Functional Languages

The Programming Language Spectrum

- Imperative
  - Von Neumann (Fortran, Pascal, Basic, C)
  - Object-oriented (Smalltalk, Eiffel, C++, Java)
  - Scripting languages (Perl, Python, JavaScript, PHP)

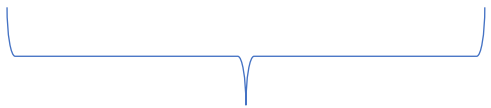
- Declarative
  - Functional (Scheme, ML, pure Lisp, FP)
  - Logic, Constraint-based (Prolog, VisiCalc, RPG)

Imperative Computational Model

- The Turing machine computes in an imperative way, **by changing the values in cells of its tape**, just as a high-level imperative program computes by changing the values of variables.
- *Assignment statement*

Lambda Calculus

- Church's model of computing is called the lambda calculus.
- It is based on the notion of **parameterized expressions** (with each parameter introduced by an occurrence of the letter λ.)
- Lambda calculus was the inspiration for functional programming: **one uses it to compute by substituting parameters into expressions**, just as one computes in a high-level functional program by passing arguments to functions.
- Ex: λ(x) = x * x * x



cube function

Imperative vs. Functional

- Imperative languages
  - Compute principally through
    - Iteration
    - Side effects (i.e., the modification of variables)
  - Underlying formal model is often taken to be a Turing machine

- Functional languages
  - Computes principally through substitution of parameters into functions
    - Employ a computational model based on the recursive definition of functions
  - Underlying formal model is the lambda calculus

Functional Programming Concepts

- Recursion
  - In the absence of side effects, it provides the only means of doing anything repeatedly
- First-Class Values
  - A value in a programming language is said to have first-class status if it can be **passed as a parameter, returned from a subroutine, or assigned into a variable**.
  - Subroutines are first-class values in all functional programming languages.
- Higher-Order Functions
  - A **higher-order function** takes a function as an argument, or returns a function as a result.

Functional Programming Concepts

- **LISP** stands for **LIS**t **P**rocessing
- Pure Lisp is purely functional; all other Lisps have imperative features
- All early Lisps dynamically scoped
- Scheme and Common Lisp statically scoped
  - Common Lisp provides dynamic scope as an option for explicitly-declared *special* functions
  - Common Lisp now the standard Lisp

Functional Programming Concepts

- Scheme is a particularly elegant Lisp
- Other functional languages
  - ML
  - Miranda
  - Haskell
  - FP
- Haskell is the leading language for research in functional programming.

A Bit of Scheme

Programming in Scheme

- The simplest possible Scheme program is a single number.
- If you ask the Scheme system to process such a program, it will simply return the number to you as its answer.
  - `212`
    - *212*
  - `-18`
    - *-18*
  - `1/10`
    - *1/10*
  - `3.14`
    - *3.14*

Name

- There are many names already in place when we start up Scheme.
- Most are names for *procedures.*
  - `sqrt`
    - `<procedure:sqrt>`
  - `+`
    - `<procedure:+>`
  - `/`
    - `#<procedure:/>`

Procedures

- The way we use a procedure is to *apply* it to some values.
  - The procedure named `sqrt` can be applied to a single number to take its square root.
    - `(sqrt 9)`
      - `3`
  - The procedure named `+` can be applied to two numbers to add them.
    - `(+ 6 10)`
      - `16`

Procedures

- In every case, an application consists of a parenthesized list of expressions, separated by spaces.
- `(sqrt 9)` is an expression
  - Its value is the procedure to apply
- Applications are themselves expressions, so they can be nested:
  - `(sqrt (+ 6 10))`
    - The value of expression is 4
    - That is the value to which the procedure named `sqrt` is applied

Exercise 1

- What is the value of each of the following expressions?

```
a) (* (+ 5 3) (- 5 3))

b) (/ (+ (* (- 17 14) 5) 6) 7)
```

Exercise 1

- What is the value of each of the following expressions?

```
a) (* (+ 5 3) (- 5 3))                    16

b) (/ (+ (* (- 17 14) 5) 6) 7)            3
```

Exercise 1

- What is the value of each of the following expressions?

```
b) (/ (+ (* (- 17 14) 5) 6) 7)
```

It is customary to break complex expressions into several lines with indentation that clarifies the structure, as follows:

```
(/ (+ (* (- 17 14)
         5)
      6)
   7)
```

Exercise 2

- What is the value of each of the following expressions?

```
a) (+ 120 5 20)
```

Prefix Notation takes arbitrary number of arguments.

```
a) (* 2 4 5)
```

```
b) (+ (* 2 5) (- 10 6) )
```

Prefix Notation allows combinations to be nested.

Exercise 2

- What is the value of each of the following expressions?

```
a) (+ 120 5 20)                    245

b) (* 2 4 5)                       40

c) (+ (* 2 5) (- 10 6) )           14
```

Exercise 3

- What is the value of each of the following expressions?

```
a) ((+ 3 4))

b) (quote (+ 3 4))
```

Exercise 3

- What is the value of each of the following expressions?

```
a) ((+ 3 4))                                    Error!
```

application: not a procedure;
expected a procedure that can be applied to arguments
given: 7

```
b) (quote (+ 3 4))                              '(+ 3 4)
```

How to Name Things Ourselves?

- In Scheme, we do this with a *definition*, such as the following:
  - `(define ark-volume (* (* 300 50) 30))`
    - Scheme first evaluates the expression (* (* 300 50) 30) and gets 450000.
    - It then remembers that ark-volume is henceforth to be a name for that value.
    - `ark-volume`
      - `450000`
    - `(/ ark-volume 1000)`
      - `450`

Naming Examples

- `(define size 5)`

- `size`
  - `5`

- `(* 10 size)`
  - `50`

Reusable Methods

- Although naming allows us to capture and reuse the results of computations, it isn't sufficient for capturing reusable *methods* of computation.
- Ex: We want to compute the total cost, including a 5 percent sales tax, of several different items.
  - `(+ 1.29 (* 5/100 1.29))`
    - `1.3545`
  - `(+ 2.40 (* 5/100 2.40))`
    - `2.52`

Reusable Methods: Lambda Expression

- Alternatively, we could define a *procedure* that takes the price of an item (such as $1.29 or $2.40) and returns the total cost of that item.
- We can specify a method of computation by using a *lambda expression*.

Reusable Methods: Lambda Expression

- In our sales tax example, the lambda expression would be as follows:
  - `(lambda (x) (+ x (* 5/100 x)))`
    - `lambda`: keyword
    - A lambda expression has two parts: a parameter list and a body
      - The parameter list in the example: `(x)`
      - The body: `(+ x (* 5/100 x))`
- However, we don't evaluate lambda expressions in isolation. Instead, we apply the resulting procedure to one or more argument values:
  - `((lambda (x) (+ x (* 5/100 x))) 2.40)`
    - `2.52`

Lambda Expression as a Part of Definition

- We usually use a lambda expression as part of a definition.
- The lambda expression produces a procedure and **`define`** simply associates a name with that procedure.
- In maths: let square (x) = x * x
- In Scheme:
```
(define square
        (lambda (x) (* x x)))
```
  - `(square 3)`
  - `(define (square x) (* x x))`

Exercise 4

a) Create a name for the tax example by using `define` to name the procedure `(lambda (x) (+ x (* 5/100 x)))`.

a) Use your named procedure to calculate the total price with tax of items costing $1.29 and $2.40.

Exercise 4

a) Create a name for the tax example by using `define` to name the procedure `(lambda (x) (+ x (* 5/100 x)))`.

```
(define tax
        (lambda (x) (+ x (* 5/100 x))))
```

b) Use your named procedure to calculate the total price with tax of items costing $1.29 and $2.40.

- `(tax 1.29)`
  - `1.3545`
- `(tax 2.40)`
  - `2.52`

Example: Defining and Using a Procedure

- ```
  (define cylinder-volume
      (lambda (radius height)
          (*(* 3.1415927 (square radius))
              height)))
  ```
  - Parameter: `radius`, `height`
  - We have already given name `square` to procedure for squaring a number.
- ```
  (cylinder-volume 5 4)
  ```

  ```
  314.15927
  ```

Example: Defining and Using a Procedure

```
(define cylinder-volume
    (lambda (radius height)
        (*(* 3.1415927 (square radius))
            height)))
```

```
(cylinder-volume 5 4)
(* (* 3.1415927 (square 5)) 4)
(* (* 3.1415927 (* 5 5)) 4)
(* (* 3.1415927 25) 4)
(* 78.5398175 4)
314.15927
```

Logical Operators

- Primitive operators:
  - <
  - >
  - =
- Logical operators:
  - and
  - or
  - not

Logical Operators

- `(> 10 5)`
  - `#t`
- `(= 2 2)`
  - `#t`
- `(< 100 5)`
  - `#f`

Logical Operators

- Compound conditions are written by placing the logical operator (***and***, ***or***, & ***not***) before the simple logical expression(s).

- `(and (= x y)(< y z))`
- `(and (= 5 5) (> 10 2))`
- `(or (= x y)(> y z))`
- `(not (= 10 4))`

Conditional Expressions

```
(define (abs x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          ((< x 0) (- x))))

(define (abs x)
    (cond ((>= x 0) x)
          ((< x 0)(- x)))))
```

Conditional Expressions

```
(define (abs x)
     (if (< x 0)
         (-x)
       x)
)
```

(if <predicate> <consequent> <alternative>)

Exercise 5

- `(if (< 2 3) 4 5)`

- ```
  (cond
      ((< 3 2) 1)
      ((< 4 3) 2)
      (else 3))
  ```
  ```

Exercise 5

- `(if (< 2 3) 4 5)` 4

-
```
(cond
    ((< 3 2) 1)
    ((< 4 3) 2)
    (else 3))
```
3

Example: Logical Operators and Conditional Expressions

```
(define tax
  (lambda (income)
    (if (< income 10000)
     0
     (* 20/100 income))))
```

- (tax 30000)
  - 6000

Exercise 6

- Write a procedure to check whether a number is divisible by 4 or 5.

- Some numeric predicate functions:
  - `even?, odd?, zero?, negative?`
  - `(even? 10)`
  - `(zero? 5)`
- `modulo`: returns the remainder or signed remainder of a division, after one number is divided by another

Exercise 6

- Write a procedure to check whether a number is divisible by 4 or 5.

```
(define (divisible_by_four_or_five? x)
     (cond ((zero? (modulo x 4)) #t)
           ((zero? (modulo x 5)) #t)
           (else #f)))

(divisible_by_four_or_five? 9)
```

Exercise 7

- “Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400.”

- Some numeric predicate functions:
  - `even?, odd?, zero?, negative?`
  - `(even? 10)`
  - `(zero? 5)`
- `modulo`: returns the remainder or signed remainder of a division, after one number is divided by another

Exercise 7

- “Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400.”

```
(define (leap? year)
     (cond ((zero? (modulo year 400)) #t)
           ((zero? (modulo year 100)) #f)
           (else (zero? (modulo year 4)))))

(leap? 2023)
```

Dynamic Typing

- Multiple types are implicitly polymorphic
- ```
  (define min (lambda (a b)
          (if (< a b) a b)))
  ```
- ```
  (min 316 107)
  ```
  - 107
- ```
  (min 4.89123 7.11819)
  ```
  - 4.89123
  ```

Recursion

- Recursion is important because in the absence of side effects it provides the only means of doing anything repeatedly.
- In a recursive procedure, all roads must lead to a base case.

- Example: Factorial
  - To compute `n!`, for any number `n`, we'll compute `(n - 1)!` and then multiply by `n`.
  - `0! = 1`

```
(factorial 3)
(* (factorial 2) 3)
(* (* (factorial 1) 2) 3)
(* (* 1 2) 3)
(* 2 3)
6
```

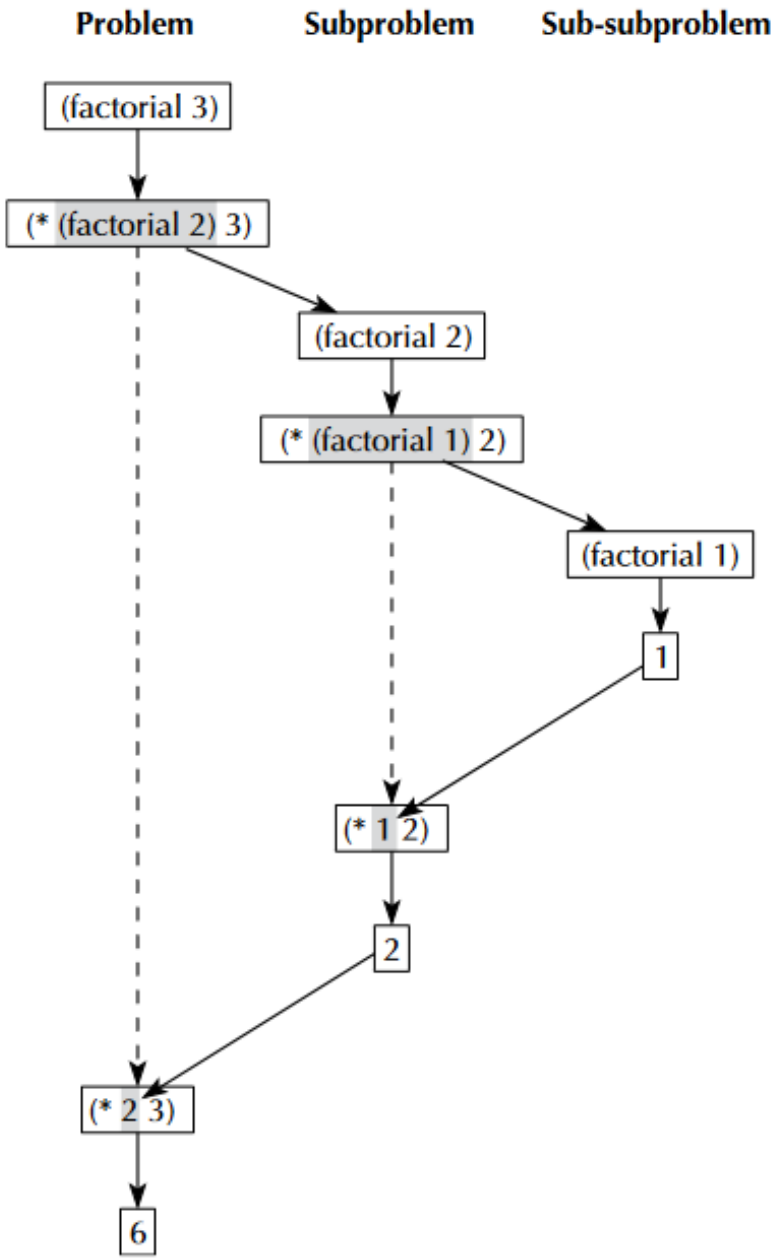


Figure 2.1 The recursive process of evaluating `(factorial 3)`.

Linear Recursion

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0)))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Linear Recursion

-
```
(define factorial
    (lambda(n)
     (if (= n 0)
         1
     (* (factorial (- n 1))
      n))))
```

- `(factorial 3)`
  - `6`

Linear Recursion

-
```
(define factorial
    (lambda(n)
     (if (= n 0)
         1
     (* (factorial (- n 1))
      n))))
```

- Tail recursion?

Linear Recursion

```
(define factorial_tail_recursion
    (lambda(n i)
     (if (= n 1)
         i
     (factorial_tail_recursion (- n 1)(* i n))
          )))
```

- (factorial_tail_recursion 3 1)
  - 6

Exercise 8

- Write a procedure called `power` such that `(power` base exponent`)` raises base to the exponent power, where exponent is a nonnegative integer. As explained in the sidebar on exponents, you can do this by multiplying together exponent copies of base. (You can compare results with Scheme's built-in procedure called `expt`. However, do not use `expt` in `power`. `Expt` computes the same values as `power`, except that it also works for exponents that are negative or not integers.)

Exponents

In this book, when we use an exponent, such as the k in x^k, it will almost always be either a positive integer or zero. When k is a positive integer, x^k just means k copies of x multiplied together. That is, $x^k = x \times x \times \cdots \times x$, with k of the x's. What about when the exponent is zero? We could equally well have said that $x^k = 1 \times x \times x \times \cdots \times x$ with k of the x's. For example, $x^3 = 1 \times x \times x \times x$, $x^2 = 1 \times x \times x$, and $x^1 = 1 \times x$. If we continue this progression with one fewer x, we see that $x^0 = 1$.

Exercise 8

- `(expt 6 3)`
  - `(* 6 6 6)`
- `(expt 6 2)`
  - `(* 6 6)`
- `(expt 6 1)`
  - `(* 6)`
- `(expt 6 0)`
  - `1`

```
(define power
 (lambda (x n)
   (if (= n 0)
       1
       (* (power x (- n 1))
        x )
   )
  )
 )
```

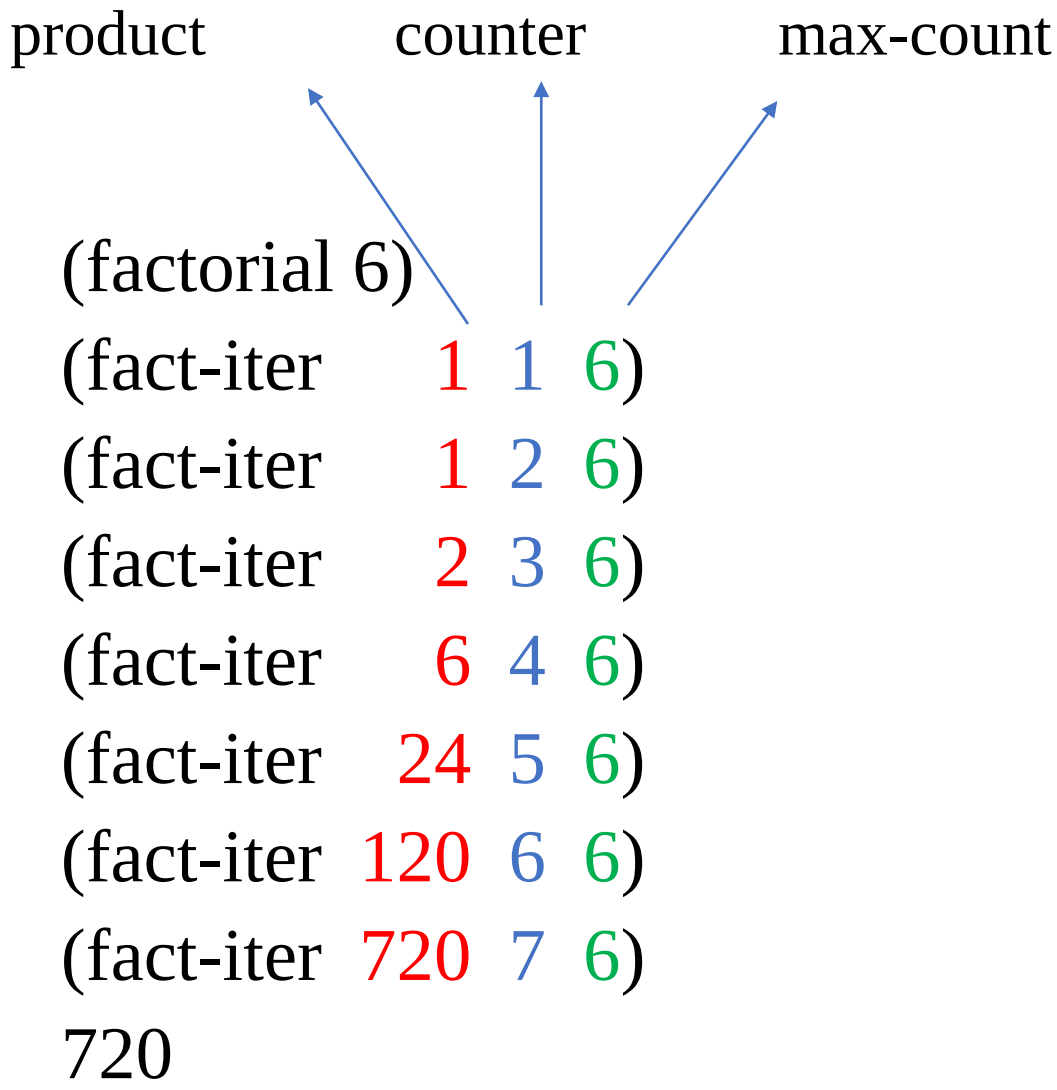
- `(power 6 3)`
  - `216`

Exercise 8

- Tail recursion?
  - Homework ☺

Linear Iteration

- Example: Factorial – (factorial n)
  - 0! = 1
  - 1! = 1
  - 2! = 1! . 2 = 1 . 2 = 2
  - 3! = 2! . 3 = 2 . 3 = 6
  - 4! = 3! . 4 = 6 . 4 = 24
  - 5! = 4! . 5 = 24 . 5 = 120
  - 6! = 5! . 6 = 120 . 6 = 720



Linear Iteration

```
product, counter = 1
do while counter < n
    product = counter * product
    counter = counter + 1
```

Linear Iteration

```
(define (factorial n)
     (fact-iter  1  1  n))

(define (fact-iter product counter max-count)
     (if  (> counter max-count)
          product
          (fact-iter (* counter product)
                     (+ counter 1)
                      max-count)))
```

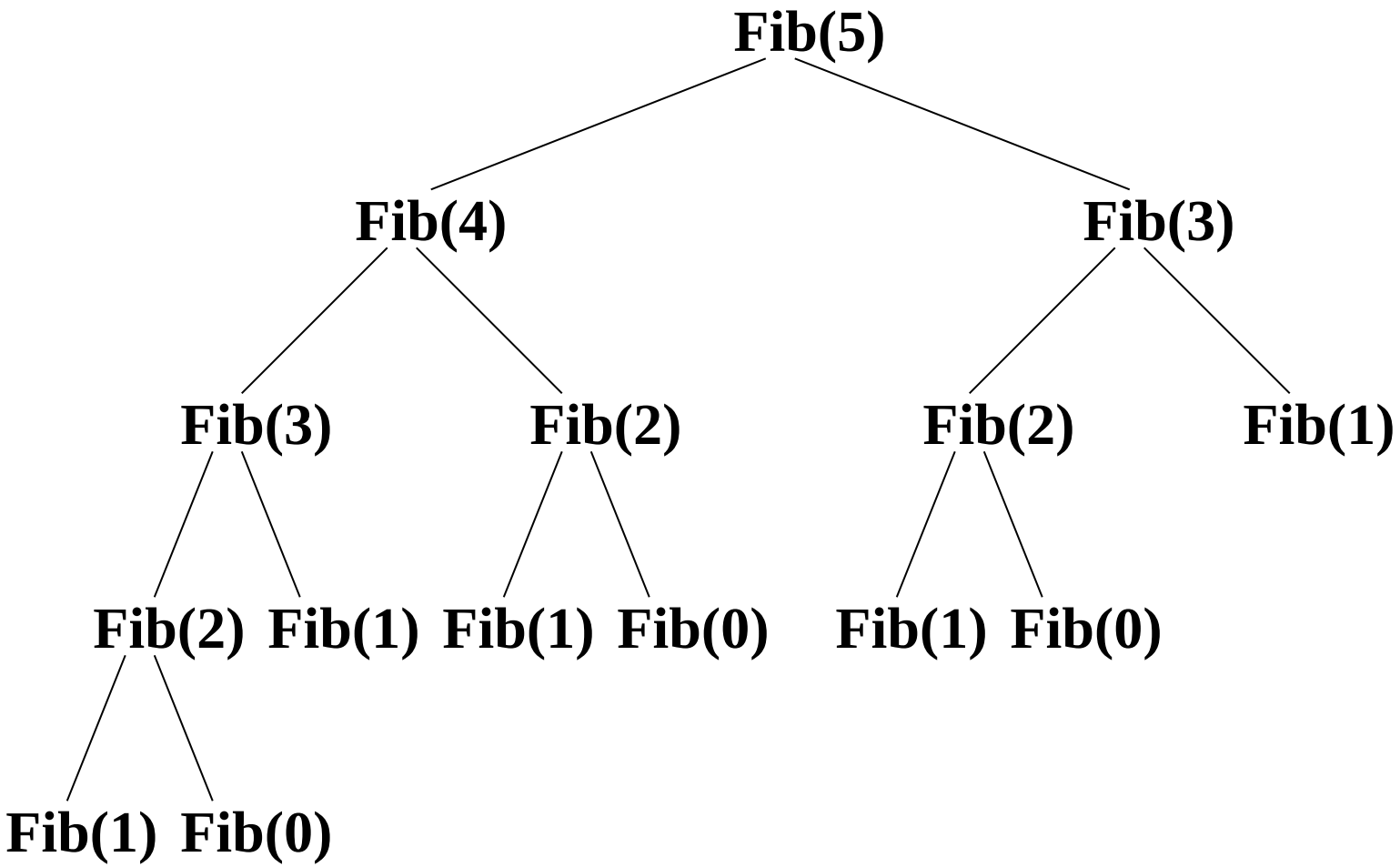
Tree Recursion

- Fibonacci numbers : 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$Fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{otherwise} \end{cases}$$

```
(define (fib n)
     (cond ((= n 0) 0)
           ((= n 1) 1)
           (else (+ (fib (- n 1))
                    (fib (- n 2))))))
```

Tree Recursion



Procedures as Parameters

- Ex: Sum of integers from *a* to *b*

Procedures as Parameters

- Ex: Sum of integers from *a* to *b*

```
(define (sum-integers  a  b)
     (if  (>  a  b)
          0
          (+  a  (sum-integers  (+ a 1)  b))))
```

Procedures as Parameters

- Ex: Sum of squares from *a* to *b*

Procedures as Parameters

- Ex: Sum of squares from *a* to *b*

```
(define (sum-squares  a  b)
     (if  (>  a  b)
          0
          (+  (square a) (sum-squres (+ a 1)  b))))
```

Procedures as Parameters

- Ex: Sum of cubes from *a* to *b*

```
(define (sum-cubes  a  b)
     (if  (>  a  b)
          0
          (+  (* a a a) (sum-cubes  (+ a 1)  b))))
```

Procedures as Parameters

- Ex: Sum of cubes from *a* to *b*

```
(define (sum-cubes  a  b)
     (if  (>  a  b)
          0
          (+  (cube a) (sum-cubes  (+ a 1)  b))))


(define (cube a)
     (* a a a))
```

Procedures as Parameters

```
(define (<name>  a  b)
     (if (>  a  b)
          0
          (+  (<term>  a) (<name>  (<next>  a)  b))))
```

Procedures as Parameters

```
(define (sum  term  a  next  b)
     (if  (>  a  b)
          0
          (+  (term  a)
              (sum  term  (next  a)  next  b))))


(define (sum-cubes  a  b)
     (define (inc x)
          (+  x  1))
     (sum  cube  a  inc  b))
```

Procedures as Parameters

```
(define (sum  term  a  next  b)
     (if  (>  a  b)
          0
          (+  (term  a)
              (sum  term  (next  a)  next  b))))
```

A ***higher-order function*** takes a function as an argument, or returns a function as a result.

```
(define (sum-cubes  a  b)
     (define (inc x)
          (+  x  1))
     (sum  cube  a  inc  b))
```

Internal Definitions

- `let` is used to create a binding list (a list of name-value associations), then evaluate an expression (based on the values of the names)
  - Temporary, local variables

```
(let   ((<var1>   <exp1>)
        (<var2>   <exp2>)
         …
        (<varN>   <expN>))
       <body>)
```

Internal Definitions

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

3

```
(let ((x 3)
      (y (+ x 1)))
      (+ x y))
```

x:undefined!

Exercise 9

```
(let
     ((x 2) (y 3))
     (let
          ((x 8)
          (z (+ x y)))
          (* z x)
     )
)
```

Exercise 9

```
(let                                                    40
     ((x 2) (y 3))
     (let
          ((x 8)
          (z (+ x y)))
          (* z x)
     )
)
```

Using *let* to Define Local Variables

- $f(x,y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y)$

Using *let* to Define Local Variables

- $f(x,y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y)$
- $a = 1 + xy$
- $b = 1 - y$
- $f(x,y) = xa^2 + yb + ab$

Using *let* to Define Local Variables

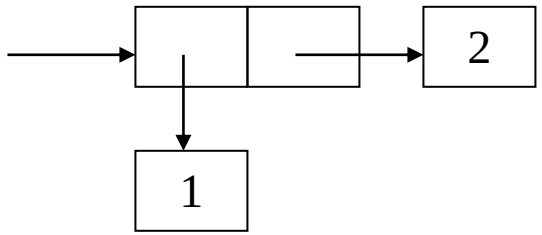
```
(define (f  x  y)
     (define  a  (+  1  (*  x  y)))
     (define  b  (-  1  y))
     (+  (*  x  (square  a))
         (*  y  b)
         (*  a  b)))
```

Using *let* to Define Local Variables

```
(define (f  x  y)
     (let ((a  (+  1  (*  x  y)))
           (b  (-  1  y)))
          (+  (*  x  (square  a))
              (*  y  b)
              (*  a  b))))
```

Pairs

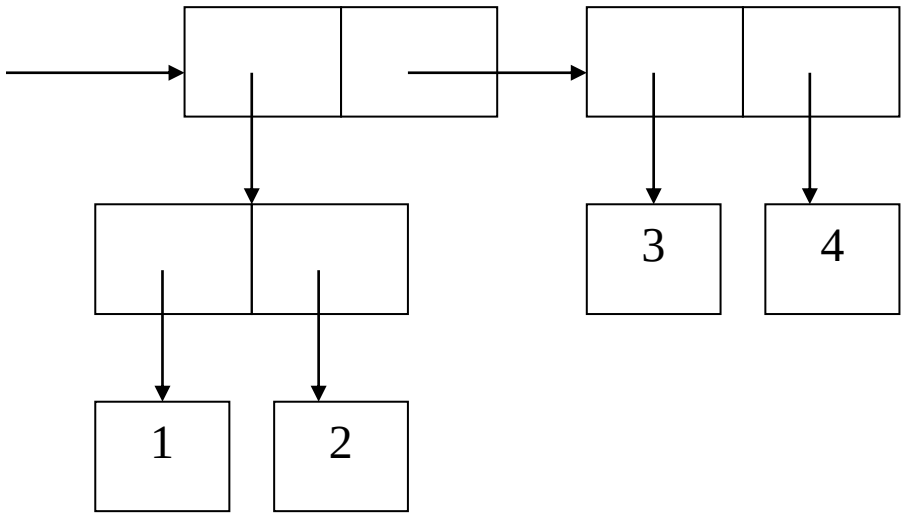
- Compound Structure
- <pair> constructor procedure: `cons <head> <rest>`
- `(cons 1  2)`



Box & Pointer Representation

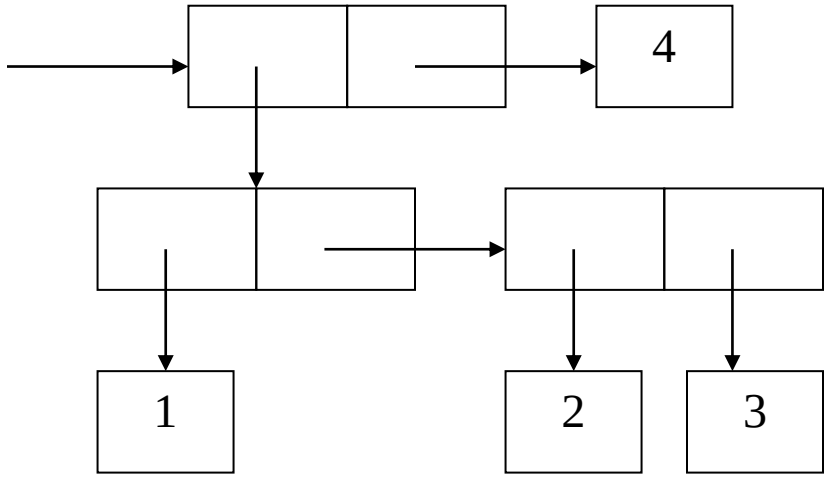
Pairs

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



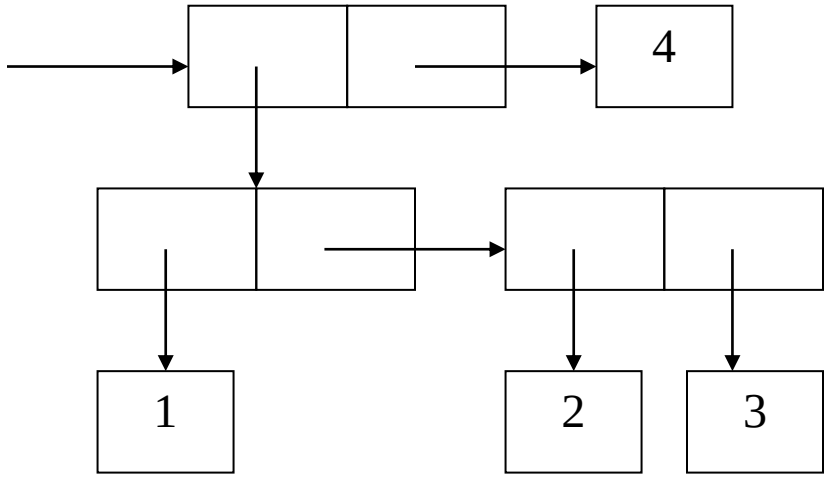
Pairs

- ?



Pairs

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



Pairs

- Compound Structure
- <pair> constructor procedure: `cons <head> <rest>`
- <head> extractor procedure: `car <pair>`
- <rest> extractor procedure: `cdr <pair>`

- `(car (cons 1 (cons 2 3)))` `1`
- `(cdr (cons 1 (cons 2 3)))` `'(2 . 3)`

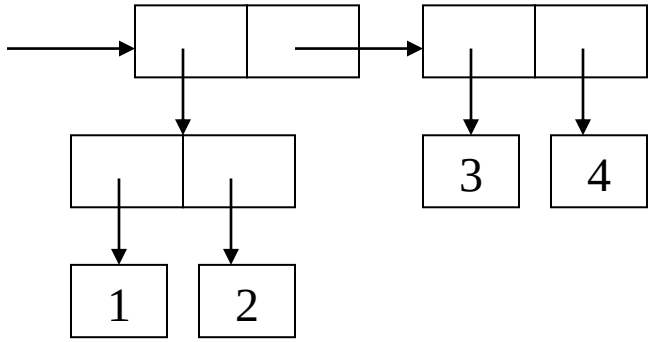
Pairs

- Compound Structure
- <pair> constructor procedure: `cons <head> <rest>`
- <head> extractor procedure: `car <pair>`
- <rest> extractor procedure: `cdr <pair>`

- `(car (cdr (cons 1 (cons 2 3))))` 2
- `(cadr <arg>) = (car (cdr <arg>))`
- `(cadr (cons 1 (cons 2 3)))` 2

Pairs

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



- `(define firstPair`
  `(cons (cons 1 2) (cons 3 4)))`

- `(car firstPair)` `'(1 . 2)`
- `(cdr firstPair)` `'(3 . 4)`

Lists

- Lists are important in functional languages because they have a **natural recursive definition** and are easily manipulated by operating on their first element and (recursively) the remainder of the list.
- Built in data type in Scheme

Lists

- The list function takes any number of values and returns a list containing the values:
- `(cons 1  2)` `(list 1  2)`
- `(list 0 1 2 3 4 5 6 7 8 9)`
  - `'(0 1 2 3 4 5 6 7 8 9)`
- `(list  'izmir  'istanbul  'izmit)`
  - `'(izmir istanbul izmit)`
- `(list 35 'izmir)`
  - `'(35 izmir)`

Lists

- List Operations
  - `(length (list 'a 'b 'c))` → `3`
  - `(list-ref (list  'a 'b 'c) 0)` → `a`
  - `(append (list 'a 'b 'c) (list 'd))` → `'(a b c d)`
  - `(member 'e (list  'a 'b 'c 'd))` → `#f`

Examples

```
(define  a  1)

(define  b  2)

(list  a  b)

(list  'a 'b)

(car  (list a  b 'c))

(cdr  (list a 'b 'c))
```

Examples

```
(define  a  1)

(define  b  2)

(list  a  b)                          (1  2)

(list  'a 'b)                         (a  b)

(car  (list a  b 'c))                  1

(cdr  (list a 'b 'c))                 (b  c)
```

Exercise 10

- Write a procedure that gets two lists of integers of the same size and returns true when each element in the first list is less than the corresponding element in the second list. For example

```
(list-< '(1 2 3 4) '(2 3 4 5))                    #t
```

- e10_v01.scm

Exercise 10 – v01

```
(define list-<
  (lambda (l1 l2)
    (cond ((null? l1) #t)
    (else
     (if (< (car l1) (car l2))
         (list-< (cdr l1) (cdr l2))
         #f)))))
```

Exercise 10

- Write a procedure that gets two lists of integers of the same size and returns true when each element in the first list is less than the corresponding element in the second list. For example

```
(list-< '(1 2 3 4) '(2 3 4 5))                    #t
```

  - What should happen if the lists are not the same size?
    - `(list-< '(1 2 3 4) '(2 3 4 5 6))`

- e10_v02.scm

Exercise 10 – v02

```
(define list-<
  (lambda (l1 l2)
    (cond ((not (= (length l1) (length l2)))
           "Lenght sizes are different!")
          ((null? l1) #t)
    (else
     (if (< (car l1) (car l2))
         (list-< (cdr l1) (cdr l2))
         #f)))))
```

Exercise 11

- Generalize the previous procedure to one called `lists-compare?`.
- This procedure should get three arguments; the first is a predicate that takes two arguments (such as <) and the other two are lists. It returns true if and only if the predicate always returns true on corresponding elements of the lists.
- We could redefine `list-<` in the following manner:
- ```
  (define list-<
      (lambda (l1 l2)
          (lists-compare? < l1 l2)))
  ```
- `e11.scm`
  ```

Exercise 11

```
(define lists-compare?
  (lambda (compare l1 l2)
    (cond ((not (= (length l1) (length l2))) "Lenght sizes are
different!")
          ((null? l1) #t)
    (else
     (if (compare (car l1) (car l2))
         (lists-compare? compare (cdr l1) (cdr l2))
         #f)))))

(define list-<
  (lambda (l1 l2)
    (lists-compare? < l1 l2)))
```

Exercise 12

- Write a Scheme function `fmember` that takes an atom and a simple list; return `#t` if the atom is in the list; `#f` otherwise.

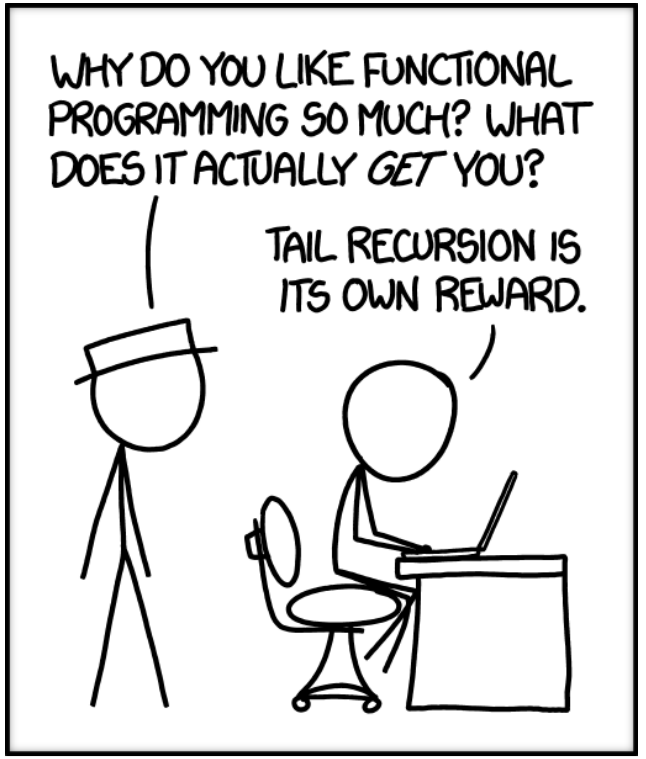
  - Predicate function `eq?` takes two parameters; it returns `#t` if the two are the same; otherwise `#f`
    - `(eq? 'A 'A)` `#t`
    - `(eq? 'A 'B)` `#f`

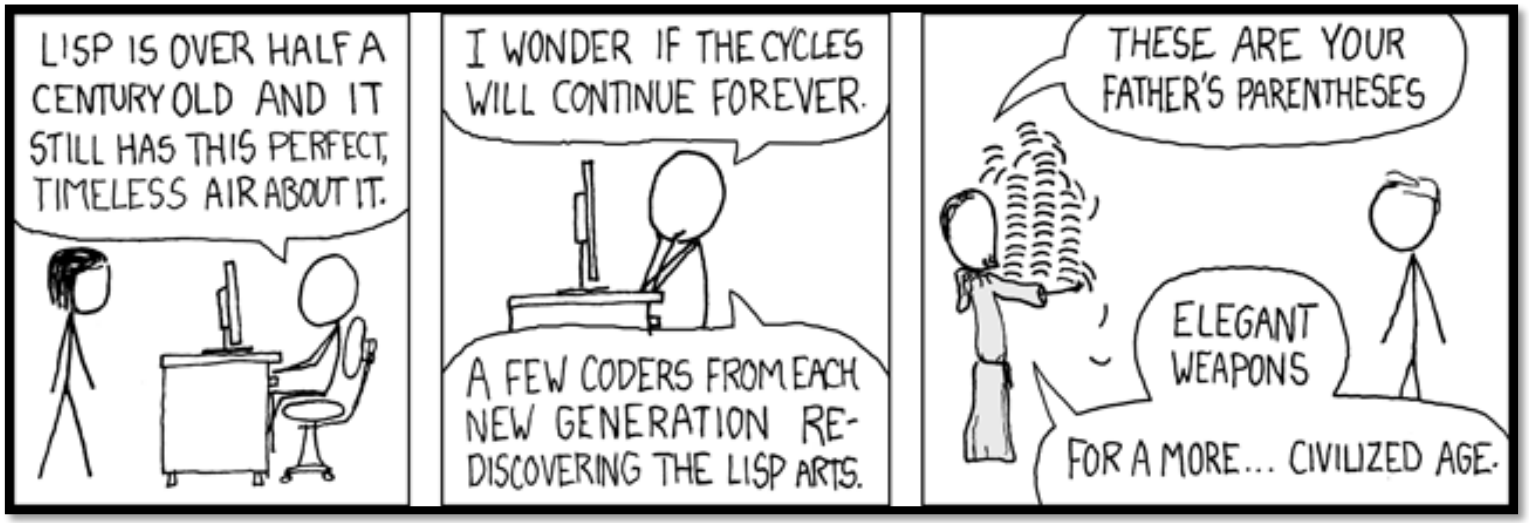
- `e12.scm`

Exercise 12

```
(define (fmember atm a_list)
     (cond
          ((null? a_list) #f)
          ((eq? atm (car a_list)) #t)
          (else (fmember atm (cdr a_list))))
     )
)
```

https://xkcd.com/1312/
https://xkcd.com/1270/
https://xkcd.com/297/





References

- M. L. Scott, Programming Language Pragmatics (4th Edition), Morgan Kaufmann, 2016.
- R. W. Sebesta, Concepts of Programming Languages (11th Edition), Pearson, 2016.
- M. Hailperin, B. Kaiser, and K. Knight, Concrete Abstractions An Introduction to Computer Science Using Scheme, 1999. http://gustavus.edu/+max/concrete-abstractions-pdfs/index.html.