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)

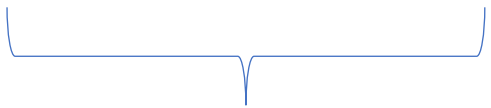
- Declarative
  - Functional (Scheme, ML, pure Lisp, FP)
  - Logic, Constraint-based (Prolog, VisiCalc, RPG)

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
```

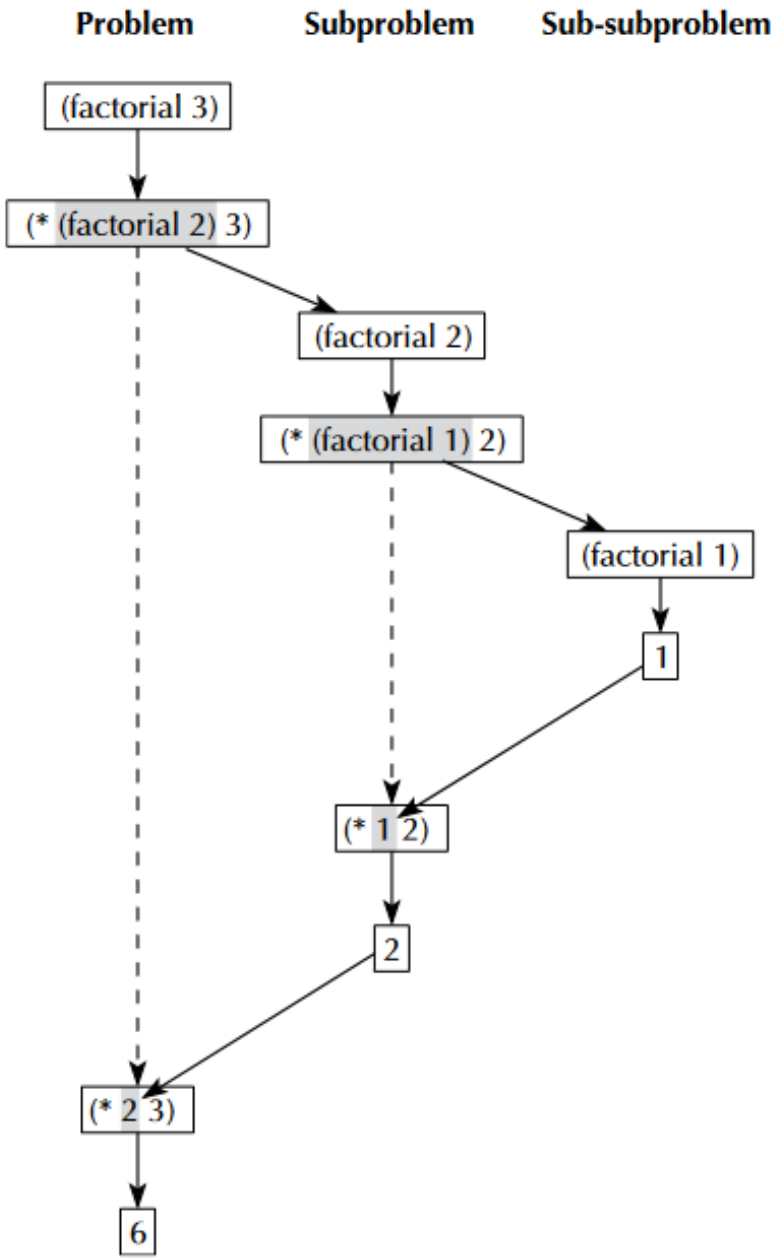


Figure 2.1 The recursive process of evaluating `(factorial 3)`.

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`

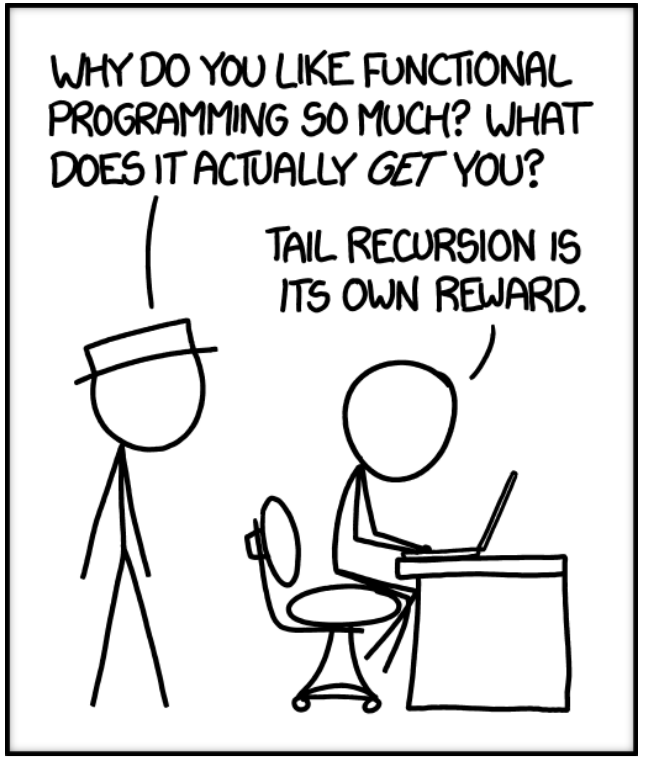
```
(define power
 (lambda (x n)
   (if (= n 0)
       1
       (* (power x (- n 1))
        x )
   )
  )
 )
```

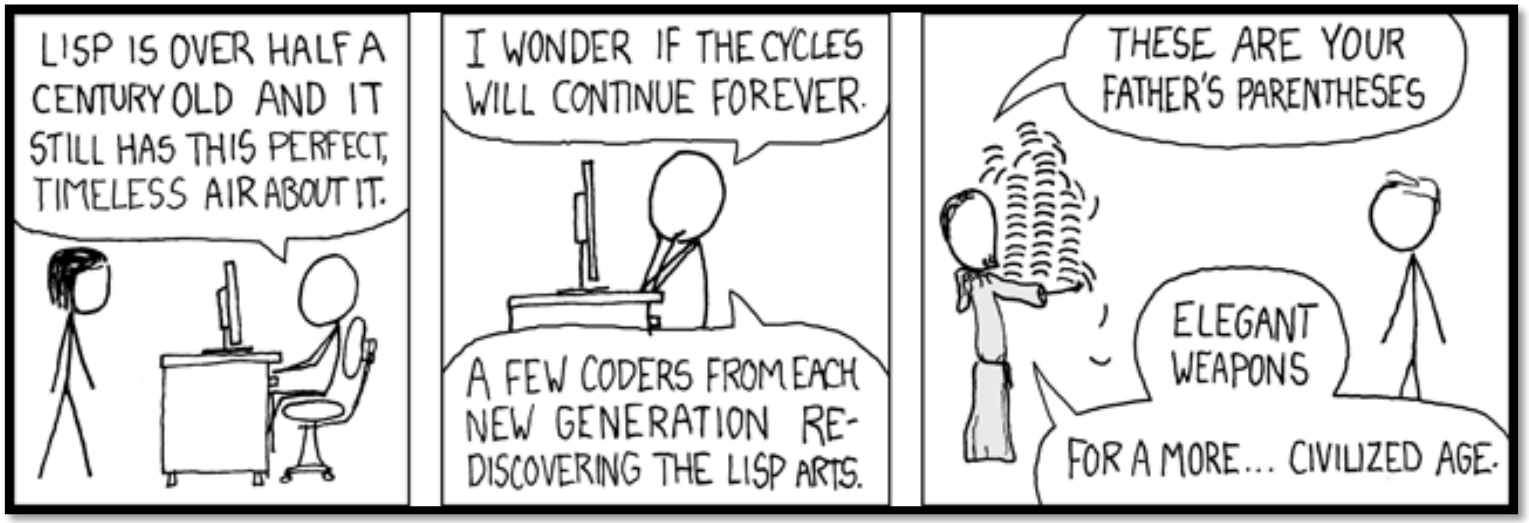
- `(power 6 3)`
  - `216`

Exercise 8

- Tail recursion?
  - Homework ☺

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.