1 Macro Expansions in Scheme

As you’ve seen in previous courses, macros allow us to programmatically manipulate the source of a program. In this course, you are required to implement parts of the syntax in your compiler using macros. This will allow you to focus on implementing a small subset of core forms while still maintaining support of the full range of syntactic forms of the source language.

We will discuss the principles of the correct use of macros using the following examples.

1.1 Expand if

Let’s implement if expressions as a macro.

We will expand expressions of the following form into semantically-equivalent expressions that do not make use of if.

(if <test> <then> <else>)

1.1.1 First Attempt

(or (and <test> <then>) <else>)
This is not a good expansion since it is not semantically equivalent to the original expression. Consider the case where <then> is evaluated as #f. In this case, <else> would still be evaluated.

1.1.2 Second Attempt

(or (and <test> <then>) (and (not <test>) <else>))

This is still not a good expansion. In this case, <test> will be evaluate twice (double evaluation), therefore, any side effect contained in test will occur more than once.

1.1.3 Third Attempt

(let ((val-test <test>))
  (or (and val-test <then>)
      (and (not val-test) <else>)))

This is also not a good expansion, as introduces val-test into the scopes of <then> and <else> (no variable hygiene).

1.1.4 Correct Expansion

(let ((then (lambda () <then>))
     (else (lambda () <else>))
     (val-test <test>))
  ((or (and val-test then) else)))

1.2 Expand or Using if

Lets implement the or operator as syntactic sugar. Recall that in Scheme, an or expression returns either the first value which is not #f or the last expression. So

> (or #f 'foo 1)
foo
> (or #f #f)
#f

In this exercise we will expand expressions of the following form into semantically-equivalent expressions that do not make use of or.

(or <exp-1> <exp-2> ... <exp-k>)

1.2.1 First Attempt

(if <exp-1>
  <exp-1>
  [macro-expand (or <exp-2> ... <exp-k>)]
)
This is not a good expansion, as \(<\text{exp-1}>\) will be evaluated twice (which would cause problems if \(<\text{exp-1}>\) contains side effects).

1.2.2 Second Attempt

\[
\begin{align*}
\text{(let ((v \(<\text{exp-1}>\)))} \\
\text{ (if v} \\
\text{ v} \\
\text{ [macro-expand (or \(<\text{exp-2}>\) \ldots \(<\text{exp-k}>\)])])}
\end{align*}
\]

This is not a good expansion, as it introduces the variable \(v\) into the scope of the nested expansion (no \textbf{variable hygiene}). This would cause problems if some \(<\text{exp-i}>\) \((2 < i < k)\) contained references to \(v\).

1.2.3 Correct Expansion

Use function definitions to create separate scopes.

\[
\begin{align*}
\text{((lambda (val_1 rest)} \\
\text{ (if val_1 val_1 (rest)))} \\
\text{\(<\text{exp-1}>\) } \\
\text{ (lambda () [macro-expand (or \(<\text{exp-2}>\) \ldots \(<\text{exp-k}>\)])])}
\end{align*}
\]

Remark. Note that this expansion definition does not correctly handle \textbf{(or)} expressions (\textbf{or} with no arguments). To deal with this, the macro-expand should identify this case specifically and return \textbf{#f}.

1.3 Add foreach Loops to the Language

As an exercise, we will add Java-style foreach loops to the scheme language:

\[
\text{(foreach \(<\text{lst-expr}>\) \(<\text{body-expr}>\) )}
\]

which is equivalent to

\[
\text{for ((\text{type} <\text{var-name}> : \(<\text{lst-expr}>\) ) } \{ \\
\text{ <body-expr> }
\}
\]

The requirements from the expansion are:

- Variable hygiene
- Correctness (no issues with side effects)
- To return the void object
1.3.1 The Expansion

(let ((lst <list-expr>))
  (body <body-expr>))
(letrec ((loop (lambda (lst)
    (if (not (null? lst))
        (begin (body (car lst))
            (loop (cdr lst)))))))
  (loop lst))