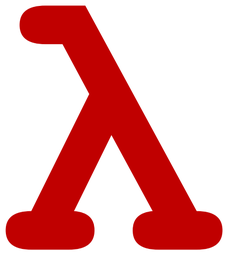
CSCI 275: Programming Abstractions

Lecture 20: MiniScheme D, Conditionals

Spring 2025

Stephen Checkoway
Slides from Molly Q Feldman



Some General Purpose Notes

Racket Style

- Never use multiple `cond`s, when you can just use one!
- Nested `define`s? `letrec` instead.

- When building something recursive, base cases *first* in the `cond`
  - `else` case should almost always be recursive call

- Logic choices should be *distinct* in a `cond`

Testing

- Generally, *always* good to test the empty list case for anything that operates over a list

- MiniScheme testing is going to be **intense!**
  - Lots of tests, best way to check that it works

- Learning how to write a good test suite: part of the class
  - Think about “code coverage”
    - Every branch of a cond should have a related test

Summary Problems

- Available!
- Your final is based strongly on these problems
    - In your interest to get feedback early
- Style, how you solve the problems, and test the solutions are part of your grade on the Final Project

Back to MiniScheme

What have we discussed already? (A-C)

EXP → number parse into `lit-exp`
| symbol parse into `var-exp`
| (*EXP EXP**) parse into `app-exp`

- Parsing and evaluating numbers
- Parsing and evaluating symbols
- Parsing and application (primitive functions)
- What an environment does

MiniScheme D: Conditionals

Heads Up! MiniScheme Booleans & If

Booleans in MiniScheme are different than in Scheme/Racket!

Booleans in MiniScheme are `True` and `False`

Like our primitive procedures, you'll need to add symbols `True` and `False` to `init-env`

Bind them to `'True` and `'False`

We'll treat anything other than `False` and `0` as being true for conditionals.

What value does MiniScheme return for this expression assuming that x is bound to 23 and y is bound to 42?

```
(if (- y x)
    25
    37)
```

A. 25

B. 37

C. It's an error because `(- y x)` is a number

Our first special form: if

EXP → number — parse into `lit-exp`

| symbol — parse into `var-exp`

| (if *EXP EXP EXP*) — parse into `ite-exp`

| (*EXP EXP**) — parse into `app-exp`

We need a new data type for the if-then-else expression: `ite-exp`

```
ite-exp

ite-exp?

ite-exp-cond

ite-exp-then

ite-exp-else
```

Parsing special forms

if, let, lambda, etc.

```
(define (parse input)
  (cond [(number? input) (lit-exp input)]
        [(symbol? input) (var-exp input)]
        [(list? input)
         (cond [(empty? input) (error ...)]
               [(eq? (first input) 'if) ...]
               [(eq? (first input) 'let) ...]
               [(eq? (first input) 'lambda) ...]
               ...
               [else (app-exp ...)])]
        [else (error 'parse "Invalid syntax ~s" input)]))
```

Make sure that input is the right length for each special form!

Parsing if-then-else expressions

If-then-else expressions are recursive

E.g., *EXP* → `( if` *EXP EXP EXP* `)`

When parsing an if-then-else expression,
you want to parse the sub expressions using `parse`

The input to `parse` will look like `'(if (lt? x 1) (+ y 100) z)`

The condition is `(second input)`

The then-branch is `(third input)`

The else-branch is `(fourth input)`

Evaluating `ite-exp`

Parse tree is recursive: `(parse '(if x 10 20))`

```
(ite-exp (var-exp 'x) (lit-exp 10) (lit-exp 20))
```

When evaluating, you should call `eval-exp` recursively

- First, call it on the conditional expression
- If the condition evaluates to `False` or `0`,
  - evaluate the last expression and return its result
- Otherwise,
  - evaluate the middle expression and return its result

What happens if you implement `eval-exp` for an `ite-exp` by calling `eval-exp` on all three parts of the expression before deciding which one to return?

```
(let ([co (eval-exp (ite-exp-cond tree) e)]
      [th (eval-exp (ite-exp-then tree) e)]
      [el (eval-exp (ite-exp-else tree) e)])
  (if co th el))
```

A. The code works perfectly

B. The code works correctly, but inefficiently on some inputs (which?)

C. The code will produce the wrong result on some inputs (which?)

D. The code will produce the wrong results on all inputs