Programming Abstractions

Lecture 21: MiniScheme D and E

Stephen Checkoway

What can MiniScheme do at this point?

MiniScheme C has numbers

MiniScheme C has pre-defined variables

MiniScheme C has procedure calls to built-in procedures

MiniScheme D: Conditionals

Booleans in MiniScheme

In Scheme: `#t` and `#f`

In MiniScheme: `True` and `False`

You'll need to add symbols `True` and `False` to `init-env`

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

In conditionals, we'll treat anything other than `False` and `0` as being true

- Why 0? Many languages treat 0 as false; Scheme does not, but MiniScheme does
- You'll have to account for this in your implementation!

New 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-cond`
- `ite-exp-then`
- `ite-exp-else`

How do we create this new datatype with this list of functions?

- `ite-exp`
- `ite-exp?`
- `ite-exp-cond`
- `ite-exp-then`
- `ite-exp-else`

A. `(new-exp ite cond then else)`

B. `(struct ite-exp cond then else)`

C. `(structure ite-exp (cond then else))`

D. `(struct ite-exp (cond then else) #:transparent)`

E. `(structure ite-exp (cond then else) #:transparent)`

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

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)]))
```

The parser

MiniScheme D

```
(define (parse input)
  (cond [(number? input) (lit-exp input)]
        [(symbol? input) (var-exp input)]
        [(list? input)
         (cond [(empty? input) (error ...)]
               [(eq? (first input) 'if)
                (if (= (length input) 4)
                    (ite-exp ...)
                    (error ...))]
               [else (app-exp ...)])]
        [else (error 'parse "Invalid syntax ~s" input)]))
```

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

C. The code works correctly, but inefficiently on all inputs

D. The code will produce the wrong result on some inputs

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

Can you evaluate all parts of the ite-exp?

What would happen if you instead called `eval-exp` on all three parts of the expression before deciding which one to return?

Think about recursive procedures using `if`

```
(define (foo n)
  (if (is-base-case? n)
      base-case-value
      (… (foo (sub1 n)) …)))
```

Primitive procedures returning booleans

Numeric procedures

- `number?`
- `eqv?` — like Scheme's `eqv?` so that it works with `True` and `False`
- `lt?` — like Scheme's <
- `gt?` — like Scheme's >
- `lte?` — like Scheme's <=
- `gte?` — like Scheme's >=

List procedures

- `null?`
- `list?`

For previous primitive procedures, we had a line like

```
[(eq? op '+) (apply + args)]
```

in apply-primitive-op.

Will

```
[(eq? op 'lt?) (apply < args)]
```

work for our less than procedure?

A. It will work because < is Racket's less than

B. It won't work because `lt?` is Racket's less than

C. It won't work because < takes two arguments and `apply` allows any number of arguments

D. It won't work because < returns `#t` or `#f`

MiniScheme E: let expressions

Let expressions

Consider

```
(let ([x (+ 3 4)]
      [y 5]
      [z (foo 8)])
  body)
```

To evaluate this, we need to extend the current environment with bindings for `x`, `y`, and `z` and then evaluate `body` in the extended environment

Extending environments

`(env list-of-symbols list-of-values previous-environment)`

Recall that the `env` constructor requires

- a list of symbols
- a list of values
- a previous environment

The parser doesn't know anything about environments but we can create a `let-exp` data type that stores

- the list of binding symbols
- the list parsed binding values
- the parsed body

Parsing let expressions

```
(let ([x (+ 3 4)] [y 5] [z (foo 8)])
  body)
```

The binding list is `(second input)` where `input` is the whole let expression

The symbols are `(map first binding-list)`

- These are *not* parsed, they're just symbols

The binding expressions are `(map second binding-list)`

- How can we parse each of these expressions?

The body is simply `(third input)` which we can `parse`

What should this code return?

```
(parse '(let ([x 10]
              [y z])
          y))
```

A.
```
(let-exp '(x y)
         (list (lit-exp 10) (var-exp 'z))
         (var-exp 'y))
```

B.
```
(let-exp (list (var-exp 'x) (var-exp 'y))
         (list (lit-exp 10) (var-exp 'z))
         (var-exp 'y))
```

C.
```
(let-exp (list (var-exp 'x) (var-exp 'y))
         '(10 z)
         (var-exp 'y))
```

D.
```
(let-exp '(x y) '(10 z) (var-exp 'y))
```

Evaluating let expressions

Evaluating a let expressions just takes a little more work

- Evaluate each of the binding expressions in the `let-exp`

```
(map (λ (exp)
        (eval-exp exp current-env))
     (let-exp-exps tree))
```

- Bind the symbols to these values by extending the current environment
- Evaluate the body of the let expression using the extended environment

What about let*?

Recall that in Scheme, let* acts like let except that variables declared earlier in the let-binding list can be used for later values

```
(define (foo x y)
  (let ([x (+ x y)]
        [y (+ x y)])
    (displayln x)
    (displayln y)))
```

```
(define (bar x y)
  (let* ([x (+ x y)]
         [y (+ x y)])
    (displayln x)
    (displayln y)))
```

`(foo 1 100)` prints `101` twice

`(bar 1 100)` prints `101` and then `201`

How could we implement `let*` in MiniScheme?