CS 131 Discussion 7

Winter 2015

Announcements

- **Homework 5**
  - Due Monday Feb 23, 23:55

Scheme

http://download.racket-lang.org/

Arithmetic and Comparisons

Adding (same thing with -, *, /)

```
> (+ 1 2)
3
```

Equivalence in value (#t = true, #f is false)

```
> (equal? "abc" "bcd")
#f
```

```
> (equal? ‘(a + “hi”) ‘(a + “hi))
#t
```

```
> (< 1 2)
#t
```

question marks are allowed in identifiers (usually used for predicates)

- ‘(a + “hi”) is a list of 3 symbols
- a, ‘+, “hi” are treated similar to *atoms* in Prolog

Quotes and Lists

Suppose I want to treat + as data (a symbol), rather than a procedure:

```
> (symbol? '+)
#t
```

```
> (symbol?  +)
#f
```

Note: Saying
(quote (1 2 3))
is the same as saying
'(1 2 3)

To create list that includes symbols

```
> (equal? (list '+ 1 2) '(+ 1 2))
#t
```

Putting quote in front of things already treated as data will have no effect

```
> (equal? (list '+ '1 '2) '(+ 1 2))
#t
```

To put procedures in the list

```
> (list + 1 2)
'(#<procedure:+> 1 2)
```

Note: Not like this

```
> (list (+ 1 2))
'(3)
```

cons, car, cdr, cadr, caadr, caddr, ...etc.

Compare to OCaml, except lists can contain whatever type they wish

cons -> ::

```
> (cons ‘a ‘(b c “hi”))
‘(a b c “hi”)
```

car -> hd

```
> (car ‘(+ 1 2))
‘+
```

cdr -> tail

```
> (cdr ‘(+ 1 2))
‘(1 2)
```

cadr -> (car (cdr (...))

```
> (cadr ‘(+ 1 2))
1
```

caadr -> (car (car (cdr (...))

...etc.

if & cond

(if [EXPR] [DOIFTRUE] [DOIFFALSE])

```
> (if (= 1 1) 1 2)
1

> (if “great” 1 2)
1
```

(cond ([EXPR1] [DOIFTRUE1]) ([EXPR2] [DOIFTRUE2]) …)

```
> (cond [(= 1 2) (/ 1 0)] [(= 1 1) “phew”])
“phew”
```

or & and

Scheme uses “short circuit evaluation”

```
> (or #f 2 3)
2
```

(or [EXPR1] [EXPR2] …)
(and [EXPR1] [EXPR2] …)

Scheme will execute every expression until it determines the value of the and/or, then return the value of the last expression it executed

```
> (and (if (= 1 1) “wow” #f) (if (= 1 1) “great” #f))
“great”

> (and (if (= 1 1) (display “wow”) #f) (if (= 1 1) “great” #f))
wow”great”
```

Defining ~~functions~~ procedures

(define ([NAME] [ARG1] [ARG2] …) [BODY])

```
> (define (square x) (* x x))
> (square 2)
4
```

```
> (define two (+ 1 1))
> two
2
```

Anonymous function

(lambda ([ARG1] [ARG2] ...) [BODY])

```
> (lambda (x) (* x x))
#<procedure>
```

```
> (lambda () (+ 1 1))
#<procedure>
```

```
> (define square (lambda (x) (* x x)))
> (square 2)
4
```

let

(let (([VAR1] [EXPR1]) ([VAR2] [EXPR2]) …) [BODY])

```
> (let ((x (+ 1 1)) (y (+ 1 2))) (+ x y))
5
```

Homework 5

Recall Homework 2 Hint

Write a matcher for a DNA pattern

DNA pattern: eg. (or a g) matches a prefix of either a or g
(list g t c) matches a prefix of (g t c)
(junk 2) matches 0 to 2 nucleotides of any value
(match-* a) matches 0 or more repetitions of a
… or any combination of the above
eg. (or (list g t c) (junk 2) (match-* a) a g)

How a matcher worked in OCaml

1. Find next matching prefix
2. If none, return None
3. Otherwise, call acceptor with matching suffix
4. Go to 1.

How a matcher works in Scheme

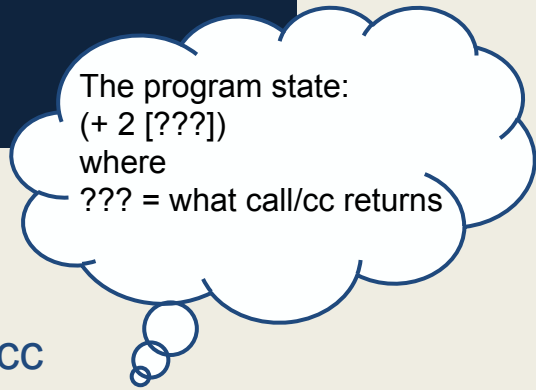
1. Find next matching prefix
2. If none, return #f
3. Save current state of execution (of the program) in a ***backtracker*** , and return (suffix, backtracker)
4. If someone calls the backtracker, “rewind time” (!!!) and go back to the state saved by the backtracker and continue to 1.

Implementing a backtracker: call/cc

(call/cc (lambda (**k**) [EXPR]))

When call/cc is invoked:

1. The current state of program execution is represented by the procedure **k** of 1 argument
2. [EXPR] is executed
3. If [EXPR does not call the procedure k
   call/cc returns the value of [EXPR]
4. If k is invoked (ie. **(k [EXPR2]**) at any point, anywhere, program “rewinds” to the state when call/cc was invoked, and call/cc now returns the value of [EXPR2].



Example

```
(+ 2 (call/cc
       (lambda (k)
         (* 5 (k 4)))))
```

=> 6

Letting other places invoke k

Although k may never be explicitly invoked, it may be saved somewhere else for later use (or reuse!)

```
> (define retry #f)
> (define factorial
    (lambd (x)
      if (= x 0)
        (call/cc (lambda (k) (set! retry k) 1))
        (* x (factorial (- x 1))))))
```

```
> (factorial 4)
24

> (retry 1)
24

> (retry 2)
48
```

More places to invoke k

```
> (define writeNtimes (lambda (n)
      (or (call/cc (lambda (k)
                     (lambda () (k #f))))
          (and (> n 0)
               (display “Foo”)
               (newline)
               (writeNtimes (- n 1))))))

> (let ((m (writeNtimes 3))) (if m (m) (void)))
Foo n is 3
Foo n is 2
Foo n is 1
```