Quasiquote

The argument of `quote` is a literal constant

\[
'(\text{if } (> \ a \ b) \ 3 \ 4)) \rightarrow (\text{if } (> \ a \ b) \ 3 \ 4))
\]

More flexible: allow “holes” in the literal, to be filled in with run-time computed values

quasiquote, or ` (backquote) allows this

- `,expr marks a hole, filled in with result of evaluating expr
- `@listexpr marks a hole, filled in with elements of list resulting from evaluating listexpr

\[
(\text{define } (\text{make-if } \text{test} \ \text{then} \ \text{else})
'(\text{if } ,\text{test} ,\text{then} ,\text{else}))
(\text{make-if } '>(\ a \ b) \ 3 \ 4)) \rightarrow (\text{if } (> \ a \ b) \ 3 \ 4))
\]

\[
(\text{define } (\text{make-call } \text{fn} \ \text{args})
'(,(\text{fn} ,@\text{args}))
(\text{make-call } '+ ' (3 4 5)) \rightarrow (+ 3 4 5))
\]

Very useful in many systems that build structured data, particularly program representations

Side-effect special forms

`set!`: rebind a variable to refer to a different value

\[
(\text{define } x 5)
(\text{set!} \ x ' (6 7))
x \rightarrow (6 7)
\]

\[
(\text{define } (\text{test lst})
(\text{set!} \ \text{lst} (\text{cons} 1 \ \text{lst}))
\text{lst})
(\text{test } '(2 3)) \rightarrow (1 2 3)
\]

Scheme’s design is more biased towards side-effecting style than ML’s

- all Scheme variables can be reassigned using `set!`
- mutation isn’t compartmentalized
- body of a function, arm of a cond, etc. is a series of expressions to evaluate
- all but last evaluated just for their side-effects
- Scheme has predefined non-recursive looping constructs

Side-effects on cons cells

`set-car!`, `set-cdr!`: rebind head, tail of cons cell

\[
(\text{define } c (\text{list} 5 6))
(\text{set-car!} \ c (\text{list} 3 4)))
\]

\[
(\text{define } c (\text{list} 3 4)))
\]

\[
(\text{set-cdr!} \ (\text{cdr} \ c) (\text{list} 7 8))
\]

\[
(\text{define } c (\text{list} 4 6 7 8)))
\]

Example

`append!!`: destructive append

\[
(\text{define } (\text{append}!! \ \text{lst1} \ \text{lst2})
(\text{cond} ((\text{null?} \ \text{lst1}) \ \text{lst2})
((\text{null?} \ (\text{cdr} \ \text{lst1}))
(\text{set-cdr!} \ \text{lst1} \ \text{lst2})
\text{lst1})
(\text{else} (\text{append}!! (\text{cdr} \ \text{lst1}) \ \text{lst2})))\))
\]

\[
(\text{define } \text{lst1} '(3 4))
(\text{define } \text{lst2} '(5 6 7))
(\text{define } \text{lst3} (\text{append}!! \ \text{lst1} \ \text{lst2})))
\text{lst3} \rightarrow (3 4 5 6 7)
\]

\[
(\text{define } \text{lst1} '(3 4))
(\text{define } \text{lst2} '(5 6 7))
(\text{define } \text{lst3} (\text{append}!! \ \text{lst1} \ \text{lst2})))
\text{lst3} \rightarrow (3 4 5 6 7)
\]

\[
(\text{define } \text{lst1} '(3 4))
(\text{define } \text{lst2} '(5 6 7))
(\text{define } \text{lst3} (\text{append}!! \ \text{lst1} \ \text{lst2})))
\text{lst3} \rightarrow (3 4 5 6 7)
\]

append!!: more efficient than append, but more complicated to use correctly in face of rampant sharing
First-class functions

Scheme supports first-class, lexically-nested, statically-scoped function values, just like ML

Translation between ML and Scheme

<table>
<thead>
<tr>
<th>ML</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fn pat =&gt; expr</code></td>
<td><code>(lambda (id₁ ... idₙ)</code></td>
</tr>
<tr>
<td><code>map f lst</code></td>
<td><code>(map f lst₁ ... lstₙ)</code></td>
</tr>
</tbody>
</table>

Scheme R5RS doesn’t have filter, fold, etc. predefined

Control constructs

Languages support mechanisms for controlling execution flow:

Basic methods:
- procedure call & return, potentially recursively
- conditional execution like `if, cond`

Advanced methods:
- looping (!)
- break, continue
- exception handling
- coroutines, threads
- ...

Continuations

Scheme supports all advanced control constructs with one notion: continuations

A continuation is a function that can be called (with a result value) to do “the rest of the program,” exiting the current task
- enables parameterizing a function by “what to do next,” “where to return to”
- enables having multiple return places, not just the one normal return, for different kinds of outcomes

Example, using normal functions as continuations:

```scheme
(find is-string? '(...)
(lambda[x] 'Yes ,x))
(lambda() 'No))
```

Current continuation

The normal return point is an implicit continuation:
- it takes the returned value and does the rest of the program

Scheme makes this continuation first-class upon request using `call-with-current-continuation (a.k.a. call/cc)`

`call/cc` takes an argument function of one argument, P, and invokes P passing the current continuation, K, as P’s argument
- if P returns V normally, call/cc returns V
- if P invokes K, passing one argument value, V, P quits and call/cc returns V

Example: computing products with an early exit

```scheme
(define (prod lst)
(call/cc (lambda (exit) ;; exit: reified context
  (foldl
   (lambda (x accum)
     (if (zero? x)
       (exit 0) ;; break out of loop, return 0
       (* x accum) ;; continue multiplying
     )
   )
   1 lst))))
```
Another example: threads

Task: implement a lightweight non-preemptive thread package

API:
- \texttt{(fork \ f)}: creates a new (initially suspended) thread, which evaluates \( \f \) when first resumed and dies when evaluation is done
- \texttt{(suspend)}: suspends the current thread, then runs each other suspended thread till it suspends again, then resumes the current thread by returning

An example, with 3 threads:
\begin{verbatim}
(define (test-threads)
  (fork (lambda ()
    (display "hi\n") (suspend)
    (display "there\n") (suspend)))
  (fork (lambda ()
    (display "joe\n") (suspend)
    (display "louis\n") (suspend)))
  (display "A\n") (suspend)
  (display "B\n") (suspend)
  (display "C\n") (suspend)
  (display "D\n") (suspend))
\end{verbatim}

Threads via continuations (part 1 of 2)

Maintain a list of suspended "thread" objects, represented by functions to call to resume the thread
\begin{verbatim}
(define thread-queue ()

(define (enq-thread! f)
  (set! thread-queue
    (append thread-queue (list f))))

(define (deq-thread!)
  (let ((f (car thread-queue)))
    (set! thread-queue (cdr thread-queue))
    f))
\end{verbatim}

Threads via continuations (part 2 of 2)

Fork adds a new thread to the queue, which dies when done
\begin{verbatim}
(define (fork f)
  (enq-thread!
    (lambda ()
      {f}
      {run-next-thread}))
\end{verbatim}

Suspend uses \texttt{call/cc} to create a handle for the current thread, saves it, then switches to the next thread in the queue
- eventually this thread will be resumed by some other thread's suspend call
\begin{verbatim}
(define (suspend)
  (call/cc (lambda (this-thread)
    (enq-thread! (lambda () (this-thread )))
    (run-next-thread)))
\end{verbatim}

Run-next-thread runs the next thread on the queue
\begin{verbatim}
(define (run-next-thread)
  (let ((next-thread (deq-thread!)))
    (next-thread))
\end{verbatim}

Summary of continuations

Normal functions can be used as continuations
\texttt{call/cc} reifies the implicit internal continuation as a function that can be manipulated like any other function

First-class continuations can do things that otherwise require special language constructs
- exception throwing
- stack-unwind protection (like Java's try-finally)
- coroutines and (non-preemptive) threads
- backtracking à la Prolog

Very powerful, which can be very confusing, and hard to implement efficiently
Example: what should happen if a \texttt{call/cc} continuation function is invoked more than once?
- e.g. \texttt{suspend} didn't dequeue the thread, but left it on the queue to be resumed again