# Scheme Computability

What can and can't Scheme do? We can begin to pin this question down by asking what functions can be computed by Scheme procedures.

Computability theory is traditionally developed using functions on the natural numbers. Computations on other types of objects such as programs, formulas, or graphs are modelled by coding these objects into natural numbers. An advantage of developing computability theory using Scheme is that Scheme programs are a special case of *S-expressions* that have an immediate representation in Scheme without indirect coding. So we consider computability over the set of S-expressions.

However, we will restrict Scheme numbers to be integers, and also omit Scheme expressions containing numerical operators like `/` or `sin` that do not return integer values. This avoids the ambiguities of numerical analysis—how accurately should the value of `(sin 3)` be calculated?, as well as other messy details of general numerical calculation—e.g., in MIT Scheme, `(= 1 1.0000000000000001)` returns `#t`,

## 1 S-Expressions and Printable Syntactic Values

The Scheme printer has standard ways of displaying certain returned values: strings and booleans are "self-evaluating" which means they print out "as themselves." Symbols print out as their names, and lists print out as a parenthesized sequence of the standard printed representations of the list elements. We call these the *printable values*. For example, the Scheme expression,

```
(list 'a 2 3 (list) "6.844 is" ("list great"))        (P)
```

describes a printable value that would print out as:

```
(a 2 3 () "6.844 is" ("great")).        (print(P))
```

The only kind of values that are not printable are procedures, list structures containing procedures, and circular `cons`-cell structures. [1]

The standard printed representations we consider are called *S-expressions*. Formally, they are defined by the following simple grammar:

$$\langle \text{s-expr} \rangle ::= \langle \text{integer} \rangle \mid \langle \text{boolean} \rangle \mid \langle \text{string} \rangle \mid \langle \text{identifier} \rangle \mid \texttt{(} \langle \text{s-expr} \rangle^* \texttt{)}$$



[1] In real Scheme, nonlist pairs are included among the values with standard printed representations. These would be displayed with "dotted pair" notation. For example, the value of

```
(cons (quote a) (cons 2 "bb"))
```

would be displayed by the Scheme printer as `(a 2 . "bb")`. Such non-list pairs are of no particular importance, and for simplicity, we exclude them from the class of values we consider printable.

In the Scheme Substitution Model, printable values are represented by a subset of ⟨syntactic-value⟩'s called the *printable syntactic values*, ⟨printable-sval⟩. For any ⟨printable-sval⟩, $P$, we let print($P$) be its printed representation. The printable syntactic values can also be described by a simple grammar:

$$\begin{aligned} \langle\text{printable-sval}\rangle &::= \langle\text{integer}\rangle \mid \langle\text{boolean}\rangle \mid \langle\text{string}\rangle \mid \texttt{(quote } \langle\text{identifier}\rangle\texttt{)} \\ &\quad \mid \langle\text{nil}\rangle \mid \texttt{(list } \langle\text{printable-sval}\rangle^{+}\texttt{)} \\ \langle\text{nil}\rangle &::= \texttt{(list)} \end{aligned}$$

Scheme supports a special form, `quote`, to create printable values. Namely, for any S-expression, $S$, the Scheme expression `(quote` $S$`)` has a printable value which prints out as $S$.

We can extend our Scheme Substitution Model to handle such quoted S-expressions by applying "desugaring" rules to translate quoted expressions into expressions of *kernel* Scheme (in which quotes only apply to identifiers). The straightforward rules are given in an appendix. The ⟨printable-sval⟩'s are precisely the ⟨syntactic-value⟩'s obtained by desugaring quoted S-expressions.

**Problem 1.** **(a)** Give a precise recursive definition of the function print(.) : ⟨printable-sval⟩ → ⟨s-expr⟩.

**(b)** Give a precise recursive definition of the unprint(.) function that maps an S-expression, $S$, to the ⟨printable-sval⟩, $P$, such that $S = \text{print}(P)$.

**Problem 2.** Define a Scheme expression, Prnbl?, such that evaluation of `(`Prnbl? $M$`)`

$$\text{returns} \begin{cases} \texttt{\#t} & \text{if } M \text{ returns some } \langle\text{printable-sval}\rangle, \\ \texttt{\#f} & \text{if } M \text{ returns some nonprintable value.} \end{cases}$$

for every Scheme expression, $M$.

Your Prnbl? procedure should work on "real" expressions $M$ which may include `set-car!` and similar list-mutating procedures and should return `#f` if $M$ returns a list structure with circular *or shared* substructures.

## 2 The Quote-mark Abbreviation

The notation `'`$S$ is a convenient abbreviation for the S-expression `(quote` $S$`)`. So, for example, we would write

`'('`$a$ `'2 b 3)`

as shorthand for

`(quote ((quote a) (quote 2) b 3))`. (1)

The Scheme expression (1) desugars into a ⟨printable-sval⟩ in kernel Scheme:

```
(list (list (quote quote) (quote a))
      (list (quote quote) 2) (quote b) 3).
```

Of course, we could abbreviate this expression with the shorthand description

```
(list (list 'quote 'a) (list 'quote 2) 'a 3).
```
(2)

Real Scheme implementations support the quote-mark abbreviation, but in these notes we will treat it purely as a notational shorthand, not an extension of Scheme.

## 3 Computable Functions on S-Expressions

In developing computability theory in a way that covers integers or strings, for example, it will be helpful to regard familiar functions such as addition of integers or duplication of strings (concatenating a string with a copy of itself) as *partial functions* with arguments ranging over *all* S-expressions. So the integer addition function will be considered a function on S-expressions that is undefined if any of its arguments is not actually an integer. Likewise, the string duplication function is undefined when its argument is not a ⟨string⟩.

The *domain*, domain $(f)$, of a partial function, $f$, is the set of elements, $s$, such that $f(x)$ is defined. So the domain of the addition function is the set of all pairs of integers, and the domain of the string duplication function is the set of strings. In the same way, we consider the print function, print(.), to be a partial function on ⟨s-expr⟩'s with domain $(\text{print}(.)) = \langle\text{printable-sval}\rangle$.

**Definition 3.1.** Let final-value(.) be the partial function mapping Scheme expressions to their final values, if any. That is, the domain of of final-value(.) is the set of a⟨expression⟩'s, $M$ such that $M{\downarrow}$, and

$$\text{final-value}(M) = V \quad \text{iff} \quad M \downarrow V.$$

Let output$(M)$ be the printed form, if any, of final-value$(M)$. That is,

$$\text{output}(M) ::= \text{print}(\text{final-value}(M)).$$

If $f$ is an $n$-argument partial function on S-expressions, we say that a Scheme expression, $F$, computes $f$ iff

$$\text{output}(\texttt{(}F\ \texttt{'}S_1\ \ldots\ \texttt{'}S_n\texttt{)}) = f(S_1, \ldots, S_n). \tag{3}$$

for all S-expressions $S_1, \ldots, S_n$.

Note that if $f, g$ are partial functions, then by standard convention, the equality $f(a) = g(a)$ is considered to hold when $f(a)$ and $g(a)$ are both undefined, as well as when both are defined and have equal values. So if $F$ computes the partial function $f$, and $S \notin$ domain $(f)$, then by convention output(`(`$F$ `'`$S$`)`) must not be defined, namely, evaluation of `(`$F$ `'`$S$`)` does not result in a printable value.

Notice that this definition works as expected on integers and strings. For example, integer addition is computed by the Scheme builtin `+`, and string duplication is computed by

```
(lambda (s) (string-append s s)).
```

**Problem 3.** Show that if $f$ is an $n$-argument computable function, then there is a closed (no free variables) expression, $F$, that computes $f$ such that `(`$F$ `'`$S_1$ `...` `'`$S_n$`)`$\uparrow$ whenever $f(S_1, \ldots, \S_n)$ is undefined. *Hint:* Use the procedure Prnbl? of Problem 2.

**Problem 4.** Explain why the output$(.)$ function is computable. This is a fundamental result we'll discuss further in Part II of these Notes.

We know a lot about Scheme programming, and this translates to knowing a lot about the properties of computable functions. For example,

**Lemma 3.2.** *The computable functions of one argument are closed under composition.*

*Proof.* If $F$ computes $f$ and $G$ computes $g$, then

$$C ::= \texttt{(lambda (x) (}F\ \texttt{(}G\ \texttt{x)))}$$

computes $f \circ g$. This claim about the expression $C$ will be unsurprising to anyone familiar with Scheme. It could be proved rigorously by appeal to the Control-context Independence of the Substitution Model rules, but a careful proof here would be a distraction from our theme of computability, and so we omit it.

□

**Problem 5.** Suppose $f$ is a two-argument function on the natural numbers that is a total computable function. Prove that the function $\min[f]$ is partial computable, where

$$\min[f](n) ::= \min\{k \in \mathbb{N} \mid f(n,k) = 0\}.$$

**Problem 6.** Let $f$ be the partial function whose domain is the set of Arithmetic Expressions from previous Notes, where $f(e)$ is the canonical form of $e$. (You may assume that variables are ordered by Scheme's `symbol<?`) Explain why $f$ is Scheme computable.

**Definition 3.3.** A set, $\mathcal{S}$, of S-expressions is Scheme *decidable* iff its membership function is Scheme computable, that is,the function $m_{\mathcal{S}} : \langle\text{s-expr}\rangle \to \langle\text{boolean}\rangle$ such that

$$m_{\mathcal{S}}(S) ::= \begin{cases} \texttt{\#t} & \text{for } S \in \mathcal{S}, \\ \texttt{\#f} & \text{for } S \notin \mathcal{S}, \end{cases}$$

is Scheme computable. A Scheme expression that computes the membership function is called a *decider* for $\mathcal{S}$.

For example, the Scheme builtin `string?` is a decider for the set ⟨string⟩ of strings.

**Problem 7.** Explain why the set, ⟨expression⟩, of Scheme expressions, and the set of *closed* ⟨expression⟩'s, are both decidable. *Hint:* See the `free-variables` procedure defined in the Substitution Model implementation.

For any set, $\mathcal{S}$, of S-expressions, let $\overline{\mathcal{S}}$ be the complement of $\mathcal{S}$, that is,the set of S-expressions *not* in $\mathcal{S}$.

**Lemma 3.4.** *If $\mathcal{S}$ is Scheme decidable, then so is $\overline{\mathcal{S}}$.*

*Proof.* Let $F$ be an expression that computes $m_{\mathcal{S}}$. Then $N ::=$ `(lambda (x) (not (`$F$ `x)))` computes $m_{\overline{\mathcal{S}}}$.

The correctness of $N$ follows from the fact that the Substitution Model rules preserve observational equivalence (Notes 8, Lemma 4.4). Again, we omit the correctness proof. □

**Problem 8.** Prove that if $\mathcal{S}_1$ and $\mathcal{S}_2$ are Scheme decidable, then so are $\mathcal{S}_1 \cup \mathcal{S}_2$ and $\mathcal{S}_1 \cap \mathcal{S}_2$.

**Definition 3.5.** For any Scheme computable partial function, $f : \langle\text{s-expr}\rangle \to \langle\text{s-expr}\rangle$, and set, $\mathcal{S}$, of S-expressions, define

$$\begin{aligned} f(\mathcal{S}) &::= \{f(S) \mid S \in (\mathcal{S} \cap \text{domain}\,(f))\}\,, \\ f^{-1}(\mathcal{S}) &::= \{S \mid S \in \text{domain}\,(f) \text{ and } f(S) \in \mathcal{S}\}\,. \end{aligned}$$

**Problem 9.** Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be sets of S-expressions. Verify that

$$\mathcal{S}_1 \leq_{\text{m}} \mathcal{S}_2,$$

iff there is a computable total function, $f$, such that $\mathcal{S}_1 = f^{-1}(\mathcal{S}_2)$. The function, $f$, is said to *many-one reduce* $\mathcal{S}_1$ to $\mathcal{S}_2$.

**Problem 10.** Prove that if $f$ is a Scheme computable total function, and $\mathcal{S}$ is decidable, then so is $f^{-1}(\mathcal{S})$.

So we could rephrase the result of Problem 10 as saying that

> *Decidability inherits downward* under many-one reducibility.

**Problem 11.** Prove that $A \leq_{\mathrm{m}} B$ iff $\overline{A} \leq_{\mathrm{m}} \overline{B}$.

The "$\leq$" notation for many-one reducibility highlights the fact that

**Lemma 3.6.** *Many-one reducibility is transitive.*

*Proof.* Suppose $f$ many-one reduces $\mathcal{S}_1$ to $\mathcal{S}_2$, and $g$ many-one reduces $\mathcal{S}_2$ to $\mathcal{S}_3$, then $g \circ f$ many-one reduces $\mathcal{S}_1$ to $\mathcal{S}_3$. □

**Definition 3.7.** A set $\mathcal{S}$ of S-expressions is *recognizable* iff there is a Scheme computable partial function whose domain is $\mathcal{S}$. A Scheme expression that computes such a function is called a *recognizer* for $\mathcal{S}$. Recognizable sets are usually called *recursively enumerable* (r.e.) sets.

**Lemma 3.8.** *If a set of S-expressions is decidable, then it and its complement are also recognizable.*

*Proof.* Suppose $\mathcal{D}$ is decidable. Then it has a decider, $D$. Then

```
(lambda (x) (or (D x) Ω0))
```

is a recognizer for $\mathcal{D}$, where $\Omega_0$ is any closed expression such that $\Omega_0\uparrow$. For example, let

$$\Omega_0 ::= \texttt{(letrec ((t (lambda () (t)))) (t))}.$$

By Lemma 3.4, $\overline{\mathcal{D}}$ is also decidable, and so is also recognizable. □

The converse of Lemma 3.8 also holds: if a set and its complement are both recognizable, then the set is wholly decidable. We'll postpone the proof of this converse to Part II.

**Problem 12.** **(a)** Prove that if $\mathcal{S}_1$ and $\mathcal{S}_2$ are recognizable, then so is $\mathcal{S}_1 \cap \mathcal{S}_2$.

**(b)** Prove that if $f$ is a Scheme computable partial function, and $\mathcal{S}$ is recognizable, then so is $f^{-1}(\mathcal{S})$.

Note that as a special case of Problem 12(b), it follows that

> Recognizability inherits downward under many-one reducibility.

**Definition 3.9.** A nonempty set, $\mathcal{S}$, of S-expressions is *computably countable* iff $\mathcal{S} = f(\mathbb{N})$ for some computable partial function, $f : \langle\text{s-expr}\rangle \to \langle\text{s-expr}\rangle$, such that $\mathbb{N} \subseteq \text{domain}(f)$. Such an $f$ is said to *count* $\mathcal{S}$.

In other words, there is a procedure to enumerate all the elements of $\mathcal{S}$, possibly with repetitions, in some order, namely, successively compute $f(0), f(1), \ldots$.

**Lemma 3.10.** *Every computably countable set is recognizable.*

*Proof.* Let $\mathcal{S}$ be computably countable, so $\mathcal{S} = f(\mathbb{N})$ for some $f$ as in Definition 3.9. Then a procedure to recognize whether any given input, $S$, is in $\mathcal{S}$, is to compute $f(0), f(1), \ldots$ stopping if and when $S$ shows up in the list. So the following expression is a recognizer for $\mathcal{S}$, where $F$ is an expression that computes $f$:

```
(lambda (s)
  (letrec
   ((try (lambda (n)
           (if (equal? s (F n)) #t (try (+ n 1))))))
           (try 0)))
```

□

In Part II we'll prove the converse of Lemma 3.10, allowing us to conclude that a nonempty set is computably countable iff it is recognizable. For now, it's informative to solve the following problems without assuming this fact.

**Problem 13.** Suppose $\mathcal{S}$ and $\mathcal{S}'$ are computably countable. Prove that:

**(a)** If $g$ is a total computable function, then $g(\mathcal{S})$ is computably countable.

**(b)** $\mathcal{S} \cup \mathcal{S}'$ and $\mathcal{S} \cap \mathcal{S}'$ are computably countable.

**(c)** $\mathcal{S}^+ ::= \{(S_1 \ldots S_n) \mid S_i \in \mathcal{S} \text{ for } 1 \leq i \leq n\}$ is computably countable.

**Problem 14.** **(a)** Prove that the set of ⟨string⟩'s is computably countable.

**(b)** Conclude that the set of symbols is computably countable. *Hint:* Look up `string->symbol` in Revised[5] Scheme Manual.

**(c)** Prove the set of S-expressions is computably countable.

# 4 Applications of Self Applications

## 4.1 Self-Reproducing Expressions

For practice with quoting, and in preparation for the non-computability arguments in the next section, we consider how to make "self-reproducing" Scheme expressions. A Scheme expression, $P$, is *self-reproducing* iff evaluation of $P$ returns a value that prints out as $P$, that is,

$$\text{output}(P) = P.$$

All self-evaluating expressions have this property of course, but it's not so obvious how to find one that is not self-evaluating. Here's how: let $L$ be an expression such that

$$\text{output}((L\ \ 'S)) = (S\ \ 'S) \tag{4}$$

for any S-expression, $S$. For example, we could define

$$L ::= \texttt{(lambda (s) (list s (list 'quote s)))}.$$

Now substituting $L$ for $S$ in (4) above yields

$$\text{output}((L\ \ 'L)) = (L\ \ 'L).$$

In other words, we can choose $P$ to be `(`$L$ `'`$L$`)`, namely,

```
P::=((lambda (s) (list s (list (quote quote) s)))
     (quote (lambda (s) (list s (list (quote quote s))))))).
```

**Problem 15.** **(a)** Check that this last $P$ is self-reproducing by evaluating it in MIT Scheme.[2]

**(b)** Exhibit two other self-reproducing expressions and check them in real Scheme. Turn in your pretty-printed output. *Hint:* $L$ need only satisfy the specification (4).

**Problem 16.** An expression $D$ is *doubly self-reproducing* iff output$(D)$ = `(`$D$ $D$`)`. Exhibit a doubly self-reproducing expression and check that it works in real Scheme. Turn in your output.

## 4.2 The Y Operator [Optional]

Self application provides a way to formulate recursive definitions in Scheme without using `letrec`, `define`, or `set!`. Although mainly a curiosity, one corollary of the construction is that a small fragment of Scheme, containing only variables, combinations, and `lambda` expressions—no numbers, lists or other data types, nor any special forms besides `lambda`—can simulate the full language, and therefore this fragment inherits all the undecidability properties of Scheme.

To explain how this works, let's begin with one of the most familiar examples:

```
(define factorial
        (lambda (x)
          (if (zero? x) 1 (* x (factorial (- x 1))))))
```

A way to understand this simple recursive definition of `factorial` begins with the observation that the variable `factorial` occurs free in the body of the definition. So we can regard the body as a function, $L$(factorial), of this free variable, namely, $L$ is defined by

[2] This works in MIT Scheme because the printer displays the value of `(quote (quote a))` as `(quote a)`. Other Scheme printers maintain the quote-mark abbreviation in their output and display the value of `(quote (quote a))` as `'a`.

```
(lambda (factorial)
   (lambda (x) (if (zero? x) 1 (* x (factorial (- x 1))))))
```

It will simplify the discussion if we rename the parameter to be `f`:

```
(define L
  (lambda (f)
    (lambda (x) (if (zero? x) 1 (* x (f (- x 1)))))))
```

Now the `factorial` procedure is a "fixed point" of `L`:

$$\texttt{factorial} \equiv \texttt{(L factorial)}.$$

In general, a fixed point of a function, $L$, is an element, $f$, such that $L(f) = f$. A "fixed point operator," $Y$, is used to obtain fixed points of procedures like $L$. Informally, we want $Y(L) = L(Y(L))$. More precisely, we want a Scheme procedure, `Y`, satisfying:

$$\texttt{(Y l)} \equiv \texttt{(lambda (z) ((l (Y l)) z))}.$$

Notice that we have wrapped the righthand side of this equivalence in `(lambda(z) (...z))`. This ensures that `(Y l)` will converge in any environment in which `l` is defined.

So instead of defining factorial recursively, we could instead have written:

```
(define factorial
  (Y (lambda (f) (lambda (x) (if (zero? x) 1 (* x (f (- x 1))))))))
```

To arrive at a definition of `Y`, let

$$M_{\texttt{l}} ::= \texttt{(lambda (x) (lambda (z) ((l (x x)) z)))},$$

so

$$\texttt{(}M_{\texttt{l}}\ \texttt{x)} \equiv \texttt{(lambda (z) ((l (x x)) z))}.$$

Then,

$$\texttt{(}M_{\texttt{l}}\ M_{\texttt{l}}\texttt{)} \equiv \texttt{(lambda (z) ((l (}M_{\texttt{l}}\ M_{\texttt{l}}\texttt{)) z))}.$$

That is, `(`$M_{\texttt{l}}\ M_{\texttt{l}}$`)` is the desired fixed point of `l`, so we could define `(Y l)` to be `(`$M_{\texttt{l}}\ M_{\texttt{l}}$`)`:

```
(define Y
  (lambda (l)
    ((lambda (x) (lambda (z) ((l (x x)) z)))
     (lambda (x) (lambda (z) ((l (x x)) z))))))
```

**Problem 17.** Evaluate `((Y L) 3)` in Scheme and in the Substitution Model interpreter, using the definitions `Y` and `L` above.

**Problem 18.** Explain what happens if we omitted the `(lambda (z) ...z)` wrapper and used the definition:

```
(define Y
  (lambda (l)
    ((lambda (x) (l (x x)))
     (lambda (x) (l (x x))))))
```

**Problem 19.** Adapt the definition of `Y` so it works for multi-argument fixed points, namely, so

$$\texttt{(Y f)} \equiv \texttt{(lambda l (apply (f (Y f)) l))}.$$

# 5 The Halting Problem

We aim to prove the most famous theorem in Computability Theory: the undecidability of the Halting Problem. The problem is to determine whether the evaluation of any given Scheme expression will "halt." To formalize this, we'll interpret halting to mean converging. It will simplify matters if we extend the Substitution Model to all S-expressions, with the convention that if any S-expression that is not a Scheme ⟨expression⟩ will be counted as an immediate error. In particular, if an ⟨s-expr⟩, $S$, is not an ⟨expression⟩, then $S\uparrow$ by convention. Now define

$$\text{Halts} ::= \{S \in \langle \text{s-expr} \rangle \mid S\downarrow\}\,.$$

**Problem 20.** Prove that every recognizable set is many-one reducible to Halts.

We remark that Halts itself is recognizable: if $E$ is a Scheme expression defining an interpreter for Scheme (what Abelson-Sussman call a "meta-circular interpreter") then

$$\texttt{(}E\ \texttt{'}M\ \texttt{empty-env)}\downarrow \quad \text{iff} \quad M\downarrow$$

for all Scheme expressions, $M$. So a meta-circular Scheme interpreter would provide a recognizer for Halts. We'll explore this further in Part II.

By Lemma 3.8, this remark implies that showing that Halts is undecidable is equivalent to showing that its complement is not even recognizable.

**Theorem 5.1.** *(The Halting Theorem)* $\overline{\text{Halts}}$ *is not recognizable.*

We'll prove this indirectly, by proving two Lemmas:

**Lemma 5.2.** *Let*

$$\text{Self-Halts} ::= \{S \mid \texttt{(}S\ \texttt{'}S\texttt{)}\downarrow\}\,.$$

*Then*

$$\text{Self-Halts} \leq_{\text{m}} \text{Halts}.$$

*Proof.* (of Theorem 5.1) By definition,

$$S \in \text{Self-Halts} \quad \text{iff} \quad \texttt{(}S\ \texttt{'}S\texttt{)} \in \text{Halts},$$

for all S-expressions, $S$. So the mapping, $f$, many-one reduces Self-Halts to Halts, where $f(S) ::=$ `(`$S$ `'`$S$`)` is computed by the expression

```
(lambda (s) (list s (list 'quote s)))
```

□

**Lemma 5.3.** *The complement,* $\overline{\text{Self-Halts}}$*, of* Self-Halts *is not recognizable.*

Note that by Lemma 5.2, $\overline{\text{Self-Halts}} \leq_{\text{m}} \overline{\text{Halts}}$. Since recognizability inherits downward, we conclude that if $\overline{\text{Halts}}$ was recognizable, then $\overline{\text{Self-Halts}}$ would also be recognizable, contradicting Lemma 5.3. This proves the Halting Theorem.

So it remains only to prove Lemma 5.3. The definition of Self-Halts was chosen to make the proof almost immediate:

*Proof.* (of Lemma 5.3) Let $\mathcal{H}$ be a recognizable set, and $H$ a recognizer for it. So by definition, of $H$,

$$S \in \mathcal{H} \quad \text{iff} \quad (H \ \ 'S)\downarrow,$$

for all S-expressions, $S$. Letting $S$ be $H$, we have

$$H \in \mathcal{H} \quad \text{iff} \quad (H \ \ 'H)\downarrow. \tag{5}$$

But by definition of Self-Halts,

$$(H \ \ 'H)\downarrow \quad \text{iff} \quad H \notin \overline{\text{Self-Halts}}. \tag{6}$$

From (5) and (6), we conclude

$$H \in \mathcal{H} \quad \text{iff} \quad H \notin \overline{\text{Self-Halts}}. \tag{7}$$

In particular, (7) implies that

$$\mathcal{H} \neq \overline{\text{Self-Halts}}$$

because $H$ is in one of these sets and not the other. Since $H$ was an arbitrary S-expression, it follows that $\overline{\text{Self-Halts}}$ is not equal to *any* recognizable set, that is, it is not recognizable. □

Any S-expression, $M$, is by definition a recognizer for a set $\mathcal{H}_M$:

**Definition 5.4.**

$$\mathcal{H}_M ::= \{S \in \langle\text{s-expr}\rangle \mid (M \ \ 'S)\downarrow\}.$$

For sets $A, B$, we say that an element, $a$, is a *witness* that $A \neq B$ when $a$ is in one of the sets and not the other. That is, $a \in (A - B) \cup (B - A)$. The proof of Lemma 5.3 shows that finding a witness that $\mathcal{H}_M \neq \overline{\text{Self-Halts}}$ is trivial: the expression $M$ is a witness. More generally, a set, $\mathcal{P}$, is said to be *productive* when there is a Scheme program that, given $M$, finds a witness that $\mathcal{P} \neq \mathcal{H}_M$:

**Definition 5.5.** A set, $\mathcal{P}$, of S-expressions is *productive* iff there is a total computable function, $w : \langle\text{s-expr}\rangle \to \langle\text{s-expr}\rangle$, such that

$$w(M) \in \mathcal{H}_M \quad \text{iff} \quad w(M) \notin \mathcal{P}.$$

Such a function, $w$, is called a *witness function* for $\mathcal{P}$.

Clearly, no productive set can be recognizable, since it differs from every recognizable set. So now we can rephrase the conclusion that comes out of the proof of Lemma 5.3: the set $\overline{\text{Self-Halts}}$ is productive with witness function equal to the identity function on S-expressions. Of course $\overline{\text{Self-Halts}}$ was carefully contrived to be productive with a trivial witness function, but there are many uncontrived examples. For example, $\overline{\text{Halts}}$ is also productive. This follows from the fact that $\overline{\text{Self-Halts}} \leq_{\text{m}} \overline{\text{Halts}}$ along with:

**Lemma 5.6.** *Productivity inherits upward under many-one reducibility.*

The concept of productivity will be useful when we return to a discussion of proof systems.

**Problem 21.** Prove Lemma 5.6.

**Problem 22.** Prove that Halts and $\overline{\text{Halts}}$ are *incomparable* under many-one reducibility.

# 6 Incompleteness

Suppose we have some formal notation for expressing mathematical assertions, and some system for proving assertions. The proof system is called *sound* if all the provable asertions are actually valid. The proof system is called *complete* if all the valid assertions are provable. In previous Notes, we saw a sound and complete proof system for arithmetic equalities.

In considering Scheme observational equivalences, we will use assertions that are S-expressions in the form of equations between Scheme expressions. In this case, the valid assertions will be the set, $\mathcal{E}$, of equations that are true when equality is interpreted to be observational equivalence. That is,

$$\mathcal{E} ::= \{(M \ = \ N) \mid M \equiv N\}.$$

## 6.1 First Incompleteness Theorem

We aim to prove:

**Theorem 6.1.** ***First Incompleteness Theorem for Scheme equivalence****: If a proof system for Scheme observational equivalences is sound, then it is incomplete.*

Notice that this is a wonderfully general theorem, which applies to *all possible* proof systems, not just some particular ones we might devise based on the various observational equivalences we have established up to this point. Of course, to be truly general, we need a notion of "proof system" that leaves no loopholes: every possible set of axioms and inference rules should count as a proof system; in fact, we want any procedure for determining validity to count as a proof system. For example, a system for proving arithmetic equations by transforming the two sides of the equation into identical canonical forms should count as a proof system—one that we know is also sound and complete.

There is one essential property we will require of a proof system. The purpose of proving an assertion is to confirm the truth of the assertion even to someone who can't understand the proof. They need only be able to check—in a purely mechanical way—that the proof is well-formed according to the rules of the proof system.

In particular, a proof system has things called *proofs* that serve to prove things called *assertions*. In a formal proof system, the assertions and proofs are objects that we can safely assume are

represented by S-expressions. The condition that a proof be checkable "without understanding" can be captured by requiring that there be a procedure for checking whether an S-expression is a proof of an assertion. That is, the set of S-expressions of the form $(A \;\; P)$, where $A$ is an assertion and $P$ is a proof of $A$ in the proof system, should be a *decidable* set, $\mathcal{D}$, of S-expressions.

Now we have a way to recognize the provable assertions in the proof system: on input $A$, starting generating all the possible S-expressions, $S_0, S_1, \ldots$; this is possible because the set of S-expressions is computably countable (Problem 14). As the S-expressions are generated, successively apply the decider for $\mathcal{D}$ to $(A \;\; S_0), (A \;\; S_1), \ldots$ until the decider returns `#t`. So if $A$ has a proof, this procedure will eventually find it and halt. Conversely, if $A$ does not have a proof, then this procedure will search forever without terminating. So this proof-search procedure is a recognizer for the set of assertions provable in the system. So we conclude:

**Theorem 6.2.** *The set of assertions provable using any given proof system is recognizable.*

**Problem 23.** Sketch how to write a Scheme program computing a decider for proofs in the arithmetic equation proof system of the Notes.

Notice that Theorem 6.2 holds regardless of the meaning of assertions. But if assertions are meaningful and a proof system is sound, then it follows that the provable assertions are a recognizable subset of the valid assertions. Now, if we show that the valid assertions are not recognizable, then the assertions provable using any given system must be a *proper* subset of the valid assertions. In other words, there must be a valid assertion that is not provable: the system is *incomplete*.

Therefore, to prove the First Incompleteness Theorem for Scheme equivalence, we need only show that the set, $\mathcal{E}$, of true equivalences is not recognizable. In fact, we can prove something stronger:

**Lemma 6.3.** *$\mathcal{E}$ is productive.*

*Proof.* We showed in the Notes on the Scheme Substitution Model that all divergent expressions are observationally equivalent. So by our convention that S-expressions that are not Scheme ⟨expression⟩'s are immediate errors, we have

$$S{\uparrow} \quad \text{iff} \quad S \equiv \Omega_0,$$

for all S-expressions, $S$.

Note that an expression whose evaluation leads to a lookup errors does not converge, but may not be observationally equivalent to $\Omega_0$. However, there is a simple way to transform any S-expression, $S$, into an expression, $\widehat{S}$, that diverges in the event of a lookup error. That is,

$$\widehat{S}{\uparrow} \quad \text{iff} \quad S{\downarrow}. \tag{8}$$

**Problem 24.** Describe a procedure mapping $S$ to an expression $\widehat{S}$ satisfying (8).

Now let $f : \langle\text{s-expr}\rangle \to \langle\text{s-expr}\rangle$ be defined by the rule

$$f(S) ::= (\widehat{S} \texttt{ = } \Omega_0)\,.$$

Then

$$\begin{array}{lll} S \in \overline{\text{Halts}} & \text{iff} & S\not\downarrow \\ & \text{iff} & \widehat{S} \equiv \Omega_0 \\ & \text{iff} & f(S) \in \mathcal{E}. \end{array}$$

Since $f$ is easy to program in Scheme, we conclude that $\overline{\text{Halts}} \leq_{\text{m}} \mathcal{E}$. Since we know $\overline{\text{Halts}}$ is productive, we have by Lemma 5.6 that $\mathcal{E}$ is also productive.

□

A consequence of Lemma 6.3 is that given any proof system for Scheme equivalences, we can find a witness to its imperfection. Specifically, given a recognizer for the equations provable in the system, we can apply the witness function for $\mathcal{E}$ to the recognizer to obtain an equation that is either

- provable but not in $\mathcal{E}$, implying that the system is unsound, or
- in $\mathcal{E}$ but not provable, implying that the system is incomplete.

Notice that we can accomplish this without assuming that the system is sound.

A final technical remark: the proof of Lemma 6.3 also demonstrates that $\overline{\text{Halts}} \leq_{\text{m}} \mathcal{E}_0$ where

$$\mathcal{E}_0 ::= \{(M \texttt{ = } \Omega_0) \mid M \equiv \Omega_0\}\,.$$

So we have

**Corollary 6.4.** *$\mathcal{E}_0$ is productive.*

## 7 Scott's Rice's Theorem

Two general theorems due to Dana Scott characterize a large class of sets that are either undecidable or not even recognizable. Scott's results extended earlier theorems, due to Rice, to a Scheme-like setting.

**Definition 7.1.** Let $\mathcal{G}$ and $\mathcal{H}$ be sets of S-expressions. A total function, $s : \langle\text{s-expr}\rangle \to \langle\text{s-expr}\rangle$, such that $s(A) = \texttt{\#t}$ for $A \in \mathcal{G}$, and $s(B) = \texttt{\#f}$ for $B \in \mathcal{H}$ is said to be a *separator* of $\mathcal{G}$ and $\mathcal{H}$. The sets are *Scheme separable* iff there is a Scheme computable separator for them. $\mathcal{G}$ and $\mathcal{H}$ are *Scheme inseparable* iff they are not Scheme separable.

By definition, any two nondisjoint sets will trivially be inseparable. Also, a set is decidable iff it and its complement are Scheme separable. This follows because the membership function for a set is, by definition, a separator of the set and its complement. In fact, the membership function separates a decidable set from every set contained in its complement. So two *disjoint* Scheme inseparable sets must both be undecidable.

**Definition 7.2.** A set $\mathcal{S}$ of S-expressions is *submodel-invariant* if

$$(M \xrightarrow{*} N \text{ and } N \in \mathcal{S}) \quad \text{implies} \quad M \in \mathcal{S}$$

for all closed Scheme expressions $M, N$.

For example, Halts is submodel-invariant set of S-expressions, because if $M \xrightarrow{*} N$, then obviously $M{\downarrow}$ iff $N{\downarrow}$. Likewise, $\overline{\text{Halts}}$ is submodel-invariant.

**Theorem 7.3.** *(**Scott's Rice's First Theorem.**) Let $\mathcal{G}$ and $\mathcal{H}$ be nonempty, submodel-invariant sets of S-expressions. Then $\mathcal{G}$ and $\mathcal{H}$ are Scheme inseparable.*

So Halts and its complement satisfy the conditions of Theorem 7.3, and so are inseparable. This provides another proof that neither Halts nor its complement is Scheme decidable.

A more interesting example is the sets $\{M \mid \text{final-value}(M) = 1\}$ and $\{M \mid \text{final-value}(M) = 2\}$. These are disjoint sets that obviously satisfy the conditions of Theorem 7.3, so they are Scheme inseparable, and hence neither is decidable.

Finally, let $\mathcal{G}_{\text{store}}$ be the set of expressions, $M$, such that the builtin operation `cons` is applied an infinite number of times in the evaluation of $M$. Then $\mathcal{G}_{\text{store}}$ and its complement are disjoint submodel-invariant sets, and so are both undecidable.

*Proof.* We prove Theorem 7.3 by contradiction: suppose there was a Scheme computable separator, $t$, for $\mathcal{G}$ and $\mathcal{H}$. Let $T$ be the Scheme expression that computes $t$.

By hypothesis, there are Scheme expressions $G_0 \in \mathcal{G}$ and $H_0 \in \mathcal{H}$. Let `s` be a variable not free in $T$, $G_0$ or $H_0$, and define the "*P*erverse" expression,

$$P ::= \texttt{(lambda(s) (if (}T\ \texttt{(list s (list 'quote s)))}\ H_0\ G_0\texttt{))}.$$

Now suppose $S$ is an S-expression such that

$$\texttt{(}T\ \texttt{'(}S\ \texttt{'}S\texttt{))}{\downarrow}\,\texttt{\#t}.$$

Then

$$\texttt{(}P\ \texttt{'}S\texttt{)} \xrightarrow{*} H_0.$$

Hence by submodel-invariance,

$$\texttt{(}P\ \texttt{'}S\texttt{)} \in \mathcal{H}.$$

Now by definition of the separator computed by $T$, we conclude that

$$\texttt{(}T\ \texttt{'(}P\ \texttt{'}S\texttt{))}{\downarrow}\,\texttt{\#f}.$$

Conversely, by the same argument

$$\texttt{(}T\ \texttt{'(}S\ \texttt{'}S\texttt{))}{\downarrow}\,\texttt{\#f} \quad \text{implies} \quad \texttt{(}T\ \texttt{'(}P\ \texttt{'}S\texttt{))}{\downarrow}\,\texttt{\#t}.$$

That is, for every S-expression, $S$,

$$\texttt{(}T\ \texttt{'(}S\ \texttt{'}S\texttt{))}{\downarrow}\,\texttt{\#t} \quad \text{iff} \quad \texttt{(}T\ \texttt{'(}P\ \texttt{'}S\texttt{))}{\downarrow}\,\texttt{\#f}. \tag{9}$$

Now let $S$ be $P$ and then $M$ be `(`$P$ `'`$P$`)` in (9). This yields

$$\texttt{(}T\ \texttt{'}M\texttt{)}\downarrow \texttt{\#t} \quad \text{iff} \quad \texttt{(}T\ \texttt{'}M\texttt{)}\downarrow \texttt{\#f}. \tag{10}$$

But (10) can only hold if `(`$T$ `'`$M$`)`$\uparrow$, contradicting the fact that the separator, $t$, is a total function. □

**Theorem 7.4.** *(**Scott's Rice's Second Theorem.**) Let $\mathcal{S}$ be a set of S-expressions with a submodel-invariant, nonempty, complement. If* $\overline{\text{Halts}} \subseteq \mathcal{S}$*, then $\mathcal{S}$ is not recognizable.*

The proof of the Second Theorem is similar to that of the First.

These two theorems reveal that no Scheme procedure can predict a nontrivial fact about the continuing evaluation or value of an arbitrary Scheme expression presented as input. Since we know that Scheme can simulate any other programming language, we can simply say that no computational procedure whatsoever can reliably determine any nontrivial property of the behavior of Scheme expressions.

**Problem 25.** Prove Scott's Rice's Second Theorem 7.4.

**Problem 26.** Strengthen Theorem 7.4 so that it directly implies that neither the set $\mathcal{G}_{\text{store}}$ above nor its complement are recognizable.

# A Desugaring `quote`

$$\begin{array}{rcll}
\texttt{(quote}\ K\texttt{)} & \rightarrow & K & \text{if } K \text{ is } \langle\text{integer}\rangle, \langle\text{boolean}\rangle, \text{ or } \langle\text{string}\rangle \\
\texttt{(quote ())} & \rightarrow & \langle\text{nil}\rangle & \\
\texttt{(quote (}S_1\ S_2\ \texttt{...))} & \rightarrow & \texttt{(list (quote}\ S_1\texttt{) (quote}\ S_2\texttt{) ...)} &
\end{array}$$

Application of these rules will desugar any $\langle$s-expr$\rangle$, $S$, into a Scheme expression, unprint$(S)$, that is a $\langle$printable-sval$\rangle$. In particular,

$$\texttt{(quote}\ S\texttt{)}\downarrow \text{unprint}(S)$$

for all S-expressions, $S$.