Massachusetts Institute of Technology
6.844, Spring '03: Computability Theory of and with Scheme
Prof. Albert Meyer

Course Notes 5
April 16
revised May 9, 2003, 1138 minutes

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 values returned by evaluation: self-evaluating values like strings and booleans 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. These are called the *printable values*. For example, the value of the Scheme expression

```
(append (list (quote a) 2 3 (list) "6.844 is")          (1)
        (list (cons "great" (quote ()))))
```

is a list structure of strings, symbols, and numbers that would print out as:

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

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

In real Scheme, nonlist pairs are included among the printable values and 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 will exclude them from the class of values we consider printable; likewise, dotted pair notation is not allowed in standard printed representations.

Formally, the grammar for ⟨s-expr⟩'s, the standard printed representations we consider, is:

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

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

We can extend our Scheme Substitution Model to handle quoted S-expressions by applying the following "desugaring" rules to translate quoted expressions into expressions of kernel Scheme—in which quotes only apply to identifiers. [1]

$$\begin{aligned}
\texttt{(quote } K\texttt{)} &\to K \qquad \text{if } K \text{ is } \langle\text{integer}\rangle, \langle\text{boolean}\rangle, \text{or } \langle\text{string}\rangle\\
\texttt{(quote ())} &\to \langle\text{nil}\rangle\\
\texttt{(quote (}S_1\ S_2\ \ldots\texttt{))} &\to \texttt{(list (quote } S_1\texttt{) (quote } S_2\texttt{)} \ldots\texttt{)}
\end{aligned}$$

Application of these rules will desugar S into a Scheme expression, unprint(S), that is actually a *syntactic value* in our formal Substitution Model for Scheme. That is,

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

for all S-expressions, S. For example, the expression (2) would desugar back into (1).

The syntactic values obtained by desugaring quoted S-expressions are called the *printable syntactic values*, ⟨printable-sval⟩. These can also be described by a simple grammar:

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

It is easy to check that unprint(.) is a bijection from ⟨s-expr⟩ to ⟨printable-sval⟩ and print(.) is its inverse.[2] Namely,

$$\begin{aligned}
\text{print}(\text{unprint}(S)) &= S && \text{for all } S \in \langle\text{s-expr}\rangle\,,\\
\text{unprint}(\text{print}(V)) &= V && \text{for all } V \in \langle\text{printable-sval}\rangle\,.
\end{aligned}$$

[1]It's wiser not to mix the desugaring rules with the other Substitution Model rules because Proposition 3.1 below fails when we apply `quote` as an operation. However, the desugaring rules could be included with the rest of the simple control rules without violating Proposition 3.1 because of the redeeming technical condition that ⟨hole⟩'s in contexts may not be quoted.

[2]MIT Scheme regrettably violates the official Scheme standard, using, for historical reasons, the same object for the empty list and the Boolean `#f`. The result is that `(quote ())` and `#f` may *both* print out as `()`—or as `#f`—depending on the printer. Officially, these should be distinct values, `(quote ())` should print out as `()`, and `(quote #f)` should print out as `#f`. This flaw is supposed to be repaired in the next release of MIT Scheme.

Problem 1. Define a Scheme expression, Prnbl?, such that evaluation of (Prnbl? M)

returns `#t`	if M returns some ⟨printable-sval⟩,
returns `#f`	if M returns some nonprintable value,

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`)`. Real Scheme implementations support this abbreviation, but in these notes we will use the quote-mark as a notational shorthand, not an extension of Scheme. So, for example, we would write

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

as shorthand for

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

The Scheme expression (3) unprints into the ⟨printable-sval⟩:

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

that itself has the shorthand description

```
(list (list 'quote 'a) (list 'quote 2) 'a 3).
```
(4)

Formally, ⟨s-expr⟩'s do not include the quote-mark, and Scheme ⟨expression⟩'s are still a subset of ⟨s-expr⟩. So

```
(quote a)
```

is a Scheme ⟨expression⟩, and

```
'a
```
(5)

is shorthand for it, but (5) is not itself an S-expression.

3 Computable Functions on S-Expressions

In these notes, Scheme will be the language without `call/cc` (or `abort`).[3] This allows us to avoid the complexity of reasoning about observational equivalence when these procedures are available, as discussed in the Notes on the Substitution Model. In particular, without `call/cc`, we have from these Notes:

Proposition 3.1. *If $M \xrightarrow{*} N$, then $M \equiv N$.*

Definition 3.2. A partial function $f : \langle\text{s-expr}\rangle^n \to \langle\text{s-expr}\rangle$ is *Scheme computable* iff there is a Scheme expression, F, such that

$$\texttt{(}F\ \ \texttt{'}S_1 \ldots \texttt{'}S_n\texttt{)} \downarrow \text{unprint}(f(S_1, \ldots, S_n))$$

for all $(S_1, \ldots, S_n) \in \text{domain}\,(f)$, and also,

$$\texttt{(}F\ \ \texttt{'}S_1 \ldots \texttt{'}S_n\texttt{)}\uparrow$$

for all $(S_1, \ldots, S_n) \notin \text{domain}\,(f)$. Such an expression F is said to *compute* the partial function f.

Note that we consider integers and strings to be special cases of S-expressions. So functions like addition on the integers or "doubling" on strings (concatenating a string with itself) can be regarded as partial functions on the set of *all* S-expressions. Namely, addition on S-expressions is undefined when either of its arguments is not an integer, and doubling is undefined when its argument is not a ⟨string⟩. Similarly, the print function print(.), which is a total function from ⟨printable-sval⟩ to S-expressions, can be regarded as a partial function on the set of all S-expressions by adopting the convention that print(S) is undefined if S is not a ⟨printable-sval⟩.

This convention allows us to say that addition and doubling are computable partial functions, *e.g.*, the Scheme builtin `+` is an expression that computes the addition function, and

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

computes the doubling function.

Definition 3.3. Let final-value(.) be the partial function mapping Scheme ⟨expression⟩'s to their final values, if any. That is,

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

and domain (final-value(.)) is the set of M that converge. Define output(M) of M to be the printed form, if any, of final-value(M). That is,

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

[3]There is no loss of generality in this restriction, because Scheme without `call/cc` can simulate Scheme with that feature. We'll discuss this more fully in Part II of these Notes.

In working with partial functions, it's useful to adopt the convention that if f, g are partial functions, then 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. With this convention, we can say that

$$F \text{ computes } f \quad \text{iff} \quad \text{output}(\texttt{(}F\ \texttt{'}S_1 \ldots \texttt{'}S_n\texttt{)}) = f(S_1, \ldots, S_n). \tag{6}$$

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

Problem 2. Explain why output(.) 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 3.4. *The computable functions of one argument are closed under composition.*

Proof. If F computes f and G computes g, then we claim that

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

computes $f \circ g$.

It's worth working through the definitions at least one time to justify this kind of claim.

Since G computes g, we have by definition of "computes" that $\texttt{(}G\ \texttt{'}S\texttt{)} \downarrow \text{unprint}(g(S))$ for any S-expression S. By definition of convergence, this means that (after garbage collection):

$$\texttt{(}G\ \texttt{'}S\texttt{)} \xrightarrow{*} \text{unprint}(g(S)). \tag{7}$$

Likewise, since F computes f, we have

$$\texttt{(}F\ \texttt{'}g(S)\texttt{)} \xrightarrow{*} \text{unprint}(f(g(S))). \tag{8}$$

Since G computes g in an empty environment, it will still do so in any environment. (We leave it to the reader to confirm this property of the Substitution Model.) In particular, suppose

$$F \downarrow \texttt{(letrec (}B_F\texttt{)}\ V_F\texttt{)}. \tag{9}$$

So by Substitution Model rules, we can conclude that

$$\begin{aligned}
\texttt{(}F\ \texttt{(}G\ \texttt{'}S\texttt{))} &\xrightarrow{*} \texttt{(letrec (}B_F\texttt{) (}V_F\ \texttt{(}G\ \texttt{'}S\texttt{)))} && \text{(by (9))}\\
&\xrightarrow{*} \texttt{(letrec (}B_F\texttt{) (}V_F\ \text{unprint}(g(S))\texttt{))} && \text{(by (7))}\\
&\xrightarrow{*} \text{unprint}(f(g(S))) && \text{(by (9) and (8)).}
\end{aligned} \tag{10}$$

Therefore,

$$\begin{aligned}
\texttt{(}C\ \texttt{'}S\texttt{)} &\xrightarrow{*} \texttt{(}F\ \texttt{(}G\ \texttt{'}S\texttt{))} && \text{(by Submodel rules)}\\
&\xrightarrow{*} \text{unprint}(f(g(S))) && \text{(by (10))}
\end{aligned}$$

as required.

□

Problem. 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. 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.

Problem 3. Note that `(lambda (x) x)` computes the identity function on S-expressions. Give an example of another Scheme expression, F, such that F computes the identity function on S-expressions, but F is not observationally equivalent to `(lambda (x) x)`.

Problem. Suppose we applied Definition 6 allowing expressions in Scheme with `call/cc`, with the convention that aborted values print the same as returned values. That is, print(`(abort` V`)`)::= print(V) for each syntactic value, V. Give an example of an expression, G that uses `call/cc` to compute a function, g, and another expression, F, without `call/cc` that computes a function, f, such that

`(lambda (x) (`F `(`G `x)))`

does *not* compute $f \circ g$.

Definition 3.5. A set, $\mathcal{S}$, of S-expressions is Scheme *decidable* iff its membership function is Scheme computable, *i.e.*, 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 $\langle$string$\rangle$ of strings.

Problem 4. Explain why the set, $\langle$expression$\rangle$, of Scheme expressions, and the set of *closed* $\langle$expression$\rangle$'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 $\mathcal{S}$, *i.e.*, the set of S-expressions *not* in $\mathcal{S}$.

Lemma 3.6. *If $\mathcal{S}$ is Scheme decidable, then so is $\overline{\mathcal{S}}$.*

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

Note that this proof implicitly appeals to Proposition 3.1 in the assuming that

$$\text{final-value}(\texttt{(}F\ M\texttt{)}) = \texttt{\#t} \quad \text{implies} \quad \text{final-value}(\texttt{(not (}F\ M\texttt{))}) = \texttt{\#f}.$$

In general, if $M\downarrow$, then Proposition 3.1 implies that $M \equiv \text{final-value}(M)$. So it's OK to replace M by final-value(M) in an evaluation. In particular, since `(`$F\ M$`)` $\downarrow$ `#t`, we can conclude that `(not (`$F\ M$`))` $\downarrow$ `(not #t)`.

Problem 5. 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 3.7. 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[4]

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

Problem 6. Prove that if f is a Scheme computable total function, and $\mathcal{S}$ is decidable, then so is $f^{-1}(\mathcal{S})$.

Definition 3.8. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be sets of S-expressions. We say that $\mathcal{S}_1$ is *many-one reducible*[5] to $\mathcal{S}_2$, in symbols,

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

iff there is a total computable 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 $\mathcal{S}_2$.

The "$\leq$" notation for many-one reducibility highlights the fact that this relation is transitive. This follows because if 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 $\mathcal{S}_3$. So we could rephrase the result of Problem 6 as saying that *decidability inherits downward* under $\leq_{\text{m}}$.

A function, f, that many-one reduces $\mathcal{S}_1$ to $\mathcal{S}_2$ can be thought of as a way to translate a membership question about $\mathcal{S}_1$ into one about $\mathcal{S}_2$. Namely, it follows from the definition that f many-one reduces $\mathcal{S}_1$ to $\mathcal{S}_2$ is equivalent to the condition that

$$S \in \mathcal{S}_1 \quad \text{iff} \quad f(S) \in \mathcal{S}_2,$$

for all S-expressions, S.

[4]If f is a partial function from a set, $\mathcal{A}$ to a set $\mathcal{B}$, then we define domain$\,(f)$, the *domain* of f, to be the set $\{a \in \mathcal{A} \mid f(a) \text{ is defined}\}$. So domain$\,(f)$ is a subset of $\mathcal{A}$ and equals $\mathcal{A}$ iff f is a total function. Note that in other settings, what we call the domain of f is called the *support* of f, and $\mathcal{A}$ is called the domain. In these Notes we follow the standard usage in Computability Theory, letting "domain" be a synonym for "support."

[5]Sipser calls $\leq_{\text{m}}$ "map reducibility."

Definition 3.9. A set $\mathcal{S}$ of S-expressions is *half-decidable* iff there is a Scheme computable partial function whose domain is $\mathcal{S}$. A Scheme expression that computes such a function is called a *half-decider* for $\mathcal{S}$.

Half-decidable sets are usually called *recursively enumerable* (r.e.) sets, and sometimes are also called *recognizable* sets.

Lemma 3.10. *If a set of S-expressions is decidable, then it and its complement are also half-decidable.*

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

$$\texttt{(lambda (x) (or (}D\texttt{ x) }\Omega_0\texttt{))}$$

is a half-decider for $\mathcal{D}$, where Ω_0 is any closed expression such that $\Omega_0\uparrow$. For example, let

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

By Lemma 3.6, $\overline{\mathcal{D}}$ is also decidable, and so is also half-decidable. □

The converse of Lemma 3.10 also holds: if a set and its complement are both half-decidable, then the set is wholly decidable. We've chosen the name "half-decidable" to emphasize this fact. We'll postpone the proof of this converse to Part II.

Problem 7. **(a)** Prove that if $\mathcal{S}_1$ and $\mathcal{S}_2$ are half-decidable, 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 half-decidable, then so is $f^{-1}(\mathcal{S})$.

Note that as a special case of Problem 7(b), it follows that also *half*-decidability inherits downward under $\leq_{\mathrm{m}}$.

Problem 8. Two sets are said to be *almost equal* iff there are only finitely many elements in one but not the other. Show that if $\mathcal{S}_1$ is half-decidable and $\mathcal{S}_2$ is a set of S-expressions which is almost equal to $\mathcal{S}_1$, then $\mathcal{S}_2$ is also half-decidable.

Definition 3.11. 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 $f(0), f(1), \ldots$. This is the reason for the phrase "recursively enumerable."

Lemma 3.12. *Every computably countable set is half-decidable.*

Proof. Let $\mathcal{S}$ be computably countable, so $\mathcal{S} = f(\mathbb{N})$ for some f as in Definition 3.11. Then a procedure to half-decide 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 half-decider 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 3.12, allowing us to conclude that a nonempty set is computably countable iff it is half-decidable. For now, it's informative to solve the following problems without assuming this fact.

Problem 9. 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 10. **(a)** Prove that the set of $\langle$string$\rangle$'s is computably countable.

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

Problem 11. 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

$$\text{output}(\texttt{(}L\ \texttt{'}S\texttt{)}) = \texttt{(}S\ \texttt{'}S\texttt{)} \tag{11}$$

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 (11) above yields

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

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 12. **(a)** Check that this last P is self-reproducing by evaluating it in MIT Scheme.[6]

(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 (11).

Problem 13. An expression D is *doubly self-reproducing* iff $\text{output}(D) = \texttt{(}D\ \ D\texttt{)}$. 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[7], 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, $L(\text{factorial})$, of this free variable, namely, L is defined by

[6]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`.

[7]The way the Y operator works has also been used as a basis for optimizations in a Scheme compiler: INSERT CITATION to `jinx` paper.

```
        (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`:

`factorial` $\equiv$ `(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:

`(Y l)` $\equiv$ `(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

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

so

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

Then,

`(`$M_{\texttt{l}}$ $M_{\texttt{l}}$`)` $\equiv$ `(lambda (z) ((l (`$M_{\texttt{l}}$ $M_{\texttt{l}}$`)) 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 14. Evaluate `((Y L) 3)` in Scheme and in the Substitution Model interpreter, using the definitions `Y` and `L` above.

Problem 15. 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 16. Adapt the definition of `Y` so it works for multi-argument fixed points, namely, so

`(Y f)` $\equiv$ `(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, and define

$$\text{Halts} ::= \{M \mid M \text{ is a closed Scheme } \langle\text{expression}\rangle \text{ and } M\!\downarrow\}\,.$$

We remark that Halts is *half*-decidable: 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 combined with a decider for closed ⟨expression⟩'s would provide a half-decider for Halts. We'll explore this further in Part II.

By Lemma 3.10, this remark implies that showing that Halts is undecidable is equivalent to showing that its complement is not even half-decidable.

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

We'll prove this indirectly, by first proving:

Lemma 5.2. *The set*

$$\text{Self-div} ::= \{M \mid M \textit{ is not a closed } \langle\text{expression}\rangle \textit{ or } \texttt{(}M\ \ \texttt{'}M\texttt{)}\!\uparrow\}$$

is not half-decidable.

Assuming Lemma 5.2 for the moment, we can easily prove The Halting Theorem:

Proof. [of Theorem 5.1] By definition,

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

for all S-expressions, S. It follows that Self-div $\leq_{\text{m}}$ $\overline{\text{Halts}}$ under the mapping, f, where $f(S) ::= \texttt{(}S\ \ \texttt{'}S\texttt{)}$. The expression

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

computes f, so by Problem 7(b), if $\overline{\text{Halts}}$ was half-decidable, then Self-div would be as well, contradicting Lemma 5.2. □

The definition of Self-div was chosen just to make the proof of Lemma 5.2 almost immediate:

Proof. To prove Lemma 5.2, let $\mathcal{H}$ be a half-decidable set, and H a half-decider for it. So by definition, of H,

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

for all S-expressions, S.

Now since H is a closed Scheme expression,

$$H \in \text{Self-div} \quad \text{iff} \quad \texttt{(}H \ \texttt{'}H\texttt{)}\uparrow,$$

by definition of Self-div. So

$$H \in \mathcal{H} \quad \text{iff} \quad \texttt{(}H \ \texttt{'}H\texttt{)}\downarrow \quad \text{iff} \quad H \notin \text{Self-div}.$$

In other words, so $\mathcal{H} \neq$ Self-div because H is in one and not the other. Since Self-div is not equal to any half-decidable set, it must not be half-decidable.

□

Any closed Scheme expression, M, is by definition a half-decider for a set $\mathcal{H}_M$:

Definition 5.3. If M is a closed Scheme expression, then

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

If M is not a closed Scheme expression, then $\mathcal{H}_M ::= \emptyset$.

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 5.2 shows that finding a witness that $\mathcal{H}_M \neq$ Self-div 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 $P \neq \mathcal{H}_M$:

Definition 5.4. 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 half-decidable, since it differs from every half-decidable set. So now we can rephrase the conclusion that comes out of the proof of Lemma 5.2: the set Self-div is productive with witness function equal to the identity function on S-expressions. Of course Self-div 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 Self-div $\leq_{\text{m}}$ $\overline{\text{Halts}}$ along with:

Lemma 5.5. *Productivity inherits upward under $\leq_{\text{m}}$.*

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

Problem 17. Prove Lemma 5.5.

Problem 18. Prove that every half-decidable set is $\leq_{\text{m}}$ Halts.

Problem 19. **(a)** Prove that $A \leq_{\text{m}} B$ iff $\overline{A} \leq_{\text{m}} \overline{B}$.

(b) Conclude that Halts and $\overline{\text{Halts}}$ are incomparable under $\leq_{\text{m}}$.

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,

$$\mathcal{E} ::= \{\texttt{(}M\ \texttt{=}\ N\texttt{)} \mid M, N \text{ are } \langle\text{expression}\rangle\text{'s and } M \equiv N\}\,.$$

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 simple requirement that proofs be mechanically checkable now implies:

Theorem 6.2. *The set of assertions provable using any given proof system is half-decidable.*

Proof. The condition that a proof be checkable "without understanding" can be understood technically as meaning that the proofs are a *decidable* set, $\mathcal{D}$, of S-expressions.

In addition, there must be a mechanical way to determine what assertion the proof proves. This is called the *upshot* of the proof. So technically, this means there is a computable function upshot : $\langle\text{s-expr}\rangle \to \langle\text{s-expr}\rangle$ that maps any proof to its upshot.

Now we have a way to enumerate the provable assertions: enumerate the S-expressions, returning the upshot of those expressions that are proofs, and returning some fixed provable assertion otherwise. To be precise, let A be some provable assertion, let proof? be a decider for the set of proofs, *upshot* be an expression that computes the upshot function, and C be an expression that computes a counting function for the set of all S-expressions (*cf.*, Problem 11). Then the following expression counts the provable assertions:

```
(lambda (n) (if (proof? (C n)) (upshot (C n)) 'A)).
```

So the provable assertions are computable countable, and therefore, by Lemma 3.12, are half-decidable as claimed.

□

Problem 20. Sketch how to write a Scheme program computing a decider for proofs in the arithmetic equation proof system of the Notes. What is the upshot function for these proofs?

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 half-decidable subset of the valid assertions. Now, if we show that the valid assertions are not half-decidable, 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 half-decidable. 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

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

for all Scheme expressions, M.

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

$$f(M) ::= \begin{cases} (M \texttt{ = } \Omega_0), & \text{if } M \text{ is a closed Scheme } \langle\text{expression}\rangle\,, \\ (\Omega_0 \texttt{ = } \Omega_0), & \text{otherwise.} \end{cases}$$

Clearly f is computable, because it is computed by the expression

```
(lambda (m) (if (D m) (list m '= 'Ω₀) '(Ω₀ = Ω₀))
```

where D is a decider for the closed Scheme expressions (cf. Problem 4). It follows that

$$\begin{aligned} M \in \overline{\text{Halts}} \quad & \text{iff} \quad M \text{ is not a closed } \langle\text{expression}\rangle\,, \text{ or } M\uparrow \\ & \text{iff} \quad M \text{ is not a closed } \langle\text{expression}\rangle\,, \text{ or } M \equiv \Omega_0 \\ & \text{iff} \quad f(M) \in \mathcal{E}. \end{aligned}$$

Hence $\overline{\text{Halts}} \leq_{\text{m}} \mathcal{E}$. Since we know $\overline{\text{Halts}}$ is productive, we have by Lemma 5.5 that $\mathcal{E}$ is also productive.

□

A consequence of Lemma 6.3 is that given any proof system for Scheme equivalences, specifically, given a half-decider for the equations provable in the system, we can apply the witness function for $\mathcal{E}$ to the half-decider 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 remark: the proof of Lemma 6.3 also demonstrates that $\overline{\text{Halts}} \leq_{\text{m}} \mathcal{E}_0$ where $\mathcal{E}_0$ is the set of valid equivalences of the form $\texttt{(}M = \Omega_0\texttt{)}$:

Definition 6.4.

$$\mathcal{E}_0 ::= \{\texttt{(}M \;\texttt{=}\; \Omega_0\texttt{)} \mid M \text{ is an } \langle\text{expression}\rangle \text{ and } M \equiv \Omega_0\}\,.$$

So we have

Corollary 6.5. *$\mathcal{E}_0$ 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 half-decidable. 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 "Perverse" 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,

$$\texttt{(}T\ \texttt{'(}S\ \texttt{'}S\texttt{))} \downarrow \texttt{\#t} \quad \text{iff} \quad \texttt{(}T\ \texttt{'(}P\ \texttt{'}S\texttt{))} \downarrow \texttt{\#f}. \tag{12}$$

Now let S be P and then M be $\texttt{(}P\ \texttt{'}P\texttt{))}$ in (12). This yields

$$\texttt{(}T\ \texttt{'}M\texttt{)} \downarrow \texttt{\#t} \quad \text{iff} \quad \texttt{(}T\ \texttt{'}M\texttt{)} \downarrow \texttt{\#f}. \tag{13}$$

But (13) can only hold if $\texttt{(}T\ \texttt{'}M\texttt{)}\uparrow$, 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 submodel-invariant, nonempty complement. If $\overline{\text{Halts}} \subseteq \mathcal{S}$, then $\mathcal{S}$ is not half-decidable.*

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 21. Prove Scott's Rice's Second Theorem 7.4.

Problem 22. Strengthen Theorem 7.4 so that it directly implies that neither the set $\mathcal{G}_{\text{store}}$ above nor its complement are half-decidable.