Week 6: Programming with Non-Deterministic Choice

CSC324 Principles of Programming Languages

David Liu, Department of Computer Science

Midterm next week!
A nondeterministic expression

Recall: streams are a way for us to describe a sequence of values over time.

\[
\text{range}(0, 100)\
\text{bisections } f \ a \ b
\]
The **ambiguous operator**, denoted -<, is an operator which represents a nondeterministic choice of one of its subexpressions.

```
(-< 1 2 3) ; 1, 2 or 3
(+ 10 (-< 1 2 3)) ; 11, 12, or 13
```

In our implementation, -< will evaluate and return its first argument, but the cooperating function next! can be used to access the others.

```
> (-< 1 2 3)
1
> (next!)
2
> (next!)
3
> (next!)
'done
```
We implement -< and next! as a “self-modifying stream”. Mutation ahead!

```scheme
;; Global variable storing any remaining choices.
(define choices (void))
(define (set-choices! val) (set! choices val))
```
;(define-syntax -<
  (syntax-rules ()
    
    
    
    
    
    
    
    
    
)
What about nested uses?

```scheme
> (+ 10 (<< 1 2 3))
11
> (next!)
12
> (next!)
13
> (next!)
'done
```

Continuations
Problem: given (+ 10 (-> 1 2 3)), our current implementation only stores the remaining choices, (-> 2 3).
It needs to store the **execution context** (+ 10 _) too.

**execution context** (of an expression)
a representation of what remains to be computed after the expression is evaluated
also called the expression’s **continuation**
In the stack-based model of program execution, the continuation of an expression is represented by the state of the call stack after the expression has been evaluated.

In functional programming, the continuation of an expression is represented by a unary function derived from the enclosing expression.
Example: \((+ (* \ 2 \ 3) (- \ 5 \ 4))\)

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ 3</td>
<td>((+ (* \ 2 \ _) (- \ 5 \ 4)))</td>
</tr>
<tr>
<td>((* \ 2 \ 3))</td>
<td>((+ \ _ (- \ 5 \ 4)))</td>
</tr>
<tr>
<td>\ 5</td>
<td>((+ 6 (- \ _ \ 4)))</td>
</tr>
<tr>
<td>\ +</td>
<td>((\ _ 6 \ 1))</td>
</tr>
</tbody>
</table>

\((+ (* \ 2 \ 3) (- \ 5 \ 4))\) \_

**shift:** capturing continuations
A shift expression:

1. Binds its enclosing continuation to `<id>`.
2. Evaluates `<body>`, with `<id>` in scope.
3. Discards its enclosing continuation, and returns the value of `<body>`.

Code demos!

Note how shift captures continuations dynamically.
Problem: given (+ 10 (-< 1 2 3)), our current implementation only stores the remaining choices, (-< 2 3).

It needs to store the execution context (+ 10 _) too.

( original )

([](-< expr1 expr2 ...)
  (begin
    (set-choices! (thunk (-< expr2 ...)))
    expr1))]

( using shift )

([](-< expr1 expr2 ...)
  (shift k
    (set-choices! (thunk (k (-< expr2 ...))))
    (k expr1))))]
Monday Oct 21: no lecture, extra office hours (10-12pm, BA4260) instead

Problem 1: shift is too powerful!
After evaluating (* 100 (\(1< 2 3\))), choices is

\[
\text{(thunk ((* 100 \_)
          (\(1< 2 3\))))}
\]

\[
\text{(thunk ((* 100 \_)
          (shift k2
              (set-choices! (thunk (k2 (\(1< 3))))
              (k2 2)))))}
\]

Calling choices captures its entire enclosing continuation, not just the (* 100 \_)!
**reset: delimiting continuations**

reset is a “barrier” that limits the effect of a shift.

Related:

- `break` affects the nearest enclosing loop
- `return` affects the nearest enclosing function
- `throw/raise` affects the nearest enclosing `try-except`

\[
\text{(reset } \text{<body}>)\]

Evaluate and return `<body>`, but any shift inside `<body>` only captures and discards continuations up to the `reset`. 
Code demos and fixing next!

Problem 2: terminating with 'done
When `next!` returns `DONE`, we want to immediately return `DONE`, without executing any of the `next!`’s continuation.

How do we discard the current continuation?
Combining multiple nondeterministic expressions

Problem: both \((-< 1 2)\) and \((-< 10 20)\) write to choices.

\[(+ (-< 1 2) (-< 10 20))\]
Idea: make choices store a collection of thunks (each representing a different choice point).

**choices as a stack**

```
(define choices null)
(define (add-choice! c) (push! choices c))
(define (get-choice!) (pop! choices))
```
choices as a stack

```
(define-syntax -<
  (syntax-rules ()
    [(<- <expr1>) <expr1>]
    [(<- <expr1> <expr2> ...)
      (shift k
        (add-choice! (thunk (k (-< <expr2> ...))))
        (k <expr1>))])
```

choices as a stack

```
(define (next!)
  (if (null? choices)
      (shift k DONE)
      (reset ((get-choice!)))))
```
A choice of number between \texttt{start} and \texttt{end} is either \texttt{start} or a choice of number between \texttt{start} and \texttt{end - 1}.

\begin{verbatim}
(define (num-between start end)
  (if (equal? start (- end 1))
      (-< start)
      (-< start (num-between (+ 1 start end)))))
\end{verbatim}