By Joshua Cantrell
jjc@cory.berkeley.edu

Normal Forms & Special Forms[1]

What are forms?

Forms are groups of characters that you can type into the interpreter to make acceptable expressions. You can make all possible Scheme expressions by producing all permutations of the forms. Scheme has a limited number of forms:

Boolean -- **#T** for true, and **#F** for false.

Number -- This can be any type of number:

- integer (**12**, **-134**, **+14**)
- rational (**4/5**, **-24/7**, **+2/3**)
- decimal (**1.23**, **-24.1**, **+41.32**).

Symbol -- These are used as *identifiers* and are composed of a sequence of letters, digits, and extended alphabetic characters from the set **+-.*/<=>!?:$%_&~^**. Possibilities are **equal?**, **1+**, **set!**. It's also good to know that Scheme is case insensitive, so **EQUAL?** and **equal?** are the same.

List -- These are formed from groups of forms surrounded by parentheses. A list is represented by **(**F_1 F_2 **...** F_n**)**, where F_i can be any form and $n \geq 0$. This means you can have lists within lists and empty lists, **()**.

Evaluating Normal Forms

When you type in a form at the Scheme interpreter prompt, it's evaluated. Normal forms are always evaluated in the same way.

Boolean -- **#T** evaluates to true and **#F** evaluates to false.

Number -- These always evaluate to the numbers they represent.

Symbol -- Symbols are known as *identifiers* because they are bound to (defined to be) values and evaluated to what they were previous bound. If a symbol is evaluated, but has not been bound to a value, an error is produced.

List -- The normal form of list evaluation follows the following steps:

1. Given the form **(**F_1 F_2 **...** F_n**)**, F_1 through F_n must all be evaluated.
2. F_1 must evaluate to a procedure, otherwise an error is produced. This procedure (operator) is called using the results of F_2 through F_n as it's actual parameters (operands).
3. The result of calling the procedure becomes the result of evaluating the list.

---

[1] Special thanks goes to Professor Hilfinger whose Fall 1995 CS61A notes I used as a basis for the information found in this document.

By Joshua Cantrell
jjc@cory.berkeley.edu

Evaluating Special Forms

Special forms are those forms which act significantly different from the previously described normal forms, so they must be identified before evaluation and treated with care. Most special forms are of the *list* form and can be identified by the first symbol in the list. One of the exceptions is the quote special form designated by the single quote, ‘.

Quoted Expression -- A quoted expression is not evaluated. An expression can be quoted in two ways. Using the list form and the single quote form.

‘a, **‘130**, **‘(a b c)**, **‘((1 2 3) (a b c))**
(quote a), **(quote 130)**, **(quote (a b c))**,
(quote ((1 2 3) (a b c)))

Define -- The form used to bind a symbol (identifier) to a value. It takes on the form **(define** <*symbol*> <*expression*>**)**, where the <*symbol*> symbol is bound to the result of evaluating the <*expression*> form.

Procedure Define -- The form used to bind a symbol (identifier) to a procedure. It takes on the form **(define (**<*symbol*> <arg_1> ... <arg_n>**)** <exp_1> ... <exp_n>**)**, where a procedure with the formal parameters (symbols), <arg_1> through <arg_n>, and the body formed by the <exp_1> through <exp_n> forms, is bound to the <*symbol*> symbol. Note that the procedure’s body expressions are not evaluated until the procedure is called.

If -- The form used to make a single condition statement. It takes on the form **(if** <*predicate*> <*consequent*> <*alternate*>**)**, where the <*predicate*> form evaluates to a Boolean value which determines which of the <*consequent*> or <*alternate*> forms is finally evaluated. If <*predicate*> evaluates to true, <*consequent*> is evaluated. Otherwise if <*predicate*> evaluates to false, <*alternate*> is evaluated.

Conditional -- The form used to make a multiple condition statement. It takes on the form

(cond (<$predicate_1$> <$consequent_{1,1}$> ... <$consequent_{1,i}$>**)**
(<$predicate_2$> <$consequent_{2,1}$> ... <$consequent_{2,j}$>**)**
...
(<$predicate_n$> <$consequent_{n,1}$> ... <$consequent_{m,k}$>**)**
(else <$consequent_{n+1,1}$> ... <$consequent_{n+1,l}$>**)**,

where each <*predicate*> form determines whether its <*consequent*> forms are evaluated. Each <*predicate*> is evaluated in order until a true result is found. No further <*predicate*> forms are evaluated past that point, and the <*predicate*> whose result was true has its <*consequent*> forms evaluated. At the end of the form is *else* which evaluates its <*consequent*> forms if all of the previous <*predicate*> forms resulted in false.

By Joshua Cantrell
jjc@cory.berkeley.edu

Or and *And* -- The forms used for logic expressions. They take on the forms:

(or <*expression*$_1$> ... <*expression*$_n$>**)** and

(and <*expression*$_1$> ... <*expression*$_n$>**)**,

where each <*expression*> is evaluated in order until a deciding case is found. For *or*, the first expression resulting in true automatically results in true without evaluation of further expressions. For *and*, the first expression resulting in false automatically results in false without evaluation of further expressions.

Lambda -- The form used to create procedures. This takes on the form

(lambda (<*arg*$_1$> ... <*arg*$_n$>**)** <*exp*$_1$> ... <*exp*$_n$>**)**, where a procedure, with the formal parameters (symbols), <*arg*$_1$> through <*arg*$_n$>, and the body formed by the <*exp*$_1$> through <*exp*$_n$> forms, is the result of evaluating the form.

Let -- The form used to define new levels of local variables. This takes on the form

(let ((<*var*$_1$> <*exp*$_1$>**)** ... **(**<*var*$_n$> <*exp*$_n$>**))** <*body*>**)**, where the local symbols, <*var*$_1$> through <*var*$_n$>, are bound to the results of evaluating their respective expressions, <*exp*$_1$> through <*exp*$_n$>, and then the expressions in <*body*> are evaluated.

Interpreting Special Form Results

Certain special forms can be substituted with an expression whose results from evaluation is equivalent to the special form's. Two of these are the *procedure define* and *let* special forms.

PROCEDURE DEFINE:

(define (<*symbol*> <*arg*$_1$> ... <*arg*$_n$>**)** <*body*>**)**

can be rewritten as:

(define <*symbol*> **(lambda (**<*arg*$_1$> ... <*arg*$_n$>**)** <*body*>**))**

LET:

(let ((<*var*$_1$> <*exp*$_1$>**)** ... **(**<*var*$_n$> <*exp*$_n$>**))** <*body*>**)**

can be rewritten as:

((lambda (<*var*$_1$> ... <*var*$_n$>**)** <*body*>**)** <*exp*$_1$> ... <*exp*$_n$>**)**