Concepts of Programming Languages

Functional Programming

(in LISP/Scheme)

Prof. Dr. Ulrike Pado

About the Next Weeks

- General theme: AI programming languages
  - Based strictly on ideas from math/logic
  - Specific objectives and use cases
  - Innovative features
  - Examples of two non-imperative programming paradigms

About the Next Weeks

- Now: Functional Programming
- Next two weeks: Logical Programming

Functional Programming

- Built around the concept of functional application (from λ calculus)
- Uses expressions (evaluate to a value) instead of statements (alter machine state)
- Uses recursion instead of iterative loops

Today's Goals

- Introduce the basics of λ calculus
- Explain the difference between functional and imperative/object oriented programming
  - Recap recursion
- Take a closer look at functions: Named f., unnamed f., higher-order f.

Basics of λ Calculus

- Functions map arguments to results
- λ calculus is a mathematical notation for functions

$$\boxed{\lambda\, x.}\boxed{x+1}$$

- Defines variable x
- Defines function body (What will happen to an argument x to reach the result?)

Function Application

$$\lambda\, x.\ x+1$$

- Function is applied to argument

$$\lambda\, x.\ x+1\ \mathbf{4}$$

- Variable x is bound to argument, argument value is substituted for x everywhere

β *reduction:* $\lambda\, x.\ x+1\ \mathbf{4} \rightarrow 4+1 = 5$

Partial Application

- Application can be partial, leaving other variables unbound

$$\lambda x.\lambda y.x+y$$

$$\lambda x.\lambda y.x+y\ 4 \rightarrow \lambda y.4+y$$

- One more application is needed to create variable-free expression
- Order of variable binding follows order of λs, application is left-associative

$$\lambda x.\lambda y.x+y\ 4\ 1 \rightarrow 4+1$$

Function Variables

λ can define variables that are functions (brackets for clarity):

$$\lambda f.\lambda y.(f\ y)$$

$$\lambda f.\lambda y.(f\ y)\ \lambda x.x+1 \rightarrow \lambda y.\lambda x.x+1\ y$$

$$\rightarrow \lambda y.y+1$$

Functions taking function arguments are called higher-order functions

Example: Two function arguments

$\lambda f.\lambda g.\lambda x.\lambda y.\ (f\ x) + (g\ y)$

$\lambda f.\ \lambda g.\lambda x.\lambda y.\ (f\ x) + (g\ y)\ \lambda a.\ a+1$

$\rightarrow \lambda g.\lambda x.\lambda y.\ (\lambda a.\ a+1\ x) + (g\ y)$

$\rightarrow \lambda g.\lambda x.\lambda y.\ (x+1) + (g\ y)$

$\lambda g.\lambda x.\lambda y.\ (x+1) + (g\ y)\ \lambda b.\ b-2$

$\rightarrow \lambda x.\lambda y.\ (x+1) + (\lambda b.\ b-2\ y)$

$\rightarrow \lambda x.\lambda y.\ (x+1) + (y-2)$

λ Calculus and Programming

- λ calculus was developed in the 1930s to define what a computable function is
- λ calculus is itself Turing complete
  - Anything that is computable can be formulated in λ calculus
- Turing completeness is a desirable property for a programming language, and modern languages are Turing complete
  - LISP (developed in the early 1960s) was one of the first such programming languages!

Today's Goals

- Introduce the basics of λ calculus
- Explain the difference between functional and imperative/object oriented programming
  - Recap recursion
- Take a closer look at functions: Named f., unnamed f., higher-order f.

Functional Programming

- Uses expressions (evaluate to a value) instead of statements (alter machine state)
  - Expressions drop out of function evaluation: Return a value (think method calls in Java)
  - Statements modify memory (think variable assignment in Java)
- Uses recursion instead of iterative loops
  - There are no stateful variables for loop indices
  - if/then/else exists (truth condition can be evaluated)

Imperative Programming

Compute the factorial function n!

E.g., 4! = 1*2*3*4

```
factorial_n = 1;
for (i=1; i++; i<=n) {
     factorial_n = factorial_n*i;
     }
```

Functional Programming

Compute the factorial function n!

E.g., 4! = 4*3*2*1

```
factorial(n) {
    if(n>1) {
        n*factorial(n-1)
    }
    else { 1 }
}
```

Recursion

- Functions call themselves repeatedly
  - Powerful concept, cf. inductive proof
- Recursion stops once base case is reached
  - This means arguments passed to each function call have to change (or recursion will not end)

```
factorial(n) {
	if(n>1) {
		n*factorial(n)
	}
	else { 1 }}
```

Efficient Recursion

- Not all recursive functions are efficient to compute
- Running example: Reversing a list
  - `'(1 2 3)` becomes `'(3 2 1)`
  - First attempt needs helper function: `append`
    `(append '(1) '(2 3))` returns `'(1 2 3)`

Reversing Lists Inefficiently

```
(define slow-rev (lambda (L)
        (if (null? L)
            '()
            (append
        (slow-rev (cdr L)) (list (car L)))
        )))
```

Check in LISP interpreter using

```
,trace (slow-rev '(1 2 3))
```

Reversing Lists Inefficiently

- Each function call causes another call to append
- Append calls can only be completed once the list has been traversed completely
- Too many calls, too many open operations to keep in memory!

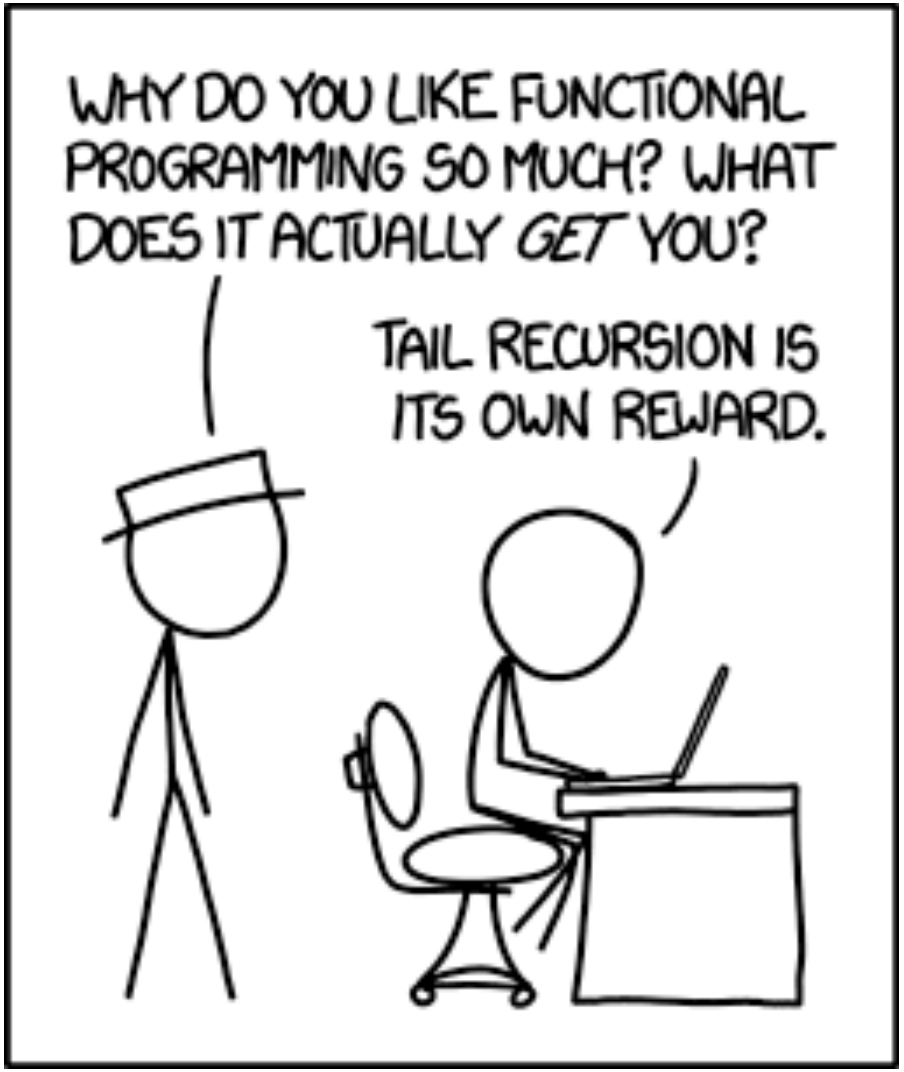
Reversing Lists Efficiently

```
(define list-rev (lambda (L A)
    (if (null? L)
        A
        (list-rev (cdr L) (cons (car L) A))
    )))
```

- Accumulator argument A collects intermediate results
- No open calls at the end of the input list!

Tail Recursion

- Tail recursion means that the recursive function call is the last computation necessary to complete the current call
- No “open” operations left on the stack that are waiting for recursive calls to return
- Faster and more memory-efficient
- Usually achieved by using accumulator argument and a wrapper function that hides accumulator from the user


WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY *GET* YOU?
TAIL RECURSION IS ITS OWN REWARD.

Today's Goals

- Introduce the basics of λ calculus
- Take a look at programming in Scheme
- Explain the difference between functional and imperative/object oriented programming
  - Recap recursion
- Take a closer look at functions: Named f., unnamed f., higher-order f.

A Closer Look at Functions

- Named functions work just like Java methods
  - Expect input arguments and return a value
  - Are called by name e.g. as `(plus 3 4)`
- Defining a named function can be troublesome overhead: Unnamed/anonymous functions

Unnamed Functions

```
(lambda (n)
   (if (= n 0) 0
         (if (> n 0) 1
               -1
         )
      )
   )
```

Higher-Order Function: Map

```
(map square `(1 2 3 4))
`(1 4 9 16)
```

- Map applies the given function to each list element

- No need to define “square”:

```
(map (lambda (x) (* x x)) '(1 2 3 4))
```

Higher-Order Function: Fold

```
(fold - 0 '(1 2 3))
```

```
 2
```

Calls:

```
(- 1 0) 1st list element - start argument
(- 2 1) 2nd list element - result so far
(- 3 1) 3rd list element - result so far
```

- Fold walks through the list from left to right, applying the given function to the next list member and the result so far
- Start argument (here: 0) is used for the first function application (where there is no “result so far”)

Higher-Order Function: Fold-right

```
(fold-right - 0 '(1 2 3))

 2
```

Calls:

```
(- 3 0)  last list element - start argument
(- 2 3)  middle element - result so far
(- 1 -1) first element - (negative) result
```

- Fold-right starts on the rightmost list element, applying the given function to the next list member and the result so far
- Start argument (here: 0) is used for the first function application

Higher-level Functions

- `map` and `reduce` take function arguments
- Powerful tool for concise programming
- Some flavor of higher-level functions exists in many modern languages