Midterm Reminders

This **Wednesday**!

Closed-book, but there’s an aid sheet we’ll provide.

Four questions, mixture of short answer and programming. (Similar format to past tests.)
Last time, we used \texttt{let/cc} to store the current continuation of a choice expression (\texttt{-< \ ...}), enabling subsequent choices to “remember” the original execution context.

\begin{verbatim}
\> (+ 10 (\texttt{-< 1 2 3}))
11
\> (next)
12
\> (next)
13
\> (next)
\texttt{done}
\end{verbatim}
Today, two goals:

1. Be able to use (next) within larger expressions.
2. Use multiple choice expressions at once.
By default...

let/cc binds the entire computational context

Calling a continuation replaces the entire computational context
The syntactic form **prompt** delimits the scope of a continuation or continuation call.
Really...

`let/cc` binds the entire computational context up to the nearest enclosing prompt.

Calling a continuation replaces the entire computational context up to the nearest enclosing prompt.
Demos and fixing get-choices
Keeping (next) consistent: abort
Storing multiple choices!
( + ( (-< 1 2) ( <-> 10 20)) )

11

(next)

21

(next)

12

(next)

22

(next)

' done
Stack-based implementation

- next-choice is now a stack of thunks (rather than a single thunk)
- each time a (\(<\ldots\)) is evaluated, push the thunk
- each time next is called, pop a thunk
<table>
<thead>
<tr>
<th>Expression</th>
<th>choices stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ (&lt; 1 2)</td>
<td>(\text{choices stack})</td>
</tr>
<tr>
<td>(-&lt; 10 20)</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>choices stack</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>(+ ((&lt; 1) 2) ((&lt; 10) 20))</td>
<td></td>
</tr>
<tr>
<td>(+ 1 ((&lt; 10) 20))</td>
<td>((+ _ ((&lt; 10) 20)) ((&lt; 2)))</td>
</tr>
<tr>
<td>Expression</td>
<td>choices stack</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>(+ ((-&lt;) 1 2) ((-&lt;) 10 20))</td>
<td></td>
</tr>
<tr>
<td>(+ 1 ((-&lt;) 10 20))</td>
<td>((+ _ ((-&lt;) 10 20)) ((-&lt;) 2))</td>
</tr>
<tr>
<td>(+ 1 10)</td>
<td>(((+ 1 _) ((-&lt;) 20)) ((-&lt;) 2))</td>
</tr>
<tr>
<td>Expression</td>
<td>choices stack</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>(next)</td>
<td>(((+ 1 _ ) (¬&lt; 20))</td>
</tr>
<tr>
<td></td>
<td>(((+ _ (¬&lt; 10 20)) (¬&lt; 2))</td>
</tr>
<tr>
<td>Expression</td>
<td>choices stack</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
</tr>
<tr>
<td>(next)</td>
<td>((+ 1 _ ) (-&lt; 20))</td>
</tr>
<tr>
<td></td>
<td>()+ _ (-&lt; 10 20)) (-&lt; 2))</td>
</tr>
<tr>
<td>(+ 1 (-&lt; 20))</td>
<td>((+ _ (-&lt; 10 20)) (-&lt; 2))</td>
</tr>
<tr>
<td>Expression</td>
<td>choices stack</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>(next)</td>
<td>(((+ 1 _) (&lt;- 20)) (&lt;- 20))</td>
</tr>
<tr>
<td></td>
<td>(((+ _ (&lt;- 10 20)) (&lt;- 2))</td>
</tr>
<tr>
<td>(+ 1 (-&lt; 20))</td>
<td>(((+ _ (&lt;- 10 20)) (&lt;- 2))</td>
</tr>
<tr>
<td>(+ 1 20)</td>
<td>(((+ _ (&lt;- 10 20)) (&lt;- 2))</td>
</tr>
</tbody>
</table>
<- : Applications and extensions
Warm-up: generating a range

A number between \textit{start} and \textit{end} is one of:

- \textit{start}
- a number between \((\textit{start} + 1)\) and \textit{end}
Generating expressions!

A rank 0 expression is an atom.
A rank k expression is a rule applied to one of:
- two rank \((k-1)\) expressions
- a rank \((k-1)\) expression and a rank \((0\) to \(k-2)\) expression
- a rank \((0\) to \(k-2)\) expression and rank \((k-1)\) expression
Generating expressions!

A rank 0 expression is an atom.
A rank k expression is a rule applied to one of:
- two rank (k-1) expressions
- a rank (k-1) expression and a rank (0 to k-2) expression
- a rank (0 to k-2) expression and rank (k-1) expression
Backtracking search

1. Define some choices and a predicate $P$.
2. Check whether the given choices satisfy $P$.
3. If they don’t, *backtrack* and make a different choice!
Encapsulating the choices stack

(define x (num-between 0 10))
; lots of code
(define y (num-between 100 200))
A generator is an object* that yields values over time.

*In our context, we’ll talk about functions.
Technical demo!

But the implementation is not required course content.