Functional Programming and Scheme

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

Recursion and Induction

Recursion

- Computationally: a procedure/function calls itself
- Semantically: defining a computation/value inductively

Induction

- Natural number: i) $0 \in \mathbb{N}$ ii) if $\mathrm{n} \in \mathbb{N}$, then $(\mathrm{n}+1) \in \mathbb{N}$
- Factorial: i) $0! = 1$ ii) $\mathrm{n}! = n * (n-1)!$

Important to think inductively in Scheme

Fibonacci Number

Numbers in the following sequence:

1 1 2 3 5 8 13 21 …

Inductive definition

- fib(1) = 1
- fib(2) = 1
- fib(n) = fib(n-1) + fib(n-2)

```
(define (fib n)
    (cond ((= n 1) 1)
          ((= n 2) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))
```

List

A list is a sequence of expressions in parentheses

`(1 2 3)` `(sqrt x)` `(x 1 “abc”)`

Programs are just lists interpreted as code

A list is either

- Empty: ()
- or non-empty: it has a ***head*** and a ***tail***, where
  the ***head*** can have any type
  the ***tail*** is itself a list

Inductive definition!

List Construction

Inductive definition

- () is a list
- If `e` is an expression, `xs` is a list, `(cons e xs)` is a list

head

tail

"`cons`" is the built-in function for constructing lists

```
(1 2 3 4)
```

is equivalent to

```
(cons 1 (cons 2 (cons 3 (cons 4 ()))
```

List Destruction

`car` and `cdr` are the destructors for lists:

if `xs` is a non-empty list, then

`(car xs)` is the head

`(cdr xs)` is the tail

Algebraically:

```
(car (cons x xs)) = x
(cdr (cons x xs)) = xs
```

List Destruction

`(car '(1 2 3))` = `1`, which is the same as

`(car (cons 1 '(2 3))))` = `1`

`(cdr '(1 2 3))` = `(2 3)`, which is the same as

`(cdr (cons 1 '(2 3))))` = `(2 3)`

Additional List Functions

`list` creates a list from its arguments

```
(list 1 2 3)
```

`null?` checks if a list is empty

`list?` checks whether something is a list

List Example

Define `lstSum`, which returns the sum of a num list

```
(define (lstSum lst)
    (if (null? lst) 0
        (+ (car lst) (lstSum (cdr lst)))))
```

Recursion Over Lists

```
(define (f l)
  (if (null? l) (base-case)
      (recursive-case, using (car l),(cdr l))))
```

List

`(1 2 3)` `(sqrt x)` `(x 1 “abc”)`

“`cons`” adds an element to a list

“`car`” returns the head of a list

“`cdr`” returns the tail of a list

“`append`” concatenates two lists

```
append ’(1 2 3) ’(4 5 6)
```

Recursion Over Lists

```
(define (f l)
  (if (null? l) (base-case)
      (recursive-case, using (car l),(cdr l))))
```

Equality Test

`equal`: return `#t` if two expressions have the same structure and content

```
(equal? 1 1)
```

```
(equal? ’(1 2) ’(1 2))
```

```
(equal? 5 ’(5))
```

```
(equal? ’(1 2 3) ’(1 (2 3)))
```

```
(equal? ’(1 2) ’(2 1))
```

List Example

Define `subs`, which takes `a b l`, and replaces `a` with `b` in `l`

```
(define (subs a b lst)
    (if (null? lst) '()
        (let ((rest (subs a b (cdr lst))))
             (if (equal? (car lst) a)
                 (cons  b          rest)
                 (cons (car lst) rest)))))
```

Higher-Order Functions

In Scheme, function is a first-class value

(Functions can go wherever expressions go)

- Functions as formal parameters

```
(define (twice f x) (f (f x)))
```

- Functions as real parameters

```
(twice sqrt 16)
```

```
(twice (lambda (x) (* x x)) 2)
```

Higher-Order Functions

In Scheme, function is a first-class value

(Functions can go wherever expressions go)

- Functions as return values

```
(define (addN n) (lambda (m) (+ m n)))
```

```
((addN 10) 20)
```

```
(twice (addN 10) 20)
```