



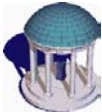
The University of North Carolina at Chapel Hill

COMP 144 Programming Language Concepts
Spring 2003

Introduction to Functional Programming

Stotts, Hernandez-Campos

1



Scope in ML

Top level environment has all pre-defined bindings

Every `val x = 5;` binding adds another row to the symbol table

Each row hides earlier bindings, does not destroy them

Local bindings can be made in functions definitions

Locals are removed from the symbol table when the function evaluation is complete

2



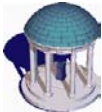
Functional Paradigm

Most issue from imperative languages remain

- What are the scope rules? Static? Dynamic?
- Binding issues, naming issues
- Types, expressions
- Scanning, parsing, semantic analysis still needed
- Runtime stack still needed (a big one)
- Control flow tends to be selection and recursion

Recursion is king, queen, and subjects

3

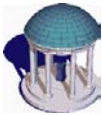


Functional Features

Most functional languages provide

- Functions as first-class values
- Higher-order functions
- List type (operators on lists)
- Recursion
- Structured function returns
- Garbage collection (dynamic memory)
- Polymorphism and type inference

4



Paradigm overlap

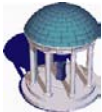
No hard dividing line

- I can write in a functional way in an imperative language
- Most functional langs have imperative subsets

What do you get in Java if you declare a class that has public data, and no methods?

```
class hmmm {  
    public int x;  
    public float y;  
    public String z;  
}
```

5



So Why Functional?

Truly teaches recursive algorithms

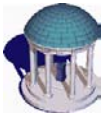
Easy polymorphism

- How do you write in Java a method that will operate equally properly on reals as well as strings? Say, a sort of a list?

Natural expressiveness for symbolic and algebraic computations

- Algorithms vs. system munging

6



History

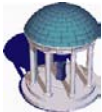
Lambda-calculus as semantic model (Church)

LISP (1958, MIT, McCarthy)

```
(defun fibonacci (n)
  (if (or (= n 0) (= n 1))
      1
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2))
        )
    )
  )
)
```

Lisp in AI

7



A Eureka! moment...

To the inventor of Lisp

To: jmc@cs.stanford.edu

Subject: your web pages

Years ago, I learned FORTRAN and Algol as my first programming languages, and when I was later exposed to Lisp I didn't "get" it. Much later, Samuel Kamin's "Programming Languages" book did much to clear up a lot of the confusion I experienced trying to understand the "why" of Lisp, but I still didn't appreciate Lisp for what it was. Some time later, I played with the problem of "if you have a computer and no software at all, what's the simplest way to create a complete programming language and development environment?". After toying around a few days with solutions for the problem, I suddenly realized I was trying to invent S-expressions and "eval"! The "aha!" hit me, and from then on I had a much better appreciation for the simplicity and elegance of Lisp.

8



History

LISP

- Dynamic scoping

Common Lisp (CL), Scheme

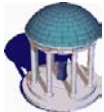
- Static scoping

ML, Haskell, Miranda

- Typing, type inference, pure functional

FP (Backus) sort of functional APL

9



LISP properties

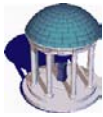
Homogeneity of programs and data

- programs are lists and a program can examine/change itself while running

Self-definition

- Easy to write a Lisp interpreter in Lisp (usually a class exercise)

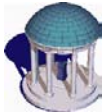
10



Can Programming Be Liberated from the von Neumann Style?

- This is the title of a lecture given by John Backus when he received the Turing Award in 1977
- He pointed out that programs should be abstract descriptions of algorithms rather than sequences of changes in the state of the memory
 - He called for raising the level of abstraction
 - A way to realize this goal is functional programming
- Programs written in modern functional programming languages are a set of mathematical relationships between objects
 - No explicit memory management takes place

11



Evaluation Order

- Functional programs are evaluated following a *reduction* (or evaluation or simplification) process
- There are two common ways of reducing expressions
 - Applicative order
 - » Impatient evaluation
 - Normal order
 - » Lazy evaluation

12



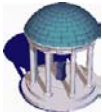
Applicative Order

- In applicative order, expressions are evaluated following the parsing tree (deeper expressions are evaluated first)

```
square (3 + 4)
= { definition of + }
  square 7
= { definition of square }
  7 * 7
= { definition of * }
  49
```

49 Normal Form

13



Normal Order

- In normal order, expressions are evaluated only their value is needed

```
square (3 + 4)
= { definition of square }
  (3 + 4) * (3 + 4)
= { definition of + applied to the first term }
  7 * (3 + 4)
= { definition of + applied to the second term }
  7 * 7
= { definition of * }
  49
```

14



Evaluation Order and Infinity

- Normal is sometimes more efficient than applicative order (why?)
- Applicative order can handle expressions that never converge to normal forms

```
fun looper x = x*looper(x):int;  
fun doit (flag,arg,func) =  
  if flag  
  then func(arg):int  
  else 1 ;  
fun do2 (flag,arg) = if flag then arg else 1 ;  
doit(true,5,looper) ;  
doit(false,5,looper) ;  
do2(false,looper(5)) ;
```

15



Haskell Evaluation Order

- Haskell is a *lazy* functional programming language
 - Expressions are evaluated in normal order
 - Identical expressions are evaluated only once

```
square (3 + 4)  
= { definition of square }  
  (3 + 4) * (3 + 4)  
= { definition of + applied to both terms }  
  7 * 7  
= { definition of * }  
  49
```

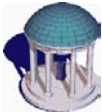
16



Values

- Expressions *denote* values
 - They are **not** values, but representations of them
 - E.g. `three infinity` denotes 3
`three infinity = 3`
- In functional programming, we denote an undefined value using the symbol \perp
 - \perp is pronounced *bottom*
 - E.g. `square infinity` denotes \perp
- A function f that satisfies $f \perp = \perp$ is said to be *strict*
- Otherwise, f is *nonstrict*

17

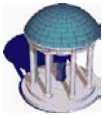


Functions

- Functions are the most important kind of value in functional programming
 - Functions are values!
- Mathematically, a function f associates an element of a set X to a unique element of second set Y
 - We write $f: X \rightarrow Y$

```
three    :: Integer -> Integer
infinity :: Integer
square   :: Integer -> Integer
smaller  :: (Integer, Integer) -> Integer
```

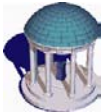
18



Functions

- A function $f: X \rightarrow Y$ is said to take *arguments* in X and return a *result* in Y
- Do not confuse a function f and an application of a function $f(x)$
 - We write $f\ x$ or $f(x)$ in ML
- Two functions are equal if they give equal results for equal arguments
 - This is the principle of *extensionality*

19



Reading Assignment

- Scott chapter 11
 - sections 1 (and 2... don't sweat the Lisp)
- Class notes on ML

20