Choices and backtracking
Execution context (of an expression): a representation of what remains to be computed *after* the expression is evaluated

Also known as the expression’s continuation
> ( <<- 1 2 3 )

1

> (next)

2

> (next)

3

> (next)

'done
> (+ 10 (-< 1 2 3))
11
> (next)
12
> (next)
13
> (next)
'done
let/cc ("let current continuation")

(let/cc <id>
  <expr> ...
)

1. Binds <id> to the continuation of the let/cc expression.
2. Evaluates each <expr> ... and returns the last one (like begin).
(+ 10 (-< 1 2 3))

(+ 10 (let/cc k
  (set! next-choice (thunk (k (-< 2 3)))))
  1))
Continuations are computed \textbf{dynamically} when the \texttt{(let/cc ...)} is evaluated
Towards **first-class** choice expressions
let/cc binds the **entire computational context**

Calling a continuation replaces the **entire computational context**
Delimited continuations and prompt
Storing multiple choice points
\[(+ (\langle 1 2 \rangle) (\langle 10 20 \rangle))\]

11

\[(\text{next})\]

21

\[(\text{next})\]

12

\[(\text{next})\]

22

\[(\text{next})\]

\['\text{done}']
Stack-based implementation

- each time a \((\_\_\_<\ ...\ ))\) 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) (-&lt; 10 20))</td>
<td>null</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; 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>(((+ _ (10 20)) (&lt; 2))</td>
</tr>
<tr>
<td>(+ 1 (&lt; 20))</td>
<td>(((+ _ (10 20)) (&lt; 2))</td>
</tr>
<tr>
<td>(+ 1 20)</td>
<td>(((+ _ (10 20)) (&lt; 2))</td>
</tr>
</tbody>
</table>
Extended example: generating expressions!
Choices and backtracking
a = amb(1, 2, 3, 4, 5)
b = amb(55, 56, 57, 58, 59, 60)
c = a + b

require prime?(c)
print(a, b, c)
Towards **declarative** programming