In (quasi quotation), a (list template D) can sometimes be confused with either an (unquotation D) or a (splicing unquotation D). The interpretation as an (unquotation) or (splicing unquotation D) takes precedence.

7.1.5. Programs and definitions

\[(\text{program}) \rightarrow (\text{command or definition})*\]
\[(\text{command or definition}) \rightarrow (\text{command}) \mid (\text{definition})\]
\[(\text{definition}) \rightarrow (\text{define} \ (\text{variable}) \ (\text{def forms}) \ (\text{body}))\]
\[(\text{define} \ (\text{variable}) \ (\text{def forms}) \ (\text{body}))\]
\[(\text{def forms}) \rightarrow (\text{variable})*\]
\[(\text{variable})^+ . \ (\text{variable})\]

7.2. Formal semantics

This section provides a formal denotational semantics for the primitive expressions of Scheme and selected built-in procedures. The concepts and notation used here are described in [93]; the notation is summarized below:

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

The reason that expression continuations take sequences of values instead of single values is to simplify the formal treatment of procedure calls and to make it easy to add multiple return values.

The boolean flag associated with pairs, vectors, and strings will be true for mutable objects and false for immutable objects.

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 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.

The storage allocator new is implementation-dependent, but it must obey the following axiom: if new \(\sigma \in L\), then \(\sigma (\text{new} \sigma \mid L) \downarrow 2 = \text{false}\).

The definition of \(K\) is omitted because an accurate definition of \(K\) would complicate the semantics without being very interesting.

If P is a program in which all variables are defined before being referenced or assigned, then the meaning of P is
\[\mathcal{E}[(\text{lambda} (I^*) P') \ (\text{undefined} ...)]\]
where \(I^*\) is the sequence of variables defined in P, \(P'\) is the sequence of expressions obtained by replacing every definition in P by an assignment, \(\text{(undefined)}\) is an expression that evaluates to undefined, and \(\mathcal{E}\) is the semantic function that assigns meaning to expressions.

7.2.1. Abstract syntax

- \(K \in \text{Con}\) constants, including quotations
- \(I \in \text{Id}\) identifiers (variables)
- \(E \in \text{Exp}\) expressions
- \(\Gamma \in \text{Com} = \text{Exp}\) commands

\[
\text{Exp} \rightarrow K \mid I \mid (E_0 E^*)
\]
\[
\mid (\text{lambda} (I^*) I^* E_0)
\]
\[
\mid (\text{lambda} (I^* . I) I^* E_0)
\]
\[
\mid (\text{lambda} I I^* E_0)
\]
\[
\mid (\text{if} E_0 E_1 E_2) \mid (\text{if} E_0 E_1)
\]
\[
\mid (\text{set!} I E)
\]

7.2.2. Domain equations

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

7.2.3. Semantic functions

- \(K : \text{Con} \rightarrow E\)
- \(\mathcal{E} : \text{Exp} \rightarrow U \rightarrow K \rightarrow C\)
- \(E^* : \text{Exp}^* \rightarrow U \rightarrow K \rightarrow C\)
- \(C : \text{Com}^* \rightarrow U \rightarrow C\rightarrow C\)

Definition of \(K\) deliberately omitted.
\[\mathcal{E}[K] = \lambda p : \text{send}(K \llbracket K \rrbracket) \llbracket\]
7. Formal syntax and semantics

2.4. Auxiliary functions

\[ \text{lookup} : U \rightarrow \text{Id}e \rightarrow L \]
\[ \text{lookup} = \lambda \rho. \rho I \]
\[ \text{extends} : U \rightarrow \text{Id}e^* \rightarrow L^* \rightarrow U \]
\[ \text{extends} = \lambda \rho I^* e^* \cdot \lambda I^* \cdot e^* \rightarrow \rho, \]
\[ \text{extends} (\rho((a^* \downarrow 1)/(I^* \downarrow 1))) (I^* \downarrow 1) \]
\[ \text{wrong number of return values} \]
\[ \text{new} : S \rightarrow (L + \{ \text{error} \}) \]
\[ \text{hold} : L \rightarrow K \rightarrow C \]
\[ \text{hold} = \lambda \lambda \sigma \cdot \text{send} (\sigma \alpha \downarrow 1) \]
\[ \text{assign} : L \rightarrow E \rightarrow C \rightarrow C \]
\[ \text{assign} = \lambda \lambda \theta \sigma \cdot \theta (\text{update } \alpha \exp) \]
\[ \text{update} : L \rightarrow E \rightarrow S \rightarrow S \]
\[ \text{update} = \lambda \alpha \exp \cdot \sigma (\exp, \text{true } / \alpha) \]
\[ \text{tienvs} : (L^* \rightarrow C) \rightarrow E^* \rightarrow C \]
\[ \text{tienvs} = \]
\[ \lambda \psi \cdot e \cdot \#e^* = 1 \rightarrow \psi (e^* \downarrow 1), \]
\[ \text{wrong number of return values} \]
\[ \text{tienvsrest} : (L^* \rightarrow C) \rightarrow E^* \rightarrow N \rightarrow C \]
\[ \text{tienvsrest} = \]
\[ \lambda \psi \cdot e \cdot \text{list } (\text{dropfirst } e \cdot \exp) \]
\[ \text{dropfirst} = \lambda n \cdot n = 0 \rightarrow I, \text{dropfirst } (I^* \downarrow 1) (n - 1) \]
\[ \text{takefirst} = \lambda n \cdot n = 0 \rightarrow \}
\[ \text{truish} : E \rightarrow T \]
\[ \text{truish} = \lambda e \cdot e = \text{false } \rightarrow \text{false } \]
\[ \text{true } \]
\[ \text{permute} : \text{Exp}^* \rightarrow \text{Exp}^* \]
\[ \text{permute} = \]
\[ \lambda \psi \cdot e \cdot \text{list } (\text{dropfirst } e \cdot \exp) \]
\[ \text{send} : E \rightarrow K \rightarrow C \]
\[ \text{send} = \]
\[ \lambda \psi \cdot e \cdot \text{list } (\text{dropfirst } e \cdot \exp) \]
\[ \text{applic}e = \]
\[ \lambda e^* . e \in F \rightarrow (e \mid F \downarrow 1) e^* \cdot \exp, \text{wrong “bad procedure”} \]
\[ \text{onary} : (E \rightarrow K \rightarrow C) \rightarrow (E^* \rightarrow K \rightarrow C) \]
\[ \text{onary} = \]
\[ \lambda e^* . e \cdot \exp \rightarrow (e \mid F \downarrow 1) e^* \cdot \exp, \text{wrong “bad procedure”} \]
\[ \text{twoary} : (E \rightarrow E \rightarrow K \rightarrow C) \rightarrow (E^* \rightarrow K \rightarrow C) \]
\[ \text{twoary} = \]
\[ \lambda e^* . e \cdot \exp \rightarrow (e \mid F \downarrow 1) e^* \cdot \exp, \text{wrong “bad procedure”} \]
\[\text{cons} : E^* \rightarrow K \rightarrow C\]
\[\text{cons} = \]
\[\text{twoary}(\lambda e_1 e_2. \text{new}\sigma \in L \rightarrow\]
\[(\lambda \sigma'. \text{new}\sigma' \in L \rightarrow\]
\[
\text{send}((\text{new}\sigma | L, \text{new}\sigma' | \text{true})\in E),\]
\[
\text{update}((\text{new}\sigma' | L)e_2\sigma'),\]
\[
\text{out of memory}''\sigma'\]
\[
\text{out of memory}''\sigma\]
\[
\text{apply} : E^* \rightarrow K \rightarrow C\]
\[\text{apply} = \]
\[\text{twoary}(\lambda e_1 e_2. e_1 \in F \rightarrow \text{valueslist}(e_2)(\lambda e^*. \text{applicte} e_1 e^*)),\]
\[
\text{valueslist} : E^* \rightarrow K \rightarrow C\]
\[\text{valueslist} = \]
\[\text{oneary}(\lambda e_1 e_2. e \in E_p \rightarrow\]
\[
\text{cdr}(e),\]
\[
\text{values} (\lambda e^*. \text{valueslist})\]
\[\text{add} : E^* \rightarrow K \rightarrow C\]
\[\text{add} = \]
\[\text{twoary}(\lambda e_1 e_2. e_1 \in R \land e_2 \in R) \rightarrow\]
\[
\text{send}((e_1 | R + e_2 | R \rightarrow \text{true}, \text{false})),\]
\[
\text{add} (\text{non-numeric argument to <})\]

7.3. Derived expression types

This section gives rewrite rules for the derived expression types. By the application of these rules, any expression can be reduced to a semantically equivalent expression in which only the primitive expression types (literal, variable, call, lambda, if, set!) occur.

\[(\text{cond} (\text{(test)} (\text{sequence}))\]
\[
\text{(clause) \ldots} \]
\[
\equiv \text{(if} (\text{test})\]
\[
\text{begin (sequence)}\]
\[
\text{(cond} (\text{clause}) \ldots)\]

\[(\text{cond} (\text{(test)})\]
\[
\text{(clause) \ldots} \]
\[
\equiv \text{(or} (\text{test} (\text{cond} (\text{clause}) \ldots))\]

\[(\text{cond} (\text{(test)} \Rightarrow (\text{recipient}))\]
\[
\text{(clause) \ldots} \]
\[
\equiv \text{(let} (\text{(test-result} (\text{test})\]
\[
\text{(thunk2} (\text{lambda} (\text{recipient})))\]
\[
\text{(thunk3} (\text{lambda} (\text{(cond} (\text{clause}) \ldots))))\]
\[
\text{if} \text{test-result}\]
\[
\text{(thunk2} \text{test-result}\]
\[
\text{thunk3})))\]