

## The von Neumann Architecture

- Otherwise known as the load/store architecture
  - Programs achieve their effect by changing the contents of a store location
  
- Are other architectures possible?

*IT Dept.*

## Functional Programming

- Programs achieve their effects by evaluating expressions
- Why?
  - There are no side effects in purely functional programming
  - It's much easier to prove correctness
  - It's much easier to parallelize – different expressions are done on different processors

*IT Dept.*

## Lisp

- Is it purely functional?
- Why or why not?

*IT Dept.*

## Lisp?

- Lisp stands for “Lots of Infuriating Superfluous Parentheses”

*IT Dept.*

## Lisp Syntax

- (+ 5 4)
  - A function followed by 2 atoms
  - Note: variables are atoms too
  
- (setq my-var 5)
- (+ my-var 5)

*IT Dept.*

## Quotes

- Things that are not to be evaluated are quoted:
  - (cons 'a '(b c d))

*IT Dept.*

## Example Code

```
(defun my-rest (a-list)
  "rest gives you all of the list, but the first element."
  (cdr a-list)
)
```

IT Dept.

## Lisp - variables

- All variables declared dynamically with a (setq var-name var-value) syntax will be **global** variables

IT Dept.

## Functions

- In Lisp
  - Functions may be used as arguments
    - mapcar is a good example of this

– Example: (mapcar '+ '(1 2 3 4) '(1 2 3 4))

IT Dept.

## S-expressions

- S-expression: a “symbolic expression” and it can be anything from an atom to a list to a function
- Examples:
  - (+ my-num 1)
  - (= @student 0)
  - my-var

IT Dept.

## Write a function

- Square – squares its argument
- Max – takes the max of its arguments
- Min – takes the min of its arguments

IT Dept.

- Define fourth power:
  - (defun fourth-power (x) (square (square x)))
- What happens here:
  - (defun powers-of (x) (square x) (fourth-power x))
- What is y's scope?:
  - (defun powers-of (x) (setq y (square x)) (fourth-power x))

IT Dept.

## Let

- Creating local variables:
  - `(let ((this 3) (that (- 67 34))) (* this that))`

IT Dept.

## If/Cond

- If:
  - `(defun absdiff (x y) (if (> x y) (- x y) (- y x)))`
- Cond:
  - `(defun absdiff (x y) (cond ((> x y) (- x y)) (t (- y x))))`

IT Dept.

## Recursion

- A recursive function:
  - `(defun power (x y) (if (= y 0) 1 (* x (power x (1- y)))))`
- How to trace it: `(trace power)`

IT Dept.

## Output

- Print:
  - `(print (* 2 3))`
- Formatting:
  - `(format t "~%omg!~%printing with ~%newlines.~%")`

IT Dept.

## Lisp

- Lisp is a dynamically typed language
  - This leads to greater flexibility at the cost of less error catching during compile time
  - Since we know less type checking is done, we can check it ourselves with predicates
    - Lisp has a lot of predicates for type checking
    - Some common predicates: `symbolp`, `atom`, `listp`, `floatp`, `functionp`,

IT Dept.

## Lambda Expressions

- An s-expression whose first element is "lambda" is a lambda expression. Lambda expressions are like anonymous functions
  - Examples
    - `(lambda (x) (* x 10))`
    - `((lambda (x y) (+ x y)) 2 3)`

IT Dept.

## Scope

- There are two kinds of scope here:
  - Lexical scope – what clisp uses and what we're used to
  - Dynamic scope – what eLisp uses

IT Dept.

## Scope Differences

- What does this code do?

- It depends on whether there is closure or not

```
(defun base (f a)
  ((lambda (f a)
    (+ (funcall f 10) a)
   ) f a))
```

```
(defun two-digit (a b) ((lambda (a b)
  (base '(lambda (c) (* a c)) b)
 ) a b))
```

```
(two-digit 2 3)
```

IT Dept.

## Other Languages

- FP: purely functional and not meant to be used as anything but a research language
- Scheme: a dialect of Lisp
- ML: uses static type checking, strongly typed, but if the compiler can figure out the type, then you don't have to put it

IT Dept.

## Lazy Evaluation

- In both ML and Lisp, parameters are called by value. It is sometimes useful for parameters to remain unevaluated until they're required.
- Infinite data structures become possible with lazy evaluation

IT Dept.