Programming Abstractions

Week 8-2: MiniScheme D and E and Lexical Bindings

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`

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`

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 is `False` or `0`, call it on the last expression
  - Otherwise, call it on the middle expression

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

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`

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` which aren't supported in MiniScheme

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 binding symbols
- the 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)`

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`

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?

Lexical Binding

Variable usage

There are two ways a variable can be used in a program:

- As a declaration
- As a "reference" or use of the variable

Scheme has two kinds of variable declarations

- the bindings of a let-expression and
- the parameters of a lambda-expression

Scope of a declaration

The scope of a declaration is the portion of the expression or program to which that declaration applies

Lexical binding

- Scope of a variable is determined by textual layout of the program
- C, Java, Scheme/Racket use lexical binding

Dynamic binding

- Scope of a variable is determined by most recent *runtime* declaration
- Bash and classic Lisp use dynamic binding

Java example

What is the scope of y in this Java program?

Could we print y instead of x in the last line?

```
public static void main(String[] args) {
    int x;
    x = 1;
    while (x < 10) {
        int y = x;
        System.out.println(y);
        x += 1;
    }
    System.out.println(x);
}
```

Scope in Scheme

Scope of variables bound (declared) in a `let` is the body of the `let`
Scope of parameters in a λ is the body of the λ

```
(let ([x 5]
      [y 10])
  (* ((λ (z) (+ z y)) 7)
     x
     y))
```

Shadowing bindings

Shadowing: Declaring a new variable with the same name as an existing variable in an enclosing scope

```
(let ([x 5]
      [y 10])
  (* ((λ (x) (+ x y)) 7)
     x
     y))
```

We say that the inner binding for x *shadows* the outer binding for x

Determining the appropriate binding

Start at the use of a variable

Search the enclosing regions starting with the innermost and working outward looking for a binding (declaration) of the variable

The first binding you find is the appropriate binding

If there are no such bindings, we say the variable is *free*

Contour diagrams

Draw the boundaries of the regions in which variable bindings are in effect

```
(λ (x)
  (λ (y)
    ((λ (x) (x y)) x)))
```

The body of a let or a lambda expression determines a contour

Each variable refers to the innermost declaration *outside* its contour

Lexical depth

The lexical depth of a variable reference is 1 less than the number of contours crossed between the reference and the declaration it refers to

```
(λ (x)
  (λ (y)
    ((λ (x) (x y)) x)))
```

In `(x y)`

- `x` has lexical depth 0
- `y` has lexical depth 1

The other `x` has lexical depth 1

What is the lexical depth of `m` in the expression `(* m x)` in this procedure?

```
(define fun
  (λ (m lst)
    (foldl (λ (x acc) (+ (* m x) acc))
           0
           lst)))
```

A. 0

B. 1

C. 2

D. 3

E. 4

Lexical addresses

(depth, position)

We can use the lexical depth of a variable along with the 0-based position of the variable in its declaration to come up with a *lexical address* of the variable

```
(let ([x 3]
      [y 4])
  (λ (a b)
    (λ (c)
      (a (+ (b x) c)))))
```

(1,0) (1,1) (2,0) (0,0)

Lexical addresses are essentially pointers to where the variable can be found on the run-time stack; can eliminate names