CSCI 275: Programming Abstractions

Lectures 18–19: MiniScheme B (conclusion) and C Start

Spring 2025

Stephen Checkoway
Slides from Molly Q Feldman

Functional Language of the Week: Scala

- Developed at EPFL
  - Academic project that has turned into mainstream language
  - 29th on the top 50 languages list

Scala is *mind bending* as it is **all of the following things:**

- Compatible with Java
  - It runs on the JVM, Scala programs act/seem like Java programs
- OOP: every value is an object
  - Subclassing, etc.
- Function: every function is a value
  - Currying is supported
  - Higher order functions

https://docs.scala-lang.org/tour/tour-of-scala.html

```java
List<Person> people;

List<Person> minors = new ArrayList<Person>(people.size());
List<Person> adults = new ArrayList<Person>(people.size());
for (Person person : people) {
   if (person.getAge() < 18)
      minors.add(person);
   else
      adults.add(person);
}
```

Java

```scala
val people: Array[Person]
// Partition `people` into two arrays `minors` and `adults`.
// Use the anonymous function `(_.age < 18)` as a predicate for
partitioning.

val (minors, adults) = people partition (_.age < 18)
```

Scala

Questions? Concerns?

Today's Goals

- Overview of making MiniScheme compute!
  - Getting to parsing and evaluating `(+ 1 2)` into `3`

MiniScheme Design

MiniScheme Design

Implementation Information

- We are implementing MiniScheme, a subset of Scheme
- We are *using* Racket to write the rules and interpret the results

Environment: `env.rkt`

- Contains the environment data type with constructor `env`
- Contains other procedures to recognize and access the symbols, values, and previous environment
- Your task is to implement

```
(env-lookup environment symbol)
```

Parser: `parse.rkt`

- Contains data types for let expressions, lambda expressions, if-then-else expressions, procedure-application expressions and so on

- Builds a parse tree out of these data types from an expression

```
> (parse '(let ([f (lambda (x) (+ x 1))]) (f 5)))
(let-exp '(f) (list (lam-exp '(x) ...))
        (app-exp ...))
```

- You get to implement all of this, bit by bit

Interpreter: `interp.rkt`

- Contains data types for closures and primitive procedures (i.e., built-in procedures)

- Takes an expression tree and an environment and returns a value

```
> (eval-exp exp-tree environment)
```

- You get to implement all of this, bit by bit, at the same time you're implementing the parser

Project Structure

Provide the definitions

```
(provide proc1 proc2 data1 data2 ...)
```

We want `parse.rkt` to be just one module in our program so make sure to provide the procedures

- `(provide parse)`
- Also the procedures for creating and manipulating the `lit-exp` by using `(provide (struct-out lit-exp))`

Read-eval-print loop

Having to call `parse` and then `eval-exp` over and over is a hassle

It'd be better if we could run a read-eval-print loop that would read in an expression from the user, parse it, and evaluate it in an environment

`minischeme.rkt` will do this but you must `(provide …)`

- In `parse.rkt, a (parse input)` procedure
- In `interp.rkt`
  - An `(eval-exp tree environment)` procedure
  - An initial environment `init-env`

    Something like

    `(define init-env (env '(x y) '(23 42) empty-env))`

MiniScheme Design

Running the read-eval-print loop

Open `minischeme.rkt` in DrRacket, click Run

Enter expressions in the box (only numbers are supported right now)

Click the `eof` button to exit MiniScheme (previously you could also type `exit`)

Notice how the prompts differ!

```
Welcome to DrRacket, version 8.5 [cs].
Language: racket, with debugging; memory limit: 128 MB.
MS> 105
105
MS> 23
23
MS> exit
returning to Scheme proper
>
```

Wrapping Up Environments

When to extend an environment?

There are only two places where an environment is extended in MiniScheme:

A. **Let expressions**

A. **Procedure calls**

A. Extending Environments: Let

Consider

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

We have three symbols `x`, `y`, and `z` and three values, 7, 5, and whatever the result of `(foo 8)` is, let's say it's 12

If `E` is the environment prior to the `let` expression, then the body should be evaluated in the environment

$E[x \mapsto 7,\ y \mapsto 5,\ z \mapsto 12]$

B. Extending environments: procedure calls

We extend the environment when we pass expressions *to arguments* during procedure calls

`(lambda (x) (first x))` called on `(list 1 2 3)`

will extend the environment by mapping `x` to `` `(1 2 3) ``

Environment of the call

`x`	`` `(1 2 3) ``

Closures store their environments!

The expression of `(lambda parameters body…)` evaluates to a *closure* consisting of

- The parameter list (a list of identifiers)
- The body as un-evaluated expressions (often just one expression)
- The environment (the mapping of identifiers to values) **at the time the lambda expression is evaluated**

Environments with closures versus calls

```
(define A 10)
(define add-a
  (lambda (x)
    (+ x A)))
```

Environment of the closure

A	10

Keep it around! Part of what the closure contains!

Calling the closure extends the closure's environment with its parameters bound to the arguments

```
(add-a 20)
```

Environment of the call

A	10
x	20

When called, the closure's body is evaluated with this new environment

Previous Slide, In General

The first expression below is a procedure call

`(exp0 exp1 … expn)`

`exp0` should evaluate to a closure with three parts

- its parameter list
- its body
- the environment in which it was created, i.e., the environment at the time the `(lambda …)` that created the closure was evaluated

`exp1 … expn` are the arguments

The closure's environment *also* needs to be extended with the parameters bound to the arguments!

Another Example

For example, imagine the parameter list was `(x y z)` and the arguments evaluated to `2`, `8`, and `'(1 2)`

If `E` is the closure's environment, then the closure's body should be evaluated with the environment `E[x ↦ 2, y ↦ 8, z ↦ '(1 2)]`

Extending environments

In both cases (let & procedure calls), we have

- A list of symbols
- A list of values
- A previous environment we're extending

We are going to want to *make a data type* representing this environment

This is Part 1 of HW5!

First Step: Lookup Only, Extension Later!

`(env-lookup environment symbol)`

Looking up `x` in an environment has two cases:

(1) If the environment is empty, then we know `x` isn't bound there so it's an error

(2) Otherwise, we look in the list of symbols of an extended environment

- If the symbol `x` appears in the list, then great, we have the value
- If the symbol `x` doesn't appear, then we lookup `x` in the previous environment

Part 1 of Homework 5: write `env-lookup`

Back to Evaluating Symbols!

Assume that `x` is bound to 10 and `y` to 25 in an environment called `init-env`.

What do we want `(eval-exp (parse 'x) init-env)` to return?

A. 10

B. 'x

C. 25

D. Error

E. Something else

How do we edit `parse` and `eval-exp` to handle symbols?
Work on adding a case to each `cond` in your small groups.
Vote A when done.

```
(define (parse input)
  (cond [(number? input) (lit-exp input)]
    HANDLE SYMBOL HERE
        [else (raise-user-error ...)]))

(define (eval-exp tree e)
  (cond [(lit-exp? tree) (lit-exp-num tree)]
    HANDLE SYMBOL HERE
        [else (error ...)]))
```

Parsing symbols

```
(define (parse input)
  (cond [(number? input) (lit-exp input)]
        [(symbol? input) (var-exp input)]
        [else (raise-user-error
                'parse
                "Invalid syntax ~s" input)]))
```

When I run `(parse 'foo)`, I get `(var-exp 'foo)`

Interpreting symbols

```
(define (eval-exp tree e)
  (cond [(lit-exp? tree) (lit-exp-num tree)]
        [(var-exp? tree)
         (env-lookup e (var-exp-symbol tree))]
        [else (error ...)]))
```

You'll need a working `env-lookup` !

```
> (env-lookup init-env 'x)
23
> (eval-exp (var-exp 'x) init-env)
23
```

MiniScheme C Overview

We have thought about this part of MiniScheme thus far

Grammar

$EXP \rightarrow$	number	parse into `lit-exp`
$\mid$	symbol	parse into `var-exp`

Let's add arithmetic and some list procedures

Let's add `+`, `-`, `*`, `/`, `car`, `cdr`, `cons`, etc.

This is the first "complex" part

- It contains some things that make more sense later, once we add `lambda` expressions

Scheme is all about lists

So far, we have only dealt with a number or a symbol as input

If the input is a list, then the kind of expression it represents depends on the **first element.** For instance:

- If the first element is `lambda`, it's a lambda expression
- If the first element is `let`, it's a let expression
- If the first element is `if`, it's an if-then-else expression

Procedure applications don't have keywords, so any nonempty list for which the first element is not one of our supported keywords is an application

`(foo x 8 y)` is an application with procedure `foo` and arguments `x`, `8`, and `y`

Which grammar rule supports procedure calls like `(+ 10 15)` and `(car lst)`?

EXP → number   parse into `lit-exp`
  | symbol   parse into `var-exp`
  | ???

A. (*PROC ARGS*)

B. (*PROC ARG**)

C. (symbol *EXP**)

D. (*EXP**)

E. (*EXP EXP**)

Challenge: many ways to call procedures

```
(+ 2 3)

((lambda (x y) (+ x y)) 2 3)

(let ([f +]) (f 2 3))
```

The parser can't identify all primitive procedures like + because symbols like `f` may be bound to primitive procedures

Important: **the parser cannot tell because it does not have access to the environment**

All that the parser can do is recognize a procedure application and parse (1) the procedure and (2) the arguments

Procedure applications

MiniScheme C

$EXP \rightarrow$ number parse into `lit-exp`
$\quad\mid$ symbol parse into `var-exp`
$\quad\mid$ ($EXP\ EXP^*$) parse into `app-exp`

An `app-exp` is a new data type that stores

- The parse tree for a procedure
- A list of parse trees for the arguments

Parsing, Recursively!

Expressions are recursive: *EXP* → (*EXP EXP**)

When parsing an application expression, you want to parse the sub expressions using parse

```
(define (parse input)
  (cond [(number? input) (lit-exp input)]
        [(symbol? input) (var-exp input)]
        [(list? input)
         (cond [(empty? input) (raise-user-error ...)]
               [else (app-exp (parse (first input))
                              (...))])]
        [else (raise-user-error ...)]))
```

What is the result of `(parse '(foo x y z))`?

A. `(app-exp 'foo '(x y z))`

B. `(app-exp (var-exp 'foo) '(x y z))`

C.
```
(app-exp (var-exp 'foo)
         (list (var-exp 'x) (var-exp 'y) (var-exp 'z)))
```

D.
```
(app-exp 'foo
         (list (var-exp 'x) (var-exp 'y) (var-exp 'z)))
```

E. It's an error because the variables `foo`, `x`, `y`, and `z` aren't defined

What is the result of `(parse '(foo (add1 x))`?

```
A. (app-exp (var-exp 'foo)
            (app-exp (var-exp 'add1) (var-exp 'x)))

B. (app-exp (var-exp 'foo)
            (list (app-exp (var-exp 'add1) (var-exp 'x))))

C. (app-exp (var-exp 'foo)
            (list (app-exp (var-exp 'add1)
                           (list (var-exp 'x))))))
```

D. It's an error

Evaluating an `app-exp`

1. Evaluate the procedure part

2. Evaluate each of the arguments

3. If the procedure part evaluates to a primitive procedure, call a Racket procedure you'll write that will perform the operation on the arguments
   - E.g., if the primitive procedure is `*`, then you'll want to call `*` on the arguments

Right now, primitive procedures are going to be the only supported procedures

Part 1 is the tricky part: what should the result of evaluating the procedure part be?

Restated: Evaluating an app-exp

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

STEP 1: Evaluate the procedure

STEP 2: Evaluate the arguments

STEP 3: Actually *apply* the procedure

Evaluating the procedure part of an `app-exp`

Consider the input `'(+ 2 3 4)`

The procedure part is `'+` which will be parsed as `(var-exp '+)`

Variable reference expressions are evaluated by looking the symbol up in the current environment

Therefore, we need our initial environment to contain a binding for the symbol `'+` (and friends) so we know what it "is"

Data Type for Primitive Procedures!

We can create a new data type `prim-proc`

We're going create a bunch of these

```
(prim-proc '+)
(prim-proc '-)
(prim-proc 'car)
(prim-proc 'cdr)
(prim-proc 'null?)
```

...

`prim-proc`

A `prim-proc` is a **value** that will be returned by `eval-exp`, just like numbers are in MiniScheme now

A `(prim-proc 'car)` is to the MiniScheme interpreter exactly the same thing `#<procedure:car>` is to DrRacket

Since prim-proc is **only** used to interpret expressions, where should this data type be defined?

Binding variables to prim-proc

In DrRacket, `+` is bound to `#<procedure:+>`

In MiniScheme, `+` needs to be bound to `(prim-proc '+)` in our initial environment, `init-env`

And similarly for `-`, `*`, `/`, `car`, `cdr`, `null?` etc.

Adding primitives to our initial environment

```
(define primitive-operators
  '(+ - * /))

(define prim-env
  (env primitive-operators
       (map prim-proc primitive-operators)
       empty-env))

(define init-env
  (env '(x y) '(23 42) prim-env))
```

Evaluating an app-exp

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

STEP 1: Evaluate the procedure [DONE!]

STEP 2: Evaluate the arguments

STEP 3: Actually *apply* the procedure

STEP 2: Evaluating the arguments

In parse, we could simply map parse over the arguments to get a list of trees corresponding to our arguments.

We cannot simply use `(map eval-exp (app-exp-args tree))` to evaluate them, why?

STEP 2: Evaluating the arguments

In parse, we could simply map parse over the arguments to get a list of trees corresponding to our arguments

We cannot simply use `(map eval-exp (app-exp-args tree))` to evaluate them, why?

`eval-exp` requires an environment! so, we need to make sure we include the environment as part of the map.

STEP 3: Applying the procedure to the arguments

```
(define eval-exp ...
  [(app-exp? tree)
   (let ([proc (eval-exp (app-exp-proc tree) e)]
         [args (map ... (app-exp-args tree)])
     (apply-proc proc args))]
  ...)
```

```
(define (apply-proc proc args)
  (cond [(prim-proc? proc)
         (apply-primitive-op (prim-proc-symbol proc) args)]
        [else (error 'apply-proc "Bad proc: ~s" proc)]))
```

1. Arguments are an evaluated procedure and a list of evaluated arguments

2. Checks whether procedure is a primitive?

   If so, it will call `apply-primitive-op`

   If not, it's an error for now; later, we'll handle this case

apply-primitive-op

`(apply-primitive-op op args)`

`op` is the name of the primitive, e.g., `'+` or `'car`

Apply-primitive-op needs to check that the arguments are the appropriate types (e.g., `+` only works if all of the arguments are numbers) and there are an appropriate number of them

If the arguments are wrong, `raise-user-error` should be used to raise an error

Recap: evaluating an `app-exp`

`eval-exp`

Determines that the passed in expression is an `app-exp`

Evaluates the procedure in the `app-exp` in the environment to get a value

Evaluates each of the arguments in the `app-exp` to get a list of values

Calls `(apply-proc proc args)`

`apply-proc`

If the passed in proc is a prim-proc, then call
`(apply-primitive-op (prim-proc-symbol proc) args)`

Otherwise, error

`apply-primitive-op`

Based on the passed in symbol, checks the arguments and then applies the corresponding Racket function to the `args` and returns the result