

The Substitution Model

Modified from a handout in Dartmouth CS18

Original author unknown

Modified by Allyn Dimock

September 10, 2004

The **substitution model** is a conceptual framework we can use to determine the value or "meaning" of a Scheme expression. It gives us a set of simple algebraic rules we can use to deduce what the result of evaluating an expression should be.

The substitution model is not intended to directly simulate the details of how the computer actually evaluates Scheme expressions – the computer must keep track of many details that are not important to our understanding of the process. So, the substitution model is an *abstract* model of how evaluation works. It is intended to give us a method of reasoning logically about programs.

1 Representation of Values

An **expression** is a primitive object, a lambda abstraction, or a combination of expressions.

Most expressions are too complex to be evaluated all at once. Therefore, we generally break down the evaluation process into a series of discrete steps, which transform complex expressions into simpler expressions having equivalent meaning. The process of **evaluating** an expression means to repeatedly apply these simplifying rules, until you obtain an expression that cannot be simplified any further. This resulting expression is said to be the **value** of the original expression.

Before we can model the process of evaluating a Scheme expression, we need to develop some notation that will allow us to differentiate between

parts of an expression which have already been evaluated, and those which are remaining to be considered. For the purposes of this handout, we will write { curly brackets } around *values* in order to distinguish them from objects which have not yet been evaluated.

1.1 Primitive Data

Primitive values include objects such as numbers and truth values. The value of numeric expressions such as

4
-3.1
2.71828
 $3 + 4i$
5/16

are the numbers that they name. We denote this by placing the number in braces, as described above. For instance, we will denote the value of the expression 5 as {5}. That is not what the computer prints out when you evaluate 5, we are simply using this notation to indicate that the expression 5 has been evaluated, and does not need to be reduced further.

The value of **#t** is {#t}, similarly for **#f**.

1.2 Primitive Procedures

The Scheme environment provides a set of **primitive procedures**, sometimes called **primitive operators**, that form the building blocks of more complex expressions. This category includes basic operations like addition, multiplication, and comparison of values. In a Scheme program, we generally access primitive procedures by writing an identifier that is associated with the primitive procedure, such as + for addition, * for multiplication, and = for comparison.

The value of a primitive procedure is the sequence of instructions that actually performs the operation. Rather than writing this out longhand, we will simply write a brief textual description of the procedure in curly braces, e.g.,

- The value of the addition operator is written {add}
- The value of the multiplication operator is written {mult}

- The value of the equality-testing operator is written `{equals}`
- et cetera

When we write `{mult}`, for example, you should take this as shorthand for “the code to multiply a sequence of values”.

1.3 Compound Procedures

Compound procedures are created by evaluating a lambda expression, which takes a list of parameters and a sequence of body expressions, and creates a procedure object. We will use the symbol `proc` to indicate compound procedures which are created in this way, and will (usually) include the arguments and body of the procedure. For example, the value obtained by evaluating the expression

```
(lambda (x y) (+ x y))
```

is represented by:

```
{proc (x y) (+ x y)}
```

We will sometimes omit the body and simply write an ellipsis (...), if it is clear which compound procedure object we are referring to from context.

2 Rules of Evaluation

Primitive expressions, such as numbers, operators, etc., evaluate directly to their associated values. The interesting question, therefore, is how to evaluate **combinations**.

Our first type of combination is

```
(operator operand1 ... operandn)
```

there the operator and the operands are expressions. Note that there can be 0 or more operands. For example

```
(+ 5 (* 3 2))
```

is a combination. The individual expressions in the combination are called **subexpressions**. In the latter example, the subexpressions are

```
+           ;; the name of a primitive procedure
5           ;; a number
(* 3 2)    ;; a combination
```

Notice that the `(* 3 2)` is itself a combination, with subexpressions `*`, `3`, and `2`.

A combination is evaluated by recursively applying two basic reduction rules (described below). The resulting value can then be substituted for the original expression.

The two basic rules are:

EVAL: To evaluate a combination (other than a special form), **evaluate** its subexpressions *in any order*, then **apply** the procedure that is the value of the left-most subexpression to the values resulting from the remaining subexpressions.

APPLY: To apply a non-primitive procedure to a set of arguments, **evaluate** the body of the procedure, with each formal parameter replaced by the corresponding argument value.

Notice that EVAL and APPLY are mutually recursive (that is, EVAL uses APPLY, and APPLY uses EVAL). As a result, the process alternates between **evaluation** and **application**, which continue recursively until the entire expression has been reduced to primitive objects of one variety or another.¹

The rules for primitive expressions are:

EVAL (primitive data)

The value of a primitive expression, such as a number or a truth value, is the value that it names. For instance, the value of `3.142` is `{3.142}`.

EVAL (variable)

The value of an variable (or **identifier**) is the object that is associated with that variable. The Scheme evaluator maintains a table (called an **environment**) that maps (or **binds**) variable names to values, and

¹Or the evaluation gets **stuck** because of some expression that does not make sense such as `(1 + 1) : 1` does not evaluate to a procedure.

the value for an variable is obtained by looking up the variable name in the table. It is an error if there is no entry in the table for the variable name.

There are various ways to associate values with names in Scheme, the most basic of which is the **define** special form. The Scheme evaluator provides several default name bindings, for the built-in primitive operations.

EVAL (primitive operator)

The value of a built-in primitive operator is a sequence of machine instructions to carry out the specified operation. We denote this, as described above, by writing a descriptive name for the operation, such as {add} for addition.

APPLY (primitive operator)

To apply a built-in procedure to its evaluated arguments, the Scheme evaluator runs the code for the procedure on those arguments. In the substitution model, we will simply supply the resulting value directly.

EVAL (value)

Any primitive value evaluates to itself. So, if we are called upon to evaluate some {value} that resulted from some other evaluation, we will obtain the same {value}.

Notice that apart from the built-in operators, the primitive objects have no **apply** rules. It is an error to attempt to apply a value other than a procedure.

When we model the evaluation of an expression using these rules, we will use two different notations. Expressions that are being **evaluated** (using the EVAL rules) will be written with (parentheses). An fully-evaluated expression that denotes the **application** of a procedure will be written with [square brackets] instead of the standard parentheses.

3 A Simple Example

Consider the evaluation of the following combination:

((lambda (x) (* x x)) 5)

Simple inspection shows that this should result in squaring the number 5. Let's walk through the evaluation a step at a time, according to the rules. Note that the step numbers are simply for reference, and are not part of the substitution model. We underline the expression that we are about to evaluate.

Step	Expression	Rule
1.	((lambda (x) (* x x)) <u>5</u>)	Evaluate lambda subexpression
2.	(<u>{proc (x) (* x x)}</u> {5})	Evaluate 5
3.	{ <u>proc (x) (* x x)</u> {5}}	Apply compound procedure to {5}
4.	(* <u>{5}</u> {5})	Evaluate *
5.	{ <u>mult</u> {5} {5}}	Apply primitive {mult}
6.	{25}	Evaluation complete

Begin with step (1). Since this is the evaluation of a combination, the EVAL rule says we must first evaluate the subexpressions. The value obtained by evaluating the lambda subexpression is a procedure object, denoted: {proc (x) (* x x)}. At step (2), we determine that the value of the expression 5 is {5}. Thus, after this first stage of evaluation, our model reduces the initial expression to the application of a procedure object to a value, illustrated at the beginning of step (3).

The APPLY rule for a compound procedure says that we should take the body of the procedure, which is (* x x), and **substitute** the corresponding arguments everywhere the parameters appear. The procedure has only one parameter, x, and the argument in this case is {5}. If we replace each x with {5}, we have the next expression (* {5} {5})

Now we have another evaluation problem to solve. Since this is a combination, the EVAL rule says to evaluate the subexpressions. By the EVAL rule for identifiers, * evaluates to {mult}, which is a primitive operator. By the EVAL rule for values, the value {5} evaluates to itself. So, at step (5) we now have the application of a primitive multiplication operator to a pair of operands.

The APPLY rule for primitive operators says to apply the operation to the arguments directly, which results in the value {25}. At this point, we have a primitive value, which cannot be reduced any further, so we are finished with evaluation.

Notice how the process involves the alternation of evaluation and application.

4 Special Forms

A special form is any phrase in the Scheme language that is parenthesized but is not a combination of an operator and operands.

Scheme has one special form for a definition. A definition takes a **name** and an expression and updates the environment to bind the name to the value of the expression.

Scheme has seventeen keywords that are used to introduce special forms for expressions.

Purpose	keywords
Procedure abstraction	<code>lambda</code>
Conditional evaluation	<code>cond</code> <code>if</code> <code>and</code> <code>or</code> <code>case</code>
Local naming	<code>let</code> <code>let*</code> <code>letrec</code>
Structured data/Programs as data	<code>quote</code> <code>quasiquote</code>
Mutable data	<code>set!</code> <code>begin</code>
Delaying evaluation	<code>delay</code>
Make FORTRAN programmers happy	<code>do</code>
Use macros	<code>let-syntax</code> <code>letrec-syntax</code>

Of these keywords for expressions, the textbook uses `lambda` `if` `cond` `and` `or` `let` starting in Chapter 1. It uses `quote` starting in Chapter 2. It uses `set!` and `begin` starting in Chapter 3. Once `set!` is introduced, the substitution model becomes inadequate for describing program behavior and a new model is introduced. Something like `delay` is introduced in Chapter 3. The other special forms are not used in S.I.C.P.

4.1 define

A definition can not be used as part of a combination, so it is not an expression. The rule for evaluating

`(define name expr)`

is:

1. evaluate `expr`
2. add a binding to the environment for `name` \mapsto value of `expr`

There is a top-level environment consisting of values for names of primitive operations and standard library routines. When `{proc (x1 ... xn)`

`body}` is being evaluated, the environment is extended temporarily with any bindings at the beginning of `body`.

The rule for **evaluating a variable** in an expression: If the variable has not already been substituted for, then it should be the result of some `define` and be visible in the environment. Replace the variable with the value that it was defined to have.

4.2 cond

A conditional has the form

```
(cond (predicate1 action1)
      (predicate2 action2)
      ...
      (predicaten actionn))
```

To evaluate a conditional

- Evaluate `predicate1`.
- If `predicate1` evaluates to `#f` then reduce the problem of evaluating to evaluate

```
(cond (predicate2 action2)
      ...
      (predicaten actionn))
```

instead.

- If `predicate1` evaluates to anything other than `#f` then the value of the `cond` expression is the value of `action1`.
- the last predicate in a `cond` can be replaced with the keyword `else`. The value of `(cond (else action))` is the value of `action`.
- As you strip away predicate / action pairs as above, you may come to the case where all that is left to evaluate is `(cond)`. The value of `(cond)` is undefined. In DrScheme, value of `(cond)` can not be used in most expressions. In DrScheme, the value of `(cond)` does not print in the read - eval - print loop of the scheme interpreter. If you type in `(cond)` you just see the next prompt. If, in DrScheme you type in `(print (cond))` you will see `#<void>` printed out.

4.3 Syntactic Sugar: `if` and `let`

The term “syntactic sugar” refers to forms in the programming language that make your programming experience sweeter: Removing them from the language would not make a difference since they can be constructed from other forms already in the language, but they certainly are convenient.

For syntactic sugar, such as `if` or `let` it is possible to translate back to the more general form and use the rules for the more general form.

Syntactic sugar that we have seen includes

```
(if predicate consequent alternate)
```

for

```
(cond (predicate consequent)
      (else alternate))
```

```
(define (name x1 ... xn) exp)
```

for

```
(define name (lambda (x1 ... xn) exp))
```

```
(let ((x1 exp1)
      (x2 exp2)
      ...
      (xn expn))
  body)
```

for

```
((lambda (x1 x2 ... xn) body)
 exp1
 exp2
 ...
 expn)
```

It is also possible to make new rules directly for the sugared expressions. For instance a rule for `let` would say to evaluate `exp1 ... expn` in any order,

then substitute the values for `x1`, ... `xn` in `exp`. Note that this rule never produces a `proc` value, whereas the unsugared form would produce a `proc` value from the `lambda` as part of evaluating the combination.

4.4 More special forms

and

This is “short-circuit and” like `&&` in C. If it only took two expressions we could treat

```
(and exp1 exp2)
```

as syntactic sugar for

```
(let ((p exp1))
    (if p exp2 #f))
```

but the actual evaluation rule is a bit more complicated since `and` takes 0 or more expressions.

To evaluate

```
(and exp1 exp2 ... expn)
```

start with the value `{#t}` and start evaluating expressions in left-to-right order. If some expression evaluates to `{#f}` then the value for the entire `and` special form is `{#f}` and none of the subexpressions after the expression yielding `{#f}` are evaluated. If no expression evaluates to `{#f}` then the value of the `and` special form is the last value calculated: that is to say the value for `expn`, except in the wierd case of `(and)` with no expressions where the value of `(and)` is `{#t}`.

or

This is “short-circuit or” like `||` in C.

```
(or exp1 exp2)
```

as syntactic sugar for

```
(let ((p exp1))
    (if p p exp2))
```

but `or`, like `and` takes 0 or more expressions.

To evaluate

```
(or exp1 exp2 ... expn)
```

start with the value `{#f}` and start evaluating expressions in left-to-right order. If some expression evaluates to *anything other than* `{#f}` then the value for the entire `or` special form is that value and none of the following subexpressions are evaluated. If all expressions evaluate to `{#f}` then the value of the `or` special form is `{#f}`, even in the wierd case of `(or)` with no expressions.

begin

The `begin` special form has the syntax

```
(begin exp1 exp2 ... expn)
```

where there is at least one expression following the `begin` keyword.

The expressions are evaluated from left to right, but the values of all but the last expression are thrown away, and the value of the last expression is the value of the entire `begin` special form.

We will see `begin` used in contexts like:

```
(begin (print x) (newline) x)
```

which evaluates to the value of `x` but first prints the value of `x` and starts a new line on the standard output.