LAB 10

Scheme

Mingxiao Wei
mingxiaowei@berkeley.edu

Oct 31, 2022

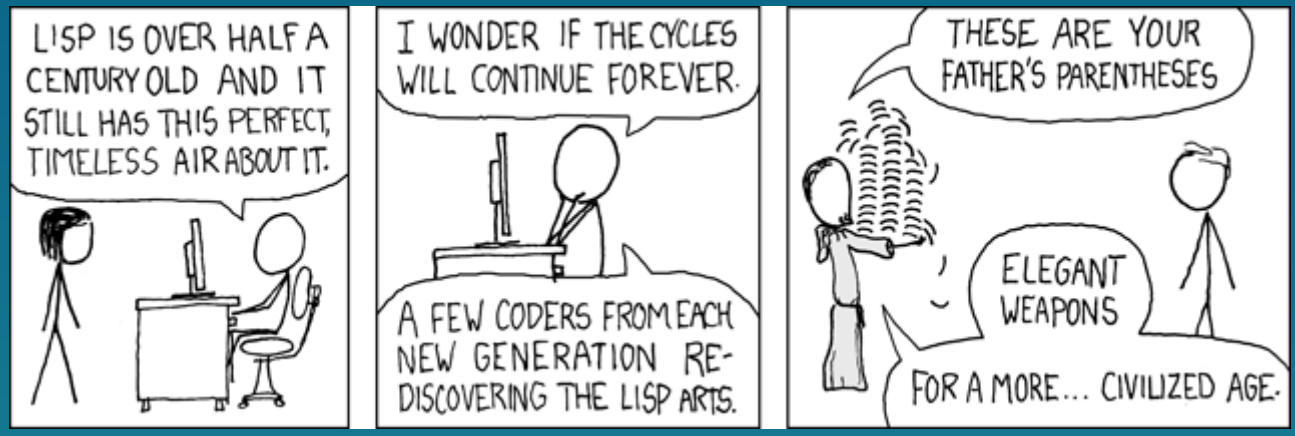
LOGISTICS 

- Congrats on finishing the midterm!
  - Regrade requests are now open!
  - Clobber policy will be announced soon
- Welcome to the world of `(scheme)` 
- Lab 10 due Wed 11/02
- Homework 07 due Thu 11/03

SCHEME 

SCHEME - INTRO

- A dialect of Lisp
- Uses prefix notations
  - `(func op1 op2)`
  - (many) nested parenthesis as a result



- Features tail-call optimizations - will be covered later
- **EVERYTHING IN SCHEME EVALUATES TO A VALUE** (In contrast, we have statements in python, which do not evaluate to anything)

SCHEME - LOGISTICS

For assignment with Scheme questions, in the assignment directory:

- To start a Scheme interpreter, type `python3 scheme`
- To run a program interactively, type `python3 scheme -i <file.scm>`
- To exit the interpreter, type `(exit)`
- To start the editor, type `python3 editor`
  - To quit the editor, use `Ctrl + c`
  - To run tests without quitting the editor, run the ok commands in a separate terminal window
- Scheme Built-In Procedure Reference
- Scheme Specification

PS: You can also edit the `.scm` file in your own editor, but our version has visualizations that makes parenthesis matching much easier!

SCHEME - PRIMITIVE EXPRESSIONS

- Primitive Expressions:
  - self-evaluating
  - Includes numbers, booleans, symbols

```
scm> 6
6
scm> 10.0
10.0
```

SCHEME - PRIMITIVE EXPRESSIONS

- Symbols
  - A symbol is a type of value in scheme
  - Act like variable names in Python, but not exactly the same
  - In Scheme, a symbol can evaluate to a value; an expression can also evaluate to a symbol

```
scm> quotient
#[quotient]  ; representation of built-in procedures
scm> 'quotient  ; Scheme uses single quotation mark
quotient  ; the string above evaluates to a symbol
scm> 'hello-world!
hello-world!  ; also a symbol value
```

SCHEME - PRIMITIVE EXPRESSIONS

- Booleans
  - `#t` in Scheme ↔ `True` in Python
  - `#f` in Scheme ↔ `False` in Python
  - `#f` is THE ONLY FALSY VALUE in Scheme! (0 is truthy!)

```
scm> #t
#t
scm> #f
#f
```

SCHEME - CALL EXPRESSIONS

```
(func op1 op2 ...)
```

- Operator is WITHIN the parenthesis, and comes first
- Operator/operands are separated by whitespace, NOT comma
- Same evaluation rule as in Python:
    1. Evaluate the operator, which should evaluate to a procedure*
    2. Evaluate the operands from left to right
    3. Apply the procedure to the operands

* In Scheme, functions are called procedures

SCHEME - CALL EXPRESSIONS

```
scm> (+ 1 (* 2 3) (- 5 4)) ; 1 + (2 * 3) + (5 - 4)
8
scm> (- (/ 10 4) 1) ; (10 / 4) - 1
1.5
scm> (modulo 10 4) ; modulo -> %
2
scm> (even? (quotient 45 2)) ; quotient -> //
#t
scm> (not (= 1 2)) ; operands to = must be numbers
#t
```

Exercise: translate `6 * 3 - 2 >= 0` to Scheme

SCHEME - QUOTES

- Scheme use a single quotation mark, `'<expression>`
  - The quotation mark only applies to the expression right after itself
- Equivalent form: `(quote <expression>)`
- Return the `<expression>` exactly as it is without any evaluation

```
scm> 'hello-world
hello-world
scm> '(+ 1 2)
(+ 1 2)
```

SCHEME - BUILT-IN PROCEDURES

Scheme	Python
`(/ a b)`	`a / b`
`(quotient a b)`	`a // b`
`(modulo a b)`	`a % b`
`(= a b)`	`a == b`
`(not (= a b))`	`a != b`

SCHEME - SPECIAL FORMS

- Special - do not follow the evaluation rules for call expressions (eg, short-circuiting)
- Check out the Scheme Specification for a complete list of special forms
- Includes `and`, `or`, `if`, `cond`, etc.

```
scm> (and 0 1 2 3) ; 0 in Scheme is truthy!
3
scm> (or 0 1 2 3)
0
scm> (and (> 1 6) (/ 1 0)) ; short-circuiting applies
#f
scm> (or (< 1 6) (/ 1 0))
#t
```

SCHEME - CONTROL STRUCTURES

`(if <predicate> <if-true> [if-false])`*

- Evaluation rules
    1. Evaluate `<predicate>`
    2. If `<predicate>` evaluates to a truthy value, evaluate `<if-true>` and return its value. Otherwise, evaluate and return `[if-false]`
    3. `[if-false]` is optional. If not provided and `<predicate>` is falsy, returns `undefined` - Scheme's version of `None` (not displayed in the interpreter unless printed)
- Only one of `<if-true>` and `[if-false]` is evaluated
- The whole special form evaluates to either `<if-true>` or `[if-false]`
- No `elif` - if more than 2 branches, use nested `if`'s or `cond`

* In our Scheme Specification, `<>` is used to denote required components while `[]` is used to denote optional components

SCHEME - CONTROL STRUCTURES

Scheme	Python

`scm> (if (> x 3)`
`1`
`2)`
	`>>> if x > 3:`
	`... 1`
	`... else:`
	`... 2`

`scm> (if (< x 0)`
`'negative`
`(if (= x 0)`
`'zero`
`'positive`
`)`
`)`
	`>>> if x < 0:`
	`... 'negative'`
	`... else:`
	`... if x == 0:`
	`... 'zero'`
	`... else:`
	`... 'positive'`

Note: Indentation / line break does NOT matter in Scheme

SCHEME - CONTROL STRUCTURES

Scheme `if`	Python `if`
A special form expression that evaluates to a value	Some statement that directs the flow of the program
Expects just a single expression for each of the true result and the false result	Each suite can contain multiple lines of code
No `elif`	Has `elif`

SCHEME - CONTROL STRUCTURES

```
(cond
    (<p1> <e1>)
    ...
    (<pn> <en>)
    [(else <else-expression>)]) ; else is optional
```

- Similar to a multi-clause if/elif/else conditional
- Takes in an arbitrary number of arguments known as clauses
  - Clause: `(<p> <e>)`
- Evaluation rules:
  1. Evaluate the predicates `<p1>`, `<p2>`, ..., `<pn>` in order until you reach one that evaluates to a truth-y value.
  2. If you reach a predicate that evaluates to a truth-y value, evaluate and return the corresponding expression in the clause.
  3. If none of the predicates are truth-y and there is an `else` clause, evaluate and return `<else-expression>`; otherwise return `undefined`

SCHEME - CONTROL STRUCTURES

```
(cond
    (<p1> <e1>)
    ...
    (<pn> <en>)
    [(else <else-expression>)]) ; else is optional
```

- `cond` is a special form because it does not evaluate every operands - short circuits upon reaching the first predicate that evaluates to a truth-y value
- Only one clause has its `<e>` evaluated (and returned)
  - The whole `cond` special form evaluates to the value of this `<e>`
- Order of clauses matters - only move on to the next clause if all previous predicates are falsy

SCHEME - CONTROL STRUCTURES

Scheme	Python

`scm> (cond`
`((> x 0) 'positive)`
`((< x 0) 'negative)`
`(else 'zero)`
`)`
	`>>> if x > 0:`
	`... 'positive'`
	`... else:`
	`... if x < 0:`
	`... 'negative'`
	`... else:`
	`... 'zero'`

`scm> (cond`
`((> x 3) 1)`
`(else 2)`
`)`
	`>>> if x < 3:`
	`... 1`
	`... else:`
	`... 2`

Note: Indentation / line break does NOT matter in Scheme

SCHEME - DEFINE VARIABLES

`(define <name> <expression>)`

- Evaluation rules
  1. Evaluate the `<expression>`
  2. Bind its value to the `<name>` in the current frame
  3. Return `<name>` as a symbol
- Evaluates to `<name>` (a symbol value)

```
scm> (define x (+ 6 1))
x
scm> x
7
scm> (+ x 2)
9
```

SCHEME - DEFINE FUNCTIONS

```
(define (<func-name> <param1> <param2> ... ) <body>)
```

- Evaluation rules
    1. Create a lambda procedure with the given parameters and `<body>`
    2. Bind its procedure to the `<func-name>` in the current frame
    3. Return `<func-name>` as a symbol
- Evaluates to `<name>` (a symbol value)
- `<body>` can have multiple expressions, in which case all expressions are evaluated from left to right, and the value of the last expression is returned
- Special form because the function body is not evaluated until the function is called

SCHEME - DEFINE FUNCTIONS

```
(define (<func-name> <param1> <param2> ... ) <body>)
```

```
scm> (define (foo x y) (+ x y))
foo
scm> (foo 2 3)
5
```

SCHEME - LAMBDA FUNCTIONS

```
(lambda (<param1> <param2> ... ) <body>)
```

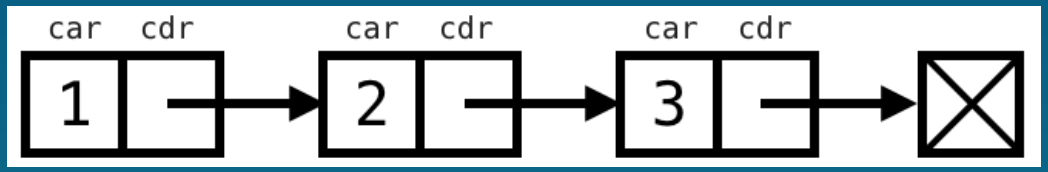
- Create and return a procedure with the given parameters and body, without alter the current environment unless we bind it to a variable.
- All Scheme procedures are lambda procedures!
- `<body>` can have multiple expressions - all expressions are evaluated from left to right, and the value of the last expression is returned

```
scm> (define foo (lambda (x y) (+ x y)))
foo
scm> (define (foo x y) (+ x y)) ; these two are equivalent
foo
scm> (foo 2 3)
5
scm> (lambda (x y) (+ x y))
(lambda (x y) (+ x y))
```

SCHEME LISTS 

SCHEME LISTS - INTRO

- All Scheme Lists are linked lists! 😮😮
- 3 ways to construct a linked list:




```
scm> (cons 1 (cons 2 (cons 3 nil))) ; nil -> Link.empty
(1 2 3)
scm> (list 1 2 3)
(1 2 3)
scm> '(1 2 3)
(1 2 3)
```

- `(car lst)` - returns **the first element** from the `lst`, analogous to `link.first`
- `(cdr lst)` - returns the rest of the `lst` as **another Scheme list**, analogous to `link.rest`

SCHEME LISTS - INTRO

```
scm> (define lst (cons 1 (cons 2 (cons 3 nil))))
lst
scm> lst
(1 2 3)
scm> (car lst)
1
scm> (cdr lst)
(2 3)
scm> (car (cdr (cdr a)))
3
```

SCHEME LISTS - CONSTRUCTOR

`(cons <first> <rest>)`

- Similar to a linked list constructor
- `<first>` is the first element of the list
- `<rest>` must be another Scheme list, or `nil` if there's no more element
- `<rest>` is required
- Useful for recursion problems

```
scm> (define a (cons 1 (cons 'a nil)))
a
scm> a
(1 a)
scm> (cons 6 a)
(6 1 a)
```

SCHEME LISTS - CONSTRUCTOR

`(list <ele1> <ele2> ...)`

- Takes in an arbitrary number of elements in the list
- Evaluate each element (which could be an expression) from left to right, and return them as a Scheme list
- Useful when we know exactly what elements are in the list

```
scm> (define a (+ 6 1))
a
scm> a
7
scm> (list (- a 1) a (+ a 1))
(6 7 8)
```

SCHEME LISTS - CONSTRUCTOR

`'(...)`

- Construct the exact list that is given, without any evaluation
- Equivalent to `(quote ...)`

```
scm> (define a (+ 6 1))
a
scm> (list (- a 1) a (+ a 1))
(6 7 8)
scm> '(cons 1 2)
(cons 1 2)
scm> '(1 (2 3 4))
(1 (2 3 4))
```

SCHEME LISTS - BUILT-IN PROCEDURES

- `(null? lst)` - checks whether or not `lst` is empty
- `(append lst1 lst2)` - concatenates two lists together and return them as a new list
- `(length lst)` - return the length of `lst`

```
scm> (null? nil)
#t
scm> (append '(c s) '(6 1 a))
(c s 6 1 a)
scm> (length '(1 (2 3) 4))
3
```

ATTENDANCE! 

go.cs61a.org/mingxiao-att

- The attendance form and slides are both linked on our section website!
- If you finish early, let me or any of the AI's know and we'll check you off
- Once again, please do remember to fill out the form by midnight today!!