Functional Programming and the Scheme Programming Language

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

September 9, 2020

Table of Contents

1 Functional Programming

2 Scheme

3 Preview of Monday's Lecture

4 Lab 2

Table of Contents

1 Functional Programming

2 Scheme

3 Preview of Monday's Lecture

4 Lab 2

Early procedural programming languages were built from a standpoint of making programming simpler compared to programming in assembly language.

Structured programming identified patterns in programs written in procedural programming language.

Certain principles of structured programming (e.g., no `GO TO`) made it easier to reason about programs in a more mathematical way.

From another point of view, we showed in Lecture 4 the lambda calculus, which can express any computable mathematical function.

It is possible to write programs in the lambda calculus, but it is too low level to comfortably write full-fledged programs (e.g., no numbers, no named functions, no strings, etc.).

Is it possible to write programs in a form that is inspired by the lambda calculus, but with "syntactic sugar" that helps programmers more easily write programs?

Is it possible to write programs in a form that is inspired by the lambda calculus, but with "syntactic sugar" that helps programmers more easily write programs? Yes.

What is Functional Programming?

Functional programming is a paradigm that emphasizes the following design and language characteristics:

What is Functional Programming?

Functional programming is a paradigm that emphasizes the following design and language characteristics:

- (Ideally) no side effects.

What is Functional Programming?

Functional programming is a paradigm that emphasizes the following design and language characteristics:

- (Ideally) no side effects.
- Immutable variables and data structures.

What is Functional Programming?

Functional programming is a paradigm that emphasizes the following design and language characteristics:

- (Ideally) no side effects.
- Immutable variables and data structures.
- Higher-order functions.

What is Functional Programming?

Functional programming is a paradigm that emphasizes the following design and language characteristics:

- (Ideally) no side effects.
- Immutable variables and data structures.
- Higher-order functions.
- No distinction between statements and expressions.

What is Functional Programming?

Functional programming is a paradigm that emphasizes the following design and language characteristics:

- (Ideally) no side effects.
- Immutable variables and data structures.
- Higher-order functions.
- No distinction between statements and expressions.
- A preference for recursion over iteration.

Side Effects

A side effect is a change in state that is not localized to the function.

Examples

```
public int linearSearch(int[] items, int key) {
  int found = -1;
  for (int i = 0; i < items.length && found == -1; i++) {
    if (items[i] == key)
      found = i;
  }
  return found;
}
```

Does the function `linearSearch` have side effects?

Examples

```
public int linearSearch(int[] items, int key) {
  int found = -1;
  for (int i = 0; i < items.length && found == -1; i++) {
    if (items[i] == key)
      found = i;
  }
  return found;
}
```

Does the function `linearSearch` have side effects? No, because all modified state is local to `linearSearch`.

Examples

```
# Python example
found = False

def linearSearch(items, key):
  for item in items:
    if item == key:
      found = True
```

Does the function `linearSearch` have side effects?

Examples

```
# Python example
found = False

def linearSearch(items, key):
  for item in items:
    if item == key:
      found = True
```

Does the function `linearSearch` have side effects? Yes, because it modifies the global variable `found`.

Examples

```
public void printBinaryTree(Tree tree) {
  if (tree) {
    printBinaryTree(tree.left);
    System.out.println(tree.contents);
    printBinaryTree(tree.right);
  }
}
```

Does the function `printBinaryTree` have side effects?

Examples

```
public void printBinaryTree(Tree tree) {
  if (tree) {
    printBinaryTree(tree.left);
    System.out.println(tree.contents);
    printBinaryTree(tree.right);
  }
}
```

Does the function `printBinaryTree` have side effects? Yes, because `println` modifies the state of the I/O console.

Examples

```
public int findNextPrime(int n) {
  int prime = -1;
  for (int i = n + 1; prime == -1; i++) {
      if (isPrime(i))
          prime = i;
  }
  return prime;
}
```

Does the function `findNextPrime` have side effects?

Examples

```
public int findNextPrime(int n) {
  int prime = -1;
  for (int i = n + 1; prime == -1; i++) {
      if (isPrime(i))
          prime = i;
  }
  return prime;
}
```

Does the function `findNextPrime` have side effects? It depends on whether `isPrime` has side effects.

Why Avoid Side Effects?

Why Avoid Side Effects?

- Makes functions easier to test by making functions just a mapping between inputs and outputs.

Why Avoid Side Effects?

- Makes functions easier to test by making functions just a mapping between inputs and outputs.
- Makes parallelization easy (no shared state means no synchronization worries)

Why Avoid Side Effects?

- Makes functions easier to test by making functions just a mapping between inputs and outputs.
- Makes parallelization easy (no shared state means no synchronization worries)
- Makes it easier to mathematically prove the properties of functions.

Are side effects completely avoidable?

Are side effects completely avoidable? No

When writing real-world programs, we still need to access I/O devices, operating system resources, databases, GUI elements, etc. All of these accesses are side effects.

Thus, we cannot completely avoid side effects when programming.

However, we can organize the functions in our code in such a way where we minimize and localize the use of side effects (for example, separating I/O code from processing logic).

Immutable Variables and Data Structures

In functional programming, all variables and data structures are immutable. The value assigned to a variable is permanent.

Wait, so what about lists? How do I add a new element to a list?

Wait, so what about lists? How do I add a new element to a list? By returning a new list that is the concatenation of the old list and a one-element list with the new element.

This is not functional code:

```
public int findNextPrime(int n) {
  int prime = -1;
  for (int i = n + 1; prime == -1; i++) {
    if (isPrime(i))
      prime = i;
  }
  return prime;
}
```

This is not functional code:

```
public int findNextPrime(int n) {
  int prime = -1;
  for (int i = n + 1; prime == -1; i++) {
    if (isPrime(i))
      prime = i;
  }
  return prime;
}
```

The culprits are `i++` and `prime = i`.

This is functional code:

```
public int findNextPrime(int n) {
  return findNextPrimeHelper(n + 1);
}

private int findNextPrimeHelper(int n) {
  if (isPrime(n))
    return n;
  else
    return findNextPrimeHelper(n + 1);
}
```

Why have immutable variables and data structures?

Why have immutable variables and data structures? Just as in the case of avoiding side effects, we simplify the reasoning of programs when variables and data structures are immutable, especially when these variables and data structures are not local to the function that operates on them (and thus would be introducing side effects).

Higher-Order Functions

In functional programming languages, functions are first-class objects. This means functions can:

- Allocate their own functions (i.e., have embedded functions)
- Accept functions as parameters
- Return functions

Sorting Example

Suppose a library offered a sort function that implemented a sorting algorithm (such as quicksort) and allowed the user to specify how the sort algorithm will compare values.

Sorting Example – Object Oriented Approach

In Java, which is an object-oriented programming language, we can either make the objects that we sort implement the `Comparable` interface and overload the `compareTo` method, or we could write a class that implements the `Comparator` interface and overload the `compare` method.

Sorting Example – Functional Approach

In a functional programming language, we can provide the sort function a comparison function that works similar to a Java `Comparator`'s `compare` method.

Sorting Example – Functional Approach

```
int compare(Person p1, Person p2) {
   if (p1.surname.equals(p2.surname)) {
     return p1.firstname.compareTo(p2.firstname);
   } else return p1.surname.compareTo(p2);
}

List<Person> sortedPersons = sort(persons, compare);
```

Find Next Prime Example

```
public int findNextPrime(int n) {
  int findNextPrimeHelper(int n) {
    if (isPrime(n))
      return n;
    else
      return findNextPrimeHelper(n + 1);
  }
  return findNextPrimeHelper(n + 1);
}
```

Note that `findNextPrimeHelper` is inside of `findNextPrime`.

All Language Constructs are Expressions

- All language constructs are expressions.
- All functions evaluate to an expression.

Recursion over Iteration

- In functional programming, there is a preference for recursion over iteration when looping.
- Performing a `while` or `for` loop often requires incrementing or decrementing a variable, which cannot be done when variables are immutable.
- All iterative constructs can be expressed in terms of recursion.

Is the lambda calculus a functional programming language?

$$
\begin{aligned}
<\lambda expr> ::= \; & <var> \\
& | \lambda <var> . <\lambda expr> \\
& | (<\lambda expr> <\lambda expr>)
\end{aligned}
$$

Is the lambda calculus a functional programming language?

$$
\begin{aligned}
<\lambda expr>::= \; & <var> \\
& |\lambda <var> . <\lambda expr> \\
& |(<\lambda expr> <\lambda expr>)
\end{aligned}
$$

Yes, since:

1. There are no side effects.
2. All variables are immutable.
3. There is support for higher-level functions (through lambda calculus abstractions).
4. All language constructs are expressions.
5. The lambda calculus supports recursion.

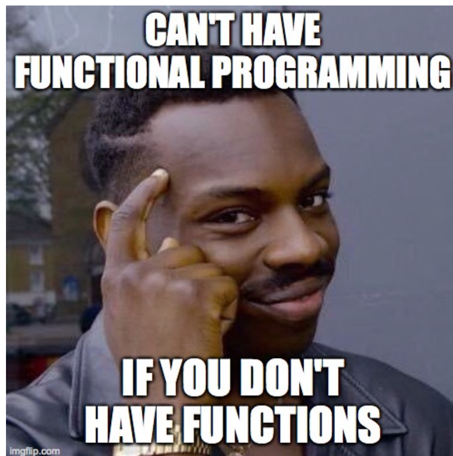
Is the Arith programming language a functional programming language?

```
<t> ::= true
      | false
      | if <t> then <t> else <t>
      | 0
      | succ <t>
      | pred <t>
      | iszero <t>
```

Is the Arith programming language a functional programming language?

```
<t> ::= true
      | false
      | if <t> then <t> else <t>
      | 0
      | succ <t>
      | pred <t>
      | iszero <t>
```

No, because there are no functions in the language.


CAN'T HAVE
FUNCTIONAL PROGRAMMING
IF YOU DON'T
HAVE FUNCTIONS
imgflip.com

In the real world, however, many functional languages are "impure"; they do support side effects and some of them have support for mutable variables and data structures.

Families of Functional Programming Languages

- Dynamically-typed
  - Lisp and its relatives (e.g., Common Lisp, Scheme, Clojure)
- Statically-typed
  - Standard ML
  - Haskell
  - OCaml
  - F#
  - Scala

We will be covering Scheme as the main functional programming language in this course; however, after the midterm we will be covering statically-typed functional programming when we discuss type systems.

Table of Contents

1 Functional Programming

2 Scheme

3 Preview of Monday's Lecture

4 Lab 2

A Brief History of Lisp

- The first version of Lisp was implemented in Fall 1958 [John McCarthy, "History of Lisp," p. 7, 1979].
- John McCarthy's 1960 paper "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" defined the semantics of Lisp in terms of the lambda calculus.
- Though Lisp was released before ALGOL 60, it already had support for higher-order functions and recursion.
- Lisp is not a pure functional language; it had support for assignment statements and even `GO` statements (akin to Fortran's `GO TO`).
- Over time a culture of functional programming practices developed and became embraced by many Lisp programmers.

More Lisp History

- Lisp originated from MIT and was heavily used there, particularly by MIT's artificial intelligence researchers.
- During the 1960s and 1970s various incompatible implementations of Lisp emerged (e.g., Maclisp, Interlisp, Lisp Machine Lisp)
    - Lisp machines were entire computer architectures that were specially designed for running Lisp programs, and were available during the 1970s and 1980s.
- In the 1980s there was an effort to unify the various implementations of Lisp. This led to the development of the Common Lisp programming language, which is multi-paradigm and supports procedural, functional, and object-oriented programming.

Examples of Modern Languages in the Lisp Family

- Common Lisp
  - Industrial-strength, multi-paradigm Lisp
- Scheme
  - Popular Lisp for education and programming language research
- Clojure
  - Lisp that runs on the Java Virtual Machine and can take advantage of Java's libraries
- GNU EMACS Lisp
  - Used for extending the capabilities of GNU Emacs

Note that these languages are incompatible with each other; the differences between each of them are like the differences between C, Java, and JavaScript.

If Common Lisp can be thought of as a "maximal Lisp," then Scheme can be thought of as a "minimal Lisp," the distillation of Lisp.

Some Very Simple Scheme Expressions

```
5
#t
#f
-2.345
"Hello!"
'Hello
```

Symbols in Scheme

- A symbol is a primitive data type.
- Symbols are preceded with a ' (e.g., `'hello`).
- Symbols are case-sensitive in Scheme (e.g., `(eq? 'hello 'Hello)` returns `#f`)
- Please note that `'hello` and `hello` are NOT the same; the former is a symbol while the latter evaluates to the value assigned to `hello`.

Function Calls in Scheme

Scheme uses prefix notation for expressing function calls, just like the lambda calculus.

```
(+ 2 5)
(- 1 3)
(* 2 4)
(/ 10 5)
(+ 1 2 3 4 5)
(+ 2 (* 3 4))
(expt 2 (/ 1 2))
```

Defining Global Variables

We can use the `define` function to map variable names to values.

Definition

`(define name value)` maps the symbol `name` to `value`.

Example:

```
(define PI 3.141592)

; find the area of the circle with radius 10
(* PI (expt 10 2))
```

Defining Functions in Scheme

Definition

`(define (function-name arg1 arg2 ... argN) function-body)` defines the function `function-name` with arguments `arg1` to `argN` that evaluates to the expression `function-body`.

Example:

```
; find the area of the circle with radius r
(define (circle-area r)
  (* PI (expt r 2)))

; compute area of circle with r = 15
(circle-area 15)
```

Defining Functions in Scheme

Note that a function with no arguments will be defined as `(define (function-name) function-body)` and would be invoked as `(function-name)`.

Defining Functions in Scheme

Note that a function with no arguments will be defined as `(define (function-name) function-body)` and would be invoked as `(function-name)`.

Please note that

```
(define f x)
```

is NOT the same as

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

Defining Functions in Scheme

Note that a function with no arguments will be defined as `(define (function-name) function-body)` and would be invoked as `(function-name)`.

Please note that

```
(define f x)
```

is NOT the same as

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

The former defines a variable `f` and sets its value to `x`, while the latter defines a function `f` with no arguments that returns `x`.

Control Flow in Scheme

Definition (if Expression)

(if expr1 expr2 expr3) evaluates expr1 first. If expr1 evaluates to #t (true), then the if expression evaluates to expr2. Else, the if expression evaluates to expr3. Note that the presence of expr3 is optional.

Examples:

```
; compute n!
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Control Flow in Scheme

```
; return the absolute value of a number
(define (my-abs n)
  (if (< n 0)
      (* -1 n)
      n))
```

Control Flow in Scheme

Definition (cond Expression)

```
(cond (cond-expr1 action-expr1)
      (cond-expr2 action-expr2)
      ...
      (else action-exprN))
```

Beginning with `cond-expr1`, it evaluates the expression `cond-exprI`. If the condition evaluates to true, then the `cond` expression evaluates to `action-exprI`. When it reaches `else`, the `cond` expression evaluates to `action-exprN`.

Boolean Functions in Scheme

Scheme has the logical Boolean functions `and`, `or`, and `not`. `and` and `or` take an unlimited number of arguments. `and` and `or` are short-circuiting; they finish evaluating the first time they reach an argument that evaluates to false or true, respectively.

Looping in Scheme

Scheme offers a `do` loop, but recursion is the style that we will be doing in this course.

Table of Contents

1 Functional Programming

2 Scheme

3 Preview of Monday's Lecture

4 Lab 2

Lists in Scheme

Lists are the core data structure of Scheme and other programming languages in the Lisp family. In fact, the programs we write in Scheme are collections of S-expressions, which are mostly lists (except for primitive elements). Scheme programs can be thought of as abstract syntax trees. We will be spending a good deal of time on Monday covering lists in Scheme.

Other Very Important Scheme Topics

- Anonymous Functions with `lambda`
- Local Variables with `let` and `let*`
- Tail Recursion
- Map, Filter, and Fold Functions

Table of Contents

1 Functional Programming

2 Scheme

3 Preview of Monday's Lecture

4 Lab 2

Lab 2

- Lab 2 is a collection of Scheme problems to solve with varying levels of difficulty.
- For features that have not been discussed in class yet, please read up to Chapter 7 of *Teach Yourself Scheme in Fixnum Days*.
- Due Monday, September 21 at 11:59pm Pacific Daylight Time.