CSC312 Principles of Programming Languages : Functional Programming Language

Overview of Functional Languages

- They emerged in the 1960's with Lisp
- Functional programming mirrors *mathematical functions*: domain = input, range = output
- *Variables* are mathematical *symbols*: not associated with memory locations.
- Pure functional programming is *state-free*: no assignment
- *Referential transparency*: a function's result depends only upon the values of its parameters.

14.1 Functions and the Lambda Calculus

The function *Square* has **R** (the reals) as domain and range.

$$Square : \mathbf{R} \rightarrow \mathbf{R}$$

$$Square(n) = n^2$$

A function is *total* if it is defined for all values of its domain. Otherwise, it is *partial*. E.g., *Square* is total.

Lambda Calculus

A clean, concise way to express a function.

Example:

A square function expressed in Python:

definitions:

```
def squareFunction (x):
        y = x * x;
        return y;
```

invocation:

```
squareFunction (100)
```

The same function expressed in Lambda Calculus:

definitions:

$\lambda x . x * x$

invocation:

$(\lambda x . x * x)$ 100

specify the formal parameter

specify the function body

Definition of (Pure) Lambda Calculus

A lambda expression is a particular way to define a function:

$$\mathit{LambdaExpression} \rightarrow \mathit{variable} \mid (\, M\ N) \mid (\, \lambda\ \mathit{variable}\ .\ M\,)$$

$$M \rightarrow \mathit{LambdaExpression}$$

$$N \rightarrow \mathit{LambdaExpression}$$

$$\boxed{\lambda\, x\ .\ x * x}$$

( ) can be omitted when no confusion would be caused.

Examples

1. compute the area of a circle.

$$\lambda x . x*x*\pi$$

2. compute the area of a rectangle.

$$\lambda x . \lambda y . x*y$$

3. compute the factorial of n?

Will be answered in future classes.

Substitution

In (λ *x* . *M*), *x* is *bound*. Other variables in *M* are *free*.

A *substitution* of *N* for all occurrences of a variable x in *M* is written *M*[*x* ← *N*]. Examples:

$$
\begin{aligned}
&x[x \leftarrow y] = y \\
&(xx)[x \leftarrow y] = (yy) \\
&(zw)[x \leftarrow y] = (zw) \\
&(zx)[x \leftarrow y] = (zy) \\
&(\lambda x \cdot (zx))[x \leftarrow y] = (\lambda x \cdot (zx)) \\
&(\lambda x \cdot (zx))[y \leftarrow x] = (\lambda u \cdot (zu))[y \leftarrow x] = (\lambda u \cdot (zu))
\end{aligned}
$$

Definition of the substitution:

1. If the free variables in N have no bound occurrences in M, then the term M[x← *N*] is formed by replacing all **free** occurrences of *x* in *M* by *N*.
2. O.w., renaming the bound variables in M until meeting condition 1.

Beta Reduction

A *beta reduction* $((\lambda\, x\, .\, M)N)$ is a substitution of all bound occurrences of x in M by N:

$$((\lambda\, x\, .\, M)N) = M[x \leftarrow N]$$

E.g.

$$((\lambda\, x\, .\, x^2)5) \ = x^2\ [x \leftarrow 5] = 5^2$$

$$(\lambda\, x\, .\, x^2)[x \leftarrow 5] = \ (\lambda\, x\, .\, x^2)$$

The typical, intuitive way for function to get evaluated.

More examples

1. $(((xyz)[x \leftarrow 3])[y \leftarrow 4])[z \leftarrow 5] = 345$
2. $(xyz)[x \leftarrow y] = yyz$
3. $((\lambda x . \lambda y . x+y) [x \leftarrow 5])[y \leftarrow 6] = \lambda x . \lambda y . x+y$
4. $(\lambda x . \lambda y . x+y)\ 5 = \lambda y . 5+y$
5. $((\lambda x . \lambda y . x+y)\ 5)\ 6 = 5+6$

Function Evaluation

In pure lambda calculus, no built-in constants or functions.

So, $((\lambda\, x\ .\ x*x)5) = 5*5$. Not 25.

In applied lambda calculus, some built-in constants and functions. All functional languages are applied lambda calculus.

$(\lambda\, x\ .\ x*x)5 \ = 5*5 = 25.$

Lazy v.s. Eager Evaluation

Lazy evaluation = delaying argument evaluation in a function call until the argument is needed.

- *Advantage: flexibility*

Eager evaluation = evaluating arguments at the beginning of the call.

- *Advantage: efficiency*

$\boxed{\mathit{if}\ (= x\ 0)\ 1\ (1/x)}$ runtime error when eager evaluation.

Status of Functions

In imperative and OO programming, functions have different (lower) status than variables.

In functional programming, functions have same status as variables; they are *first-class entities*.

- *They can be passed as arguments in a call.*
- *They can transform other functions.*

Functional Form

A function that operates on other functions is called a *functional form*. E.g., we can define

$g(f, [x1, x2, \ldots]) = [f(x1), f(x2), \ldots]$, so that

$g(Square, [2, 3, 5]) = [4, 9, 25]$

Quick Review

- Functional Languages:
  - *State-free; referential transparency* (depends only upon the values of its parameters.)
  - *Functions are first-order entities*
- Lambda Calculus:
  - *Bound variables :* λx
  - *Substitution :* $M[x \leftarrow N]$
  - *Beta reduction:* $((\lambda x . M)\mathrm{N}) = \mathrm{M}[\mathrm{x} \leftarrow N]$

Functional Form

A function that operates on other functions is called a *functional form*. E.g., we can define

$g(f, [x1, x2, \ldots]) = [f(x1), f(x2), \ldots]$, so that

$g(Square, [2, 3, 5]) = [4, 9, 25]$

Contents

14.3 Haskell

A more modern functional language

Key distinctions from other functional languages (e.g., Lisp):

- Lazy Evaluation
- An Extensive Type System
- Cleaner syntax
- Notation closer to mathematics
- Infinite lists

Minimal Syntax

```
-- equivalent definitions of factorial comment
fact1 n = if n==0 then 1 else n*fact1(n-1)

fact2 n
        | n==0        = 1
        | otherwise   = n*fact2(n-1)

fact3 0 = 1
fact3  n = n * fact3(n - 1)
```

Compiler: ghc
ghci (interactive mode)

Available in our department machines.

Freely downloadable.

Infinite Precision Integers

Infinite precision integers:

```
> fact2 30
> 26525285981219105863630848000000
```

14.3.2 Expressions

Infix notation. E.g.,

5*(4+6)-2 -- evaluates to 48

5*4^2-2 -- evaluates to 78

… or prefix notation. E.g.,

(-) ((*) 5 ((+) 4 6)) 2

Operators

Precedence	Left-Associative	Non-Associative	Right-Associative
9	!, !!, //		.
8			**, ^, ^^
7	*, /, ‘div’, ‘mod’, ‘rem’, ‘quot’		
6	+, -	:+	
5		\\	:, ++
4		/=, <, <=, ==, >, >=, ‘elem’, ‘notElem’	
3			&&
2			||
1	», »=	:=	
0			$, ‘seq’

`--`	Start of comment line
`{-`	Start of short comment
`-}`	End of short comment
`+`	Add operator
`-`	Subtract/negate operator
`*`	Multiply operator
`/`	Division operator
	Substitution operator, as in `e{f/x}`
`^`, `^^`, `**`	Raise-to-the-power operators
`&&`	And operator
`||`	Or operator
`<`	Less-than operator
`<=`	Less-than-or-equal operator
`==`	Equal operator
`/=`	Not-equal operator
`>=`	Greater-than-or-equal operator
`>`	Greater-than operator
`\`	Lambda operator
`.`	Function composition operator
	Name qualifier
`|`	Guard and `case` specifier
	Separator in list comprehension
	Alternative in data definition (enum type)

`\`	Lambda operator
`.`	Function composition operator
	Name qualifier
`|`	Guard and `case` specifier
	Separator in list comprehension
	Alternative in data definition (enum type)

`++`	List concatenation operator
`:`	Append-head operator ("cons")
`!!`	Indexing operator
`..`	Range-specifier for lists
`\\`	List-difference operator
`<-`	List comprehension generator
	Single assignment operator in do-constr.
`;`	Definition separator
`->`	Function type-mapping operator.
	Lambda definition operator
	Separator in case construction
`=`	Type- or value-naming operator
`::`	Type specification operator, "has type"
`=>`	Context inheritance from class
`()`	Empty value in IO () type
`>>`	Monad sequencing operator
`>>=`	Monad sequencing operator with value passing
`>@>`	Object composition operator (monads)
`(..)`	Constructor for export operator (postfix)

`[` and `]`	List constructors, "," as separator
`(` and `)`	Tuple constructors, "," as separator
	Infix-to-prefix constructors
‘ and ‘	Prefix-to-infix constructors
’ and ’	Literal char constructors
`"` and `"`	String constructors
`_`	Wildcard in pattern
`~`	Irrefutable pattern
`!`	Force evaluation (strictness flag)
`@`	"Read As" in pattern matching

14.3.3 Lists and List Comprehensions

A *list* is a series of expressions separated by commas and enclosed in brackets.

The empty list is written [].

evens = [0, 2, 4, 6, 8] declares a list of even numbers.

evens = [0, 2 .. 8] is equivalent.

List Generator

A list comprehension can be defined using a *generator*:

```
moreevens = [2*x | x <- [0..10]]
```

The condition that follows the vertical bar says, "all integers x from 0 to 10."

The symbol <- suggests set membership ($\in$).

Infinite Lists

Generators may include additional conditions, as in:

```
factors n = [f | f <- [1..n], n `mod` f == 0]
```

This means "all integers from 1 to n that divide f evenly."

List comprehensions can also be infinite. E.g.:

```
mostevens = [2*x | x <- [0,1..]]
mostevens = [0,2..]
```

List Transforming Functions

Suppose we define evens = [0, 2, 4, 6, 8]. Then:

```
head evens          -- gives 0
tail evens          -- gives [2,4,6,8]
head (tail evens)   -- gives 2
tail (tail evens)   -- gives [4,6,8]
tail [6,8]          -- gives [8]
tail [8]            -- gives []
```

List Transforming Functions

The operator : concatenates a new element onto the head of a list. E.g.,

4:[6, 8] gives the list [4, 6, 8].

[6, 8]:4 -- illegal

The operator ++ concatenates two lists. E.g.,

[2, 4]++[6, 8] gives the list [2, 4, 6, 8].

4++[6, 8] -- illegal

[4]++[6, 8] -- [4,6,8]

List Transforming Functions

Here are some more functions on lists:

```
null []            -- gives True
null evens         -- gives False
[1,2]==[1,2]       -- gives True
[1,2]==[2,1]       -- gives False
5==[5]             -- gives an error (mismatched args)
type evens         -- gives [Int] (a list of integers)
```

14.3.4 Elementary Types and Values

Numbers	
integers	types Int (finite; like int in C, Java) and Integer (infinitely many)
floats	type Float
Numerical Functions	abs, acos, atan, ceiling, floor, cos, sin log,logBase, pi, sqrt
Booleans	type Bool; values True and False
Characters	type Char; e.g., \`a\`, \`?\`
Strings	type String = [Char]; e.g., "hello"

14.3.5 Control Flow

Conditional

else part is mandatory.

```
if x>=y && x>=z then x
else if y>=x && y>=z then y
        else z
```

Guarded command (used widely in defining functions)

```
| x>=y && x>=z      = x
| y>=x && y>=z      = y
| otherwise         = z
```

14.3.6 Defining Functions

A Haskell Function is defined by writing:

its *prototype* (name, domain, and range) on the first line, and its *parameters and body* (meaning) on the remaining lines.

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
   | x>=y && x>=z         = x
   | y>=x && y>=z         = y
   | otherwise            = z
```

Note: if the prototype is omitted, Haskell interpreter will infer it.

Iterative Factorial

```
factorial n = product [1 .. n]
```

Using Pattern Matching

```
mysum [ ]       = 0
mysum (x:xs)    = x + mysum xs
```

Functions are polymorphic

Omitting the prototype gives the function its broadest possible meaning. E.g.,

$\boxed{\text{max3:: Ord a=> a -> a -> a -> a}}$

```
max3 x y z
      | x>=y && x>=z      = x
      | y>=x && y>=z      = y
      | otherwise         = z
```

is now well-defined for any argument types:

```
> max3 6 4 1
6
> max3 “alpha” “beta” “gamma”
“gamma”
```

The member Function

```
member :: Eq a => [ a ] -> a -> Bool
member alist elt
    | alist == [ ]           = False
    | elt == head alist      = True
    | otherwise              = member (tail alist) elt
```

Pattern Matching

```
member  [ ]  elt      =  False
member (x:xs) elt  = elt == x || member xs elt
```

Re: the latter can also be written:

```
member (elt:xs) elt  = True
member (x:xs) elt    = member xs elt

member (x:xs) elt  = if  elt ==x then True
                         else member xs elt
```