#\epsilon^* = \#\mathrm{I}^* \to \\
\qquad\qquad\quad \mathit{tievals}(\lambda\alpha^* \,.\, (\lambda\rho' \,.\, \mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa')) \\
\qquad\qquad\qquad (\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*)) \\
\qquad\qquad\quad \epsilon^*, \\
\qquad\qquad\quad \mathit{wrong}\ \text{“wrong number of arguments”}\rangle \\
\qquad\quad \text{in } \mathtt{E}) \\
\qquad\quad \kappa \\
\qquad\quad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L})\ \mathit{unspecified}\,\sigma), \\
\qquad \mathit{wrong}\ \text{“out of memory”}\ \sigma
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda (I}^*\texttt{ . I) } \Gamma^* \texttt{ E}_0\texttt{)}]\!] = \\
\quad \lambda\rho\kappa \,.\, \lambda\sigma \,. \\
\qquad \mathit{new}\,\sigma \in \mathtt{L} \to \\
\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma \mid \mathtt{L}, \\
\qquad\qquad \lambda\epsilon^*\kappa' \,.\, \#\epsilon^* \geq \#\mathrm{I}^* \to \\
\qquad\qquad\quad \mathit{tievalsrest} \\
\qquad\qquad\qquad (\lambda\alpha^* \,.\, (\lambda\rho' \,.\, \mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa')) \\
\qquad\qquad\qquad\quad (\mathit{extends}\,\rho\,(\mathrm{I}^* \,\S\, \langle \mathrm{I}'\rangle)\,\alpha^*)) \\
\qquad\qquad\qquad \epsilon^* \\
\qquad\qquad\qquad (\#\mathrm{I}^*), \\
\qquad\qquad\quad \mathit{wrong}\ \text{“too few arguments”}\rangle \text{ in } \mathtt{E}) \\
\qquad\quad \kappa \\
\qquad\quad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L})\ \mathit{unspecified}\,\sigma), \\
\qquad \mathit{wrong}\ \text{“out of memory”}\ \sigma
\end{array}
$$

$$\mathcal{E}[\![\texttt{(lambda I } \Gamma^* \texttt{ E}_0\texttt{)}]\!] = \mathcal{E}[\![\texttt{(lambda (. I) } \Gamma^* \texttt{ E}_0\texttt{)}]\!]$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(if E}_0 \texttt{ E}_1 \texttt{ E}_2\texttt{)}]\!] = \\
\quad \lambda\rho\kappa \,.\, \mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon \,.\, \mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa, \\
\qquad\qquad\qquad\qquad\qquad \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(if E}_0 \texttt{ E}_1\texttt{)}]\!] = \\
\quad \lambda\rho\kappa \,.\, \mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon \,.\, \mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa, \\
\qquad\qquad\qquad\qquad\qquad \mathit{send}\ \mathit{unspecified}\,\kappa))
\end{array}
$$

Here and elsewhere, any expressed value other than *undefined* may be used in place of *unspecified*.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(set! I E)}]\!] = \\
\quad \lambda\rho\kappa \,.\, \mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\mathit{single}(\lambda\epsilon \,.\, \mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I}) \\
\qquad\qquad\qquad\qquad\qquad \epsilon \\
\qquad\qquad\qquad\qquad\qquad (\mathit{send}\ \mathit{unspecified}\,\kappa)))
\end{array}
$$

$$\mathcal{E}^*[\![\,]\!] = \lambda\rho\kappa \,.\, \kappa\langle\,\rangle$$

$$
\begin{array}{l}
\mathcal{E}^*[\![\mathrm{E}_0\ \mathrm{E}^*]\!] = \\
\quad \lambda\rho\kappa \,.\, \mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}(\lambda\epsilon_0 \,.\, \mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\,(\lambda\epsilon^* \,.\, \kappa\,(\langle\epsilon_0\rangle \,\S\, \epsilon^*))))
\end{array}
$$

$$\mathcal{C}[\![\,]\!] = \lambda\rho\theta \,.\, \theta$$

$$\mathcal{C}[\![\Gamma_0\ \Gamma^*]\!] = \lambda\rho\theta \,.\, \mathcal{E}[\![\Gamma_0]\!]\,\rho\,(\lambda\epsilon^* \,.\, \mathcal{C}[\![\Gamma^*]\!]\rho\theta)$$

7.2.4. Auxiliary functions

$$
\begin{array}{l}
\mathit{lookup} : \mathtt{U} \to \mathrm{Ide} \to \mathtt{L} \\
\mathit{lookup} = \lambda\rho\mathrm{I} \,.\, \rho\mathrm{I}
\end{array}
$$

$$
\begin{array}{l}
\mathit{extends} : \mathtt{U} \to \mathrm{Ide}^* \to \mathtt{L}^* \to \mathtt{U} \\
\mathit{extends} = \\
\quad \lambda\rho\mathrm{I}^*\alpha^* \,.\, \#\mathrm{I}^* = 0 \to \rho, \\
\qquad\qquad \mathit{extends}\,(\rho[(\alpha^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)])\,(\mathrm{I}^* \dagger 1)\,(\alpha^* \dagger 1)
\end{array}
$$

$$\mathit{wrong} : \mathtt{X} \to \mathtt{C} \qquad \text{[implementation-dependent]}$$

$$
\begin{array}{l}
\mathit{send} : \mathtt{E} \to \mathtt{K} \to \mathtt{C} \\
\mathit{send} = \lambda\epsilon\kappa \,.\, \kappa\langle\epsilon\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{single} : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K} \\
\mathit{single} = \\
\quad \lambda\psi\epsilon^* \,.\, \#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1), \\
\qquad\qquad \mathit{wrong}\ \text{“wrong number of return values”}
\end{array}
$$

Example: Argument Evaluation

(4) The argument forms (and procedure form) may in principle be evaluated in any order. This is unlike the usual LISP left-to-right order. (All SCHEME interpreters implemented so far have in fact performed left-to-right evaluation, but we do not wish programs to depend on this fact. Indeed, there are some reasons why a clever interpreter might want to evaluate them right-to-left, e.g. to get things on a stack in the correct order.)

AIM-452

A list whose first element is not the keyword of a special form indicates a procedure call. The operator and operand expressions are evaluated and the resulting procedure is passed the resulting arguments. In contrast to other dialects of Lisp the order of evaluation is not specified, and the operator expression and the operand expressions are always evaluated with the same evaluation rules.

AIM-848

R3RS →R7RS \ R6RS

The order of evaluation within a call is unspecified. We mimic that here by applying arbitrary permutations permute and unpermute, which must be inverses, to the arguments in a call before and after they are evaluated. This {still requires | is not quite right since it suggests, incorrectly, } that the order of evaluation is constant throughout a program (for any given number of arguments), but it is a closer approximation to the intended semantics than a left-to-right evaluation would be.

$$
\begin{aligned}
&\mathcal{E}[\![(E_0\ E^*)]\!] = \\
&\quad \lambda\rho\kappa\ .\ \mathcal{E}^*(\text{permute}\ (\langle E_0\rangle\ \S\ E^*)) \\
&\qquad\qquad \rho \\
&\qquad\qquad (\lambda\epsilon^*\ .\ ((\lambda\epsilon^*\ .\ \text{applicate}\ (\epsilon^* \downarrow 1)\ (\epsilon^* \dagger 1)\ \kappa) \\
&\qquad\qquad\qquad (\text{unpermute}\ \epsilon^*)))
\end{aligned}
$$

Some SICP statistics

- the let* form appears exactly twice - in exercise 4.7;
- the letrec form appears exactly twice - in exercise 4.20/21;
- the pattern "named let" does not occur at all;
- the define form is used significantly more frequently than the let form - excluding chapter 1 (where the ratio is 129 to 7), the ratio varies between 3 and 7 to 1;
- most define forms are used on a global level - however, depending on the chapter, up to 1 out of 3 define forms are used locally;
- the do special form is not used; neither is the case form;
- it would really take a lot more space to report on of how little (or not at all) standard library functions are used;
- streams are extensively used, but are absent from any of the Scheme standards.

A Rationale for Slip

- fully SICP compliant;
- first-class treatment of the REPL;
- left-to-right evaluation everywhere[1];
- only one let-form (identical to Scheme's let*);
- first-class define-form, both global and local, usable everywhere[2];
- (almost) no forward referencing;
- set! and define return a value;
- immutables are cloned, not flagged;
- no do- or case-form; no syntax forms
- reduced R5RS primitives
- streams;

[1]not strictly necessary to have a begin-form
[2]if clauses are thunks

Previously ...

An Executable Denotational Semantics for Scheme

Anton van Straaten © anton@appsolutions.com AppSolutions

Abstract.
An executable implementation of the denotational semantics for the Scheme language, as defined in R5RS.
The core of this implementation consists of a faithful translation of the R5RS denotational semantics into the Scheme language.
A denotational semantics (DS) definition of a language can be used for a variety of purposes, including analysis, verification, compilation, and interpretation.
This implementation provides a Scheme interpreter which has been built around the core DS definitions. Other applications of this DS implementation are also possible.

At the time of development, I could not find any other publicly available implementations of the R5RS DS. This kind of work has certainly been done before - Will Clinger has mentioned on comp.lang.scheme that Jonathan Rees did such a translation as part of the development of R3RS, and that:

"Since then, several people have back-translated the denotational semantics into Scheme, usually to test some proposed change to the language or to aid in a mechanical proof of some property of Scheme or an implementation of Scheme. For example, the semantics of multiple values in R5RS was tested that way."
However, none of these translations seems to be publicly available, although my checking of this was limited primarily to searching the WWW and Usenet. At a seminar, I did ask Dr. Clinger about the code he had mentioned, but he told me that he no longer had it, and that he would have made it available otherwise.

"*Denotational semantics is just a very peculiar language with poor syntax.*"
-- Matthias Blume

Notation

$\langle \ldots \rangle$	sequence formation
$s \downarrow k$	kth member of the sequence s (1-based)
$\#s$	length of sequence s
$s \S t$	concatenation of sequences s and t
$s \dagger k$	drop the first k members of sequence s
$t \rightarrow a, b$	McCarthy conditional "if t then a else b"
$\rho[x / i]$	substitution "ρ with x for i"
x in D	injection of x into domain D
$x \mid D$	projection of x to domain D

The Environment/Store model

$\text{initial-store} = \lambda\alpha.\alpha = \emptyset \rightarrow 0$

```
(define null-address 0)

(define (initial-store address)
  (if (= address null-address) 0))
```

$\text{initial-environment} = \lambda I.\varnothing$

```
(define unspecified-value unspecified-value)

(define (initial-environment variable)
  unspecified-value)
```

The Store model

$locate = \lambda\alpha\sigma.\sigma\ \alpha$

```
(define (locate address store)
  (store address))
```

$new = \lambda\sigma.(locate\ \emptyset\ \sigma) + 1$

```
(define (new store)
  (+ (locate null-address store) 1))
```

$bind\text{-}address = \sigma[\varepsilon/\alpha] = \lambda\alpha\varepsilon\sigma.\lambda\alpha'.\alpha' = \alpha \rightarrow \varepsilon\ ,\ \sigma\ \alpha'$

```
(define (bind-address address value store)
  (lambda (address-prime)
    (if (= address-prime address)
      value
      (store address-prime))))
```

$extend = \lambda\sigma\alpha.\sigma[\alpha/\emptyset]$

```
(define (extend address store)
  (bind-address null-address address store))
```

The Environment model

$$\text{lookup} = \lambda I\rho.\rho\ I$$

```
(define (lookup variable environment)
  (environment variable))
```

$$\text{bind-variable} = \rho[\alpha/I] = \lambda I\alpha\rho.\lambda I'.I = I' \rightarrow \alpha\ ,\ \rho\ I'$$

```
(define (bind-variable variable address environment)
  (lambda (variable-prime)
    (if (equal? variable-prime variable)
      address
      (environment variable-prime))))
```

The REP loop

```
(define (expression-evaluator expression continuation
                                         environment store recovery)
  (if (pair? expression)
   (begin
     (define operator (car expression))
     (define operands (cdr expression))
     (if (equal? operator 'begin)
       (parse-begin operands continuation)
        ... )
    (if (symbol? expression)
      (variable-evaluator expression continuation
                                     environment store recovery)
      (constant-evaluator expression continuation
                                     environment store recovery))))
```

```
(define (continuation-loop value environment store)
  (display value)
  (newline)
  (display ">>>")
  (define expression (read))
  (expression-evaluator expression continuation-loop
                                      environment store environment))
```

The REP loop - curry'ed

```
(define (expression-evaluator expression continuation)
  (if (pair? expression)
    (begin
      (define operator (car expression))
      (define operands (cdr expression))
      (if (equal? operator 'begin)
        (parse-begin operands continuation)
           ... )
    (if (symbol? expression)
      (variable-evaluator expression continuation)
      (constant-evaluator expression continuation))))
```

```
(define (continuation-loop value)
  (lambda (environment)
    (lambda (store)
      (display value)
      (newline)
      (display ">>>")
      (define expression (read))
      ((((expression-evaluator expression continuation-loop)
                                      environment) store) environment))))
```

Error recovery

```
(define (wrong message)
  (lambda (environment)
    (lambda (store)
      (lambda (recovery)
        (define value (string-append "error: " message))
        (((continuation-loop value) recovery) store)))))
```

Def/Get/Set in an Environment/Store

$$
\begin{aligned}
def = \lambda I\varepsilon\kappa.\lambda\rho.\lambda\sigma.(\lambda\alpha.(\lambda\sigma'.(\lambda\rho'.((send\ \varepsilon\ \kappa)\rho')\sigma')\\
\rho[\alpha/I])\\
(\lambda\sigma''.\sigma''[\varepsilon/\alpha])(extend\ \alpha\ \sigma))\\
(new\ \sigma)
\end{aligned}
$$

```
(define (def variable value continuation)
  (lambda (environment)
    (lambda (store)
      (define address (new store))
      (define updated-environment
                     (bind-variable variable address environment))
      (define extended-store (extend address store))
      (define updated-store (bind-address address value extended-store))
      (((send value continuation) updated-environment) updated-store))))
```

Def/get/set in an Environment (cont'd)

set = λIεκ.λρ.λσ.(λα.α = ∅ →
((wrong "unknown variable")ρ)σ ,
((send ε κ)ρ)σ[ε/α])
(lookup I ρ)

```
(define (set variable value continuation)
  (lambda (environment)
    (lambda (store)
      (define address (lookup variable environment))
      (if (equal? address unspecified-value)
        (((wrong "unknown variable") environment) store)
        (begin
          (define updated-store (bind-address address value store))
          (((send value continuation) environment) updated-store))))))
```

Def/get/set in an Environment (cont'd)

get = λIκ.λρ.λσ.(λα.α = ∅ →
((wrong “unknown variable”)ρ)σ ,
((send (locate α σ) κ)ρ)σ)
(lookup I ρ)

```
(define (get variable continuation)
  (lambda (environment)
    (lambda (store)
      (define address (lookup variable environment))
      (if (equal? address unspecified-value)
        (((wrong "unknown variable") environment) store)
        (begin
          (define value (locate address store))
          (((send value continuation) environment) store))))))
```

The Evaluation Dispatcher

```
(define (expression-evaluator expression continuation)
  (if (pair? expression)
    (begin
      (define operator (car expression))
      (define operands (cdr expression))
      (if (equal? operator 'begin)
        (parse-begin operands continuation)
        (if (equal? operator 'define)
          (parse-define operands continuation)
          (if (equal? operator 'if)
            (parse-if operands continuation)
            (if (equal? operator 'lambda)
              (parse-lambda operands continuation)
              (if (equal? operator 'quote)
                (parse-quote operands continuation)
                (if (equal? operator 'set!)
                  (parse-set! operands continuation)
                  (application-evaluator operator operands
                                                              continuation))))))))
    (if (symbol? expression)
      (variable-evaluator expression continuation)
      (constant-evaluator expression continuation))))
```

Slip Abstract Syntax

$K \in Con$ constants
$I \in Ide$ variables
$\varepsilon \in E$ expressions

$\varepsilon \rightarrow K \mid I \mid (\varepsilon_0\ \varepsilon^*)$
$\mid$ (begin ε^+)
$\mid$ (define I ε)
$\mid$ (define (I_0 I^*) ε^+)
$\mid$ (define (I_0 I^+ . I) ε^+)
$\mid$ (define (I_0 . I) ε^+)
$\mid$ (if ε_0 ε_1) $\mid$ (if ε_0 ε_1 ε_2)
$\mid$ (lambda (I^*) ε^+)
$\mid$ (lambda (I^+ . I) ε^+)
$\mid$ (lambda I ε^+)
$\mid$ (quote ε)
$\mid$ (set! I ε)

$\mid$ (set! (I_0 I^*) ε^+) ?
$\mid$ (set! (I_0 I^+ . I) ε^+)
$\mid$ (set! (I_0 . I) ε^+)

Slip Domain Equations

$\alpha \in L$ locations

$\nu \in N$ natural numbers

$T = \{$ false, true $\}$ booleans

Q symbols

H characters

R numbers

$Ep = L \times L$ pairs

$Ev = L^*$ vectors

$Es = H^*$ strings

$M = \{$ false, true, null, unspecified $\}$ miscellaneous

$\varphi \in F = L \times (E^* \times K \rightarrow U \rightarrow S \rightarrow U)$ procedure values

$\varepsilon \in E = Q + H + R + Ep + Ev + Es + M + F$ expressed values

$\sigma \in S \subset L \times E$ stores

$\rho \in U \subset Ide \times L$ environments

$\kappa \in K = E \rightarrow U \rightarrow S \rightarrow U$ continuations

Semantic Functions

$$\mathcal{K} : \mathrm{Con} \rightarrow \mathrm{E}$$

$$\mathcal{E} : \mathrm{E} \rightarrow \mathrm{K} \rightarrow \mathrm{U} \rightarrow \mathrm{S} \rightarrow \mathrm{U}$$

$$\mathcal{E}^* : \mathrm{E}^* \rightarrow \mathrm{K} \rightarrow \mathrm{E}^* \rightarrow \mathrm{U} \rightarrow \mathrm{S} \rightarrow \mathrm{U}$$

$$\mathcal{E}^+ : \mathrm{E}^* \rightarrow \mathrm{K} \rightarrow \mathrm{U} \rightarrow \mathrm{S} \rightarrow \mathrm{U}$$

A Simple Semantic/Evaluation Function

$$\mathcal{E}[\![(\text{set! I E})]\!] = \lambda\kappa.\mathcal{E}[\![\text{E}]\!]\ (\lambda\varepsilon.\text{set I}\ \varepsilon\ \kappa)$$

```
(define (set!-evaluator variable expression continuation)
  (define (continuation-value value)
    (set variable value continuation))
  (expression-evaluator expression continuation-value))
```

Sequences and Lists

$$\mathcal{E}^+[\![E^+]\!] = \lambda\kappa.\mathcal{E}[\![E^+ \downarrow 1]\!]\ (\lambda\varepsilon.\#E^+ = 1 \rightarrow (\text{send}\ \varepsilon\ \kappa)\ ,\ \mathcal{E}^+[\![E^+ \dagger 1]\!]\ \kappa)$$

```
(define (sequence-evaluator expressions continuation)
  (define (continuation-value value)
    (define rest-of-expressions (cdr expressions))
    (if (null? rest-of-expressions)
      (send value continuation)
      (sequence-evaluator rest-of-expressions continuation)))
  (define expression (car expressions))
  (expression-evaluator expression continuation-value))
```

$$\mathcal{E}^*[\![E^*]\!] = \lambda\kappa\ell.\#E^* = 0 \rightarrow \text{send}\ \ell\ \kappa\ ,$$

$$\mathcal{E}[\![E^* \downarrow 1]\!]\ (\lambda\varepsilon.(\mathcal{E}^*[\![E^* \dagger 1]\!]\ \kappa\ (\ell\ \S\ \langle\varepsilon\rangle))$$

```
(define (list-evaluator expressions continuation list)
  (define (continuation-value value)
    (define extended-list (cons value list))
    (define rest-of-expressions (cdr expressions))
      (list-evaluator rest-of-expressions continuation extended-list))
  (if (null? expressions)
    (send (reverse list) continuation)
    (begin
      (define expression (car expressions))
      (expression-evaluator expression continuation-value))))
```

Immutables

$$\mathcal{E}[\![K]\!] = \lambda\kappa.\text{send}\ (\mathcal{K}[\![K]\!])\ \kappa$$

```
define (constant-evaluator datum continuation)
    (define constant-expression (constant datum))
    (send constant-expression continuation))
```

```
(define (constant datum)
  (if (pair? datum)
    (cons (constant (car datum)) (constant (cdr datum)))
    (if (string? datum)
      (string-append datum "")
      (if (vector? datum)
        (list->vector (constant (vector->list datum)))
        datum))))
```

If with Thunks

$$\mathcal{E}[\![(\text{if } E_0\ E_1\ E_2)]\!] = \lambda\kappa.\mathcal{E}[\![E_0]\!]\ (\lambda\varepsilon.\lambda\rho.((\varepsilon = \text{false} \rightarrow \mathcal{E}[\![E_2]\!]\ ,\ \mathcal{E}[\![E_1]\!])\ (\lambda\varepsilon.\lambda\rho_{\varnothing}.(\text{send } \varepsilon\ \kappa)\rho))\rho)$$

```
(define (if-then-else-evaluator predicate consequent alternate continuation)
  (define (continuation-predicate boolean)
    (lambda (environment-predicate)
      (define (continuation-clause value)
        (lambda (ignore-environment)
          ((send value continuation) environment-predicate)))
      (define expression (if boolean consequent alternate))
      ((expression-evaluator expression continuation-clause)
                                                 environment-predicate)))
  (expression-evaluator predicate continuation-predicate))
```

Lambda expressions

$$\mathcal{E}[\![(\text{lambda } (\text{I}^*)\ \text{E+})]\!] = \lambda \text{I}^*\text{E+}.\lambda\kappa.\lambda\rho.$$
$$(\text{send } (\lambda\varepsilon^*\kappa\text{call}.\lambda\rho\text{call}.\#\text{I}^* = \#\varepsilon^* \rightarrow$$
$$(\text{bind I}^*\ \varepsilon^*\ (\lambda\varepsilon_\varnothing.\text{body E+ }\kappa\text{call }\rho\text{call}))\rho\ ,$$
$$\text{wrong “non-matching argument list”})\ \kappa)\rho$$

```
(define (lambda-fixed-arity-function-evaluator parameters expressions
                                                             continuation)
  (lambda (environment)
    (define (procedure arguments continuation-call)
      (lambda (environment-call)
        (define (continuation-body ignore-value)
          (body expressions continuation-call environment-call))
        (if (= (length parameters) (length arguments))
          ((bind parameters arguments continuation-body) environment)
          (wrong "non-matching argument list"))))
    ((send procedure continuation) environment)))
```

First-class define

$$\mathcal{E}[\![(\text{define I E})]\!] = \lambda\kappa.\text{def I } \varnothing\ (\lambda\varepsilon_{\varnothing}.\mathcal{E}[\![\text{E}]\!]\ (\lambda\varepsilon.\text{set I } \varepsilon\ \kappa))$$

```
(define (define-variable-evaluator variable expression continuation)
  (define (continuation-define ignore-value)
    (define (continuation-value value)
      (set variable value continuation))
    (expression-evaluator expression continuation-value))
  (def variable unspecified-value continuation-define))
```

$$\mathcal{E}[\![(\text{define } (\text{I}_0\ \text{I}^*)\ \text{E+})]\!] = \lambda\kappa.\text{def I}_0\ \varnothing\ (\lambda\varepsilon_{\varnothing}.\mathcal{E}[\![(\text{lambda } (\text{I}^*)\ \text{E+})]\!]\ (\lambda\varphi.\text{set I}_0\ \varphi\ \kappa))$$

```
(define (define-fixed-arity-function-evaluator variable parameters
                                               expressions continuation)
  (define (continuation-define ignore-value)
    (define (continuation-closure procedure)
      (set variable procedure continuation))
    (lambda-fixed-arity-function-evaluator parameters expressions
                                                  continuation-closure))
  (def variable unspecified-value continuation-define))
```

Function Application

$$applicate = \lambda \varphi \varepsilon^* \kappa . \varphi\ \varepsilon^*\ \kappa$$

```
(define (applicate procedure arguments continuation)
  (procedure arguments continuation))
```

$$bind = \lambda I^* \varepsilon^* \kappa . I^* = \langle\rangle \rightarrow send\ \varepsilon^*\ \kappa\ ,$$
$$def\ I^*{\downarrow}1\ \varepsilon^*{\downarrow}1\ (\lambda \varepsilon_{\varnothing} . bind\ I^*{\dagger}1\ \varepsilon^*{\dagger}1\ \kappa)$$

```
(define (bind parameters arguments continuation)
  (define (continuation-bind ignore-value)
    (bind (cdr parameters) (cdr arguments) continuation))
  (if (null? parameters)
    (send arguments continuation)
    (def (car parameters) (car arguments) continuation-bind)))
```

$$body = \lambda E^+ \kappa . \lambda \rho . \mathcal{E}^+ [\![E^+]\!]\ (\lambda \varepsilon . \lambda \rho_{\varnothing} . (send\ \varepsilon\ \kappa) \rho)$$

```
(define (body expressions continuation environment)
  (define (continuation-closure value)
    (lambda (ignore-environment)
      ((send value continuation) environment)))
  (sequence-evaluator expressions continuation-closure))
```

Evaluation Startup

```
(define (wrap primitive)
  (lambda (arguments continuation)
    (define value (apply primitive arguments))
    (send value continuation)))
```

```
(define primitives (list (cons 'circularity-level 0)
                         (cons 'false #f)
                         (cons 'true #t)
                         (cons '- (wrap -))
                         (cons '* (wrap *))
                         ...
```

```
(define (initialize primitives)
  (if (null? primitives)
    (continuation-loop "Denotational Semantics Slip")
    (begin
      (define (continuation-define ignore-value)
        (initialize (cdr primitives)))
      (define variable (caar primitives))
      (define value (cdar primitives))
      (def variable value continuation-define))))

(((initialize primitives) initial-environment) initial-store))
```

Higher-Order Primitive Functions

```
(define (ds-apply arguments continuation) ...

(define (ds-call-with-current-continuation arguments continuation)
  (lambda (environment)
    (define (current-continuation arguments ignore-continuation)
      (define (parse-value value residue)
        (if (null? residue)
          ((send value continuation) environment)
          (wrong "one argument required in current continuation")))
      (parse-pair arguments parse-value))
    (define (parse-procedure procedure residue)
      (if (null? residue)
        ((applicate procedure (list current-continuation) continuation)
                                                               environment)
        (wrong "one argument required in call-with-current-continuation")))
    (parse-pair arguments parse-procedure)))

(define (ds-eval arguments continuation) ...
(define (ds-for-each arguments continuation) ...
(define (ds-force arguments continuation) ...
(define (ds-load arguments continuation) ...
(define (ds-map arguments continuation) ...
```

An (Executable) DS for Slip #1

$\mathcal{E}[\![(E_0\ E^*)]\!] = \lambda\kappa.\mathcal{E}[\![E_0]\!]\ (\lambda\varepsilon.\varepsilon \in F \rightarrow\ \mathcal{E}^*[\![E^*]\!]\ (\lambda\varepsilon^*.\text{applicate}\ \varepsilon\ \varepsilon^*\ \kappa)\ \langle\rangle\ ,$
$\qquad\qquad \text{wrong “procedure expected”})$

$\mathcal{E}[\![(\text{begin}\ E^+)]\!] = \lambda\kappa.\text{send}\ \mathcal{E}^+[\![\ E^+]\!]\ \kappa$

$\mathcal{E}[\![K]\!] = \text{send}\ \mathcal{K}[\![K]\!]\ \kappa$

$\mathcal{E}[\![(\text{define}\ I\ E)]\!] = \lambda\kappa.\text{def}\ I\ \varnothing\ (\lambda\varepsilon_\varnothing.\mathcal{E}[\![E]\!]\ (\lambda\varepsilon.\text{set}\ I\ \varepsilon\ \kappa))$

$\mathcal{E}[\![(\text{define}\ (I_0\ I^*)\ E^+)]\!] = \lambda\kappa.\text{def}\ I_0\ \varnothing\ (\lambda\varepsilon_\varnothing.\mathcal{E}[\![(\text{lambda}\ (I^*)\ E^+)]\!]\ (\lambda\varphi.\text{set}\ I_0\ \varphi\ \kappa))$

$\mathcal{E}[\![(\text{define}\ (I_0\ I^+\ .\ I)\ E^+)]\!] = \lambda\kappa.\text{def}\ I_0\ \varnothing\ (\lambda\varepsilon_\varnothing.\mathcal{E}[\![(\text{lambda}\ (I^+\ .\ I)\ E^+)]\!]\ (\lambda\varphi.\text{set}\ I_0\ \varphi\ \kappa))$

$\mathcal{E}[\![(\text{define}\ (I_0\ .\ I)\ E^+)]\!] = \lambda\kappa.\text{def}\ I_0\ \varnothing\ (\lambda\varepsilon_\varnothing.\mathcal{E}[\![(\text{lambda}\ I\ E^+)]\!]\ (\lambda\varphi.\text{set}\ I_0\ \varphi\ \kappa))$

$\mathcal{E}[\![(\text{if}\ E_0\ E_1)]\!] = \lambda\kappa.\mathcal{E}[\![E_0]\!]\ (\lambda\varepsilon.\lambda\rho.\varepsilon = \text{false} \rightarrow (\text{send}\ \varnothing\ \kappa)\rho\ ,$
$\qquad\qquad (\mathcal{E}[\![E_1]\!]\ (\lambda\varepsilon.\lambda\rho_\varnothing.(\text{send}\ \varepsilon\ \kappa)\rho))\rho)$

$\mathcal{E}[\![(\text{if}\ E_0\ E_1\ E_2)]\!] = \lambda\kappa.\mathcal{E}[\![E_0]\!]\ (\lambda\varepsilon.\lambda\rho.((\varepsilon = \text{false} \rightarrow \mathcal{E}[\![E_2]\!]\ ,$
$\qquad\qquad \mathcal{E}[\![E_1]\!])\ (\lambda\varepsilon.\lambda\rho_\varnothing.(\text{send}\ \varepsilon\ \kappa)\rho))\rho)$

$\mathcal{E}[\![(\text{lambda}\ (I^*)\ E^+)]\!] = \lambda I^*E^+.\lambda\kappa.\lambda\rho.(\text{send}\ (\lambda\varepsilon^*\kappa_{call}.\lambda\rho_{call}.\#I^* = \#\varepsilon^* \rightarrow$
$\qquad\qquad (\text{bind}\ I^*\ \varepsilon^*\ (\lambda\varepsilon_\varnothing.\text{body}\ E^+\ \kappa_{call}\ \rho_{call}))\rho\ ,$
$\qquad\qquad \text{wrong “non-matching argument list”})\ \kappa)\rho$

$\mathcal{E}[\![(\text{lambda}\ (I^+\ .\ I)\ E^+)]\!] = \lambda I^*IE^+.\lambda\kappa.\lambda\rho.(\text{send}\ (\lambda\varepsilon^*\kappa_{call}.\lambda\rho_{call}.\#I^* <= \#\varepsilon^* \rightarrow$
$\qquad\qquad (\text{bind}\ I^*\ \varepsilon^*\ (\lambda\varepsilon^{*\prime}.\text{def}\ I\ \varepsilon^{*\prime}\ (\lambda\varepsilon_\varnothing.\text{body}\ E^+\ \kappa_{call}\ \rho_{call})))\rho\ ,$
$\qquad\qquad \text{wrong “non-matching argument list”})\ \kappa)\rho$

$\mathcal{E}[\![(\text{lambda}\ I\ E^+)]\!] = \lambda IE^+\kappa.\lambda\rho.(\text{send}\ (\lambda\varepsilon^*\kappa_{call}.\lambda\rho_{call}.\text{def}\ I\ \varepsilon^*\ (\lambda\varepsilon_\varnothing.(\text{body}\ E^+\ \kappa_{call})\rho_{call})\rho)\ \kappa)\rho$

$\mathcal{E}[\![(\text{quote}\ E)]\!] = \lambda\kappa.\text{send}\ \mathcal{K}[\![E]\!]\ \kappa$

$\mathcal{E}[\![(\text{set!}\ I\ E)]\!] = \lambda\kappa.\mathcal{E}[\![E]\!]\ (\lambda\varepsilon.\text{set}\ I\ \varepsilon\ \kappa)$

$\mathcal{E}[\![I]\!] = \lambda\kappa.\text{get}\ I\ \kappa$

An (Executable) DS for Slip #2

$\mathcal{E}^+[\![E^+]\!] = \lambda\kappa.\mathcal{E}[\![E^+\downarrow 1]\!]\ (\lambda\varepsilon.\#E^+ = 1 \rightarrow (\text{send}\ \varepsilon\ \kappa)\ ,\ \mathcal{E}^+[\![E^+\dagger 1]\!]\ \kappa)$
$\mathcal{E}^*[\![E^*]\!] = \lambda\kappa l.\#E^* = 0 \rightarrow \text{send}\ l\ \kappa\ ,\ \ \mathcal{E}[\![E^* \downarrow 1]\!]\ (\lambda\varepsilon.\mathcal{E}^*[\![E^*\dagger 1]\!]\ \kappa\ (l\ \S\ \langle\varepsilon\rangle))$

bind-address = $\sigma[\varepsilon/\alpha] = \lambda\alpha\varepsilon\sigma.\lambda\alpha'.\alpha' = \alpha \rightarrow \varepsilon\ ,\ \ \sigma\ \alpha'$
bind-variable = $\rho[\alpha/I] = \lambda I\alpha\rho.\lambda I'.I = I' \rightarrow \alpha\ ,\ \rho\ I'$
extend = $\lambda\sigma\alpha.\sigma[\alpha/0]$
locate = $\lambda\alpha\sigma.\sigma\ \alpha$
lookup = $\lambda I\rho.\rho\ I$
new = $\lambda\sigma.(\text{locate}\ 0\ \sigma) + 1$

def = $\lambda I\varepsilon\kappa.\lambda\rho.\lambda\sigma.(\lambda\alpha.(\lambda\sigma'.(\lambda\rho'.((\text{send}\ \varepsilon\ \kappa)\rho')\sigma')\rho[\alpha/I])(\lambda\sigma''.\sigma''[\varepsilon/\alpha])(\text{extend}\ \alpha\ \sigma))(\text{new}\ \sigma)$
set = $\lambda I\varepsilon\kappa.\lambda\rho.\lambda\sigma.(\lambda\alpha.\alpha = \varnothing \rightarrow ((\text{wrong “unknown variable”}\ I)\rho)\sigma\ ,$
$((\text{send}\ \varepsilon\ \kappa)\rho)\sigma[\varepsilon/\alpha])(\text{lookup}\ I\ \rho)$
get = $\lambda I\kappa.\lambda\rho.\lambda\sigma.(\lambda\alpha.\alpha = \varnothing \rightarrow ((\text{wrong “unknown variable”}\ I)\rho)\sigma\ ,$
$((\text{send}\ (\text{locate}\ \alpha\ \sigma)\ \kappa)\rho)\sigma)(\text{lookup}\ I\ \rho)$

applicate = $\lambda\varphi\varepsilon^*\kappa.\varphi \in F \rightarrow \varphi\ \varepsilon^*\ \kappa\ ,\ \text{wrong “procedure expected”}$
bind = $\lambda I^*\varepsilon^*\kappa.I^* = \langle\rangle \rightarrow \text{send}\ \varepsilon^*\ \kappa\ ,\ \text{def}\ I^*\downarrow 1\ \varepsilon^*\downarrow 1\ (\lambda\varepsilon_\varnothing.\text{bind}\ I^*\dagger 1\ \varepsilon^*\dagger 1\ \kappa)$
body = $\lambda E^+\kappa.\lambda\rho.\mathcal{E}^+[\![E^+]\!]\ (\lambda\varepsilon.\lambda\rho_\varnothing.(\text{send}\ \varepsilon\ \kappa)\rho)$

ds.slip

```
Start Slip from /Users/tjdhondt/Desktop/SLIPcharbased/testcode
>(define (f n) (if (> n 1) (* n (f (- n 1))) 1))

>(f 10)
3628800
>(load "ds")
Denotational Semantics Slip
>>>>(define (f n) (if (> n 1) (* n (f (- n 1))) 1))
#<procedure procedure>
>>>>(f 10)
3628800
>>>>(load "ds")
Denotational Semantics Slip
>>>>(define (f n) (if (> n 1) (* n (f (- n 1))) 1))
#<procedure procedure>
>>>>(f 10)
LOG   0: stop the world
LOG   1: stop the world
LOG   2: stop the world
3628800
>>>>
```

ds.rkt

```
Welcome to DrRacket, version 5.3 [3m].
Language: R5RS; memory limit: 128 MB.
> (define (f n) (if (> n 1) (* n (f (- n 1))) 1))
> (f 10)
3628800
> (load "dsSlip.rkt")
Denotational Semantics Slip
>>> (define (f n) (if (> n 1) (* n (f (- n 1))) 1))
#<procedure:procedure>
>>>(f 10)
3628800
>>>(load "ds.rkt")
Denotational Semantics Slip
>>> (define (f n) (if (> n 1) (* n (f (- n 1))) 1))
#<procedure:procedure>
>>>(f 10)
3628800
>>>
```