CSC151.02 2014S, Class 31: Naming Local Procedures

Overview

- Preliminaries.
  - Admin.
  - Exam questions.
  - Other questions.
- Topics.
  - Why have local procedures.
  - Creating local procedures with \texttt{letrec}.
  - Creating local procedures with named let.
- Lab.

Preliminaries

Upcoming Work

- Work on the exam! Due Thursday night.
- Reading for Friday: Characters and Strings
- No writeup for today. You’re working hard enough on the exam.
- You’ll be recovering from the exam, so there will not be a quiz on Friday.

Admin

- Continue with Tuesday’s partners.
- Extra credit:
  - CS Table Friday (forthcoming)

Exam Questions

\textit{How do l-s and r-s work?}

\begin{verbatim}
(define l-s
  (lambda (fun left)
    (lambda (right) (fun left right)))))
\end{verbatim}

E.g., if we have + which expect two parameters

\begin{verbatim}
(l-s + 2) is (lambda (x) (+ 2 x))
\end{verbatim}
Right section, in contrast

(define r-s
  (lambda (fun right)
    (lambda (left) (fun left right))))

(r-s / 2) is (lambda (x) (/ x 2))

Sectioning, in general:

(foo _ _ _ _) => fill in one parameter, to give a function that expects fewer parameters => (foo _ _ 3 _)

For problem 3, is it okay that the output is one of 0, 64, 128, 192, 256?

Yes, that’s fine.

Can you answer problem 3 for me?

No

Can you help me think about the problem?

Outputs are 064, 164, 264, 364, and 4*64

So, we need an expression that maps one range of values to 0, the next to 1, the next to 2, and so on and so forth. If I wanted to go from 0-255 to 0 or 1, I’d divide by 128 and round down

I need to divide into more sections.

And it’s a bit more subtle than that

Can you help us think about how you would do six?

(define image-flatten
  (lambda (image valid-components)
    (image-variant image FUN)))

Can you spend the time to create an example for me?

Sure.

(image-variant image (lambda (color) ...))

Other Questions

Can you explain about placeholders and bindings?

Scheme keeps track of names and values associated with those names
If you write `(define x 2)`, we get something like the following

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
</tr>
</tbody>
</table>

But sometimes you reuse names

```
(let ([x (+ x 5)])
  ...)
```

In this code, the Scheme interpreter evaluates (+ x 5) using the old table, then bounds x to the result in an extended table

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>7</td>
</tr>
</tbody>
</table>

What happens with recursive procedures?

```
(let ([rac (lambda (lst) (if (null? (cdr lst)) (car lst) (rac (cdr lst))))]) (rac (list 1 2 3 4 5)))
```

We want the two racs to be the same.

But the inner rac refers to a previously defined procedure.

We get around this by using an alternate to `let`

- let’s behavior: Evaluate then put in the table
- Desired behavior: Add to table, then evaluate, then update table

We use `letrec` rather than `let` to get this behavior.

Why is `letrec` different than `define`, other than that we write even more parentheses and square brackets?

Sometimes we want to limit access. `letrec` says "You can only call the procedure here."

```
(letrec ([proc ...])
  ; proc is available here
  ...
)
```

; Now it’s now longer available

Why have local procedures

Creating local procedures with `letrec`