CSCI: 4500/6500 Programming Languages

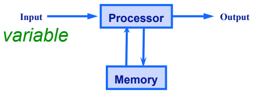
Functional Programming Languages

Part 1: Introduction

Thanks again to Profs David Evan's, University Virginia and Prof. Sebesta, author of our other book

---

Review: Language Perspectives



- **Imperative**: Mode of computation - *a variable (state)*
  - » Von Neumann Machines
    - – modify variables in memory
  - » Turing machines - imperative - changes values in cells (variables) on tape
- **Functional**: Mode of computation - *a function*
  - » Lambda calculus
  - » apply a function (a program) to transform its input (parameters) to output (result)
- **Relational**: Mode of computation - *constraints*
  - » programmer writes set of axioms that allow the computer to discover a constructive proof for a particular set of inputs

---

Functional Programming

- Do everything by using functions and evaluate them
  - » Great advantages:
    - – no side effects
    - – no mutable state
- Based on "mathematical functions"
  - » Historically from Church's model of computation called the lambda calculus (λ - calculus)
    - – Study of function application and recursion
- Example Languages: LISP, Scheme, FP, ML, Miranda and Haskell

---

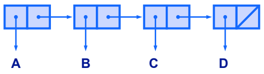
Functional programming: *Focus on Functions*

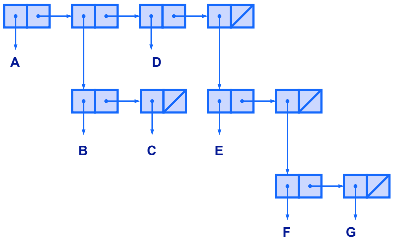
- First class objects:
  - » can be created during execution
  - » stored in data structures
  - » can be used as parameters or inputs to other functions
  - » can be returned
- Higher order functions:
  - » can take other functions as arguments
  - » and/or return function as results
  - » Basic building blocks of functional languages!

History: LISP first functional programming language

- LISt Processing Language (McCarthy (MIT) 1959)
  - » Processes data in lists
- Two objects (originally)
  - » atoms and lists
- Lists are delimiting their items in parenthesis.
  - » Simple list: (A B C D)
- Functions and data are represented in the same form, e.g.:
  - » (A B C) as data is a simple list of 3 atoms: A, B and C
  - » (A B C) as a function is interpreted as the function named "A" applied to two parameters, B and C: (+ 4 5)
    - – Cambridge Polish (*parenthesized* prefix notation)

LISP





- List forms parenthesized collection of *sub* lists and/or atoms:
- Stored as a linked list each node has two pointers
  - » First pointer to a representation of the element (e.g., symbol or number) or another sublist
  - » Second pointer next element of list
- Example:
  - » (A B C D)
  - » (A (B C) D (E (F G)))

Variants of LISP

- Pure (original Lisp)
  - » purely functional no imperative features (e.g., assignment statement)
  - » dynamically scoped (as all early versions of LISP) more on this later.
- All other Lisp's have some imperative features (e.g., variable, assignment)
- COMMON Lisp
  - » brought all LISPs under a common umbrella
    - – HUGE, and very complicated, provides dynamic scope as an option
- Scheme a mid-1970s dialect of LISP designed to be cleaner, more modern and simpler version than dialects of Lisps
  - » Statically scoped

Scope: A Preview

- Static scoping: variables always refers to its nearest enclosed binding (between name and object). Lexiographic -- Compile time
- Dynamic scoping: binding depends on the flow of control at run time and the order subroutines are called, refers to the closest active binding

```
a: integer    // global

procedure first()
  {
  a = 1       // global or local?
  }
procedure second
  {
  a: integer // local
  first()
  }
a = 2
if read_integer() > 0
  second()
else
  first()
write_integer(a)
```

```
Static:  prints 1 a is global scope
         of a is closest enclosed a, so
         for "first"'s a refers to global a
Dynamic: prints 1 or 2: if we go to second
         first, first's a refers to second's
         local a (closest active binding).
```

Introduction to Scheme

- Mid-1970s dialect to Lisp, designed to be cleaner, more modern and simpler than contemporary dialects of LIPS
- Uses static scoping
- Functions are first class entities
  - » Can be values of expressions and elements of a list
  - » Can be assigned variables and passed as parameters
- Have some imperative features (will not focus on these)

Scheme

- Is a collection of function definitions and lots of parenthesis.
  - » primitive functions (a form of an expression)
    - – +, - *
    - – (+ 3 4)
    - – ((+ 3 4)) -> error
      - Calls + with 3 and 4 as parameters, then call 7 as a 0 parameter function = a run time error
  - » A simple expression could just be value
    - – 5
    - – 5 is evaluated to be "5"

How do we create more complex functions?

- *Lambda* (λ) expressions
  - » (lambda (parameters) expression)
  - » (lambda (x) (* x x)
    - – is a nameless function that returns the square of its parameters (nameless don't need to use it again).
    - – can be applied like normally containing a list that contains the actual parameters
  - » ((lambda (x) (* x x)) 7). Here x is called a bound variables and does not change after being bound to a parameter (we can bind a name to a lambda expression too, by using define)
- ((lambda (a b) (if (< a b) a b)) 5 6)
- ((lambda (a b) (if (< a b) a b)) 6 5)

Give an expression a name: *"define"*

- Binds name to a value
  - » (define symbol expression)
  - » (define pi 3.14159)
- Binds a name to a *Lambda* (λ)
  - » expression is abbreviated (no word "lambda" is needed)
  - » takes two lists as parameters
    - – prototype of function
      - function name followed by formal parameters
    - – one or more expressions to which name is to be bound
  - » (define (function_name parameters) expression {expression})
  - » Example:
    - – (define (square number) (* number number))
    - – (square 5)
      - displays 25

Haskell Curry:
Combinatory Logic
(precursor of lambda
calculus). Combinator –
higher order function

Currying

- Transforms a multiple argument function so that it can be called as a chain of functions each with a single argument.
  - » Example: Allows languages to reduce the function (+ 1 4) [plus-one] to one argument. Pre apply +1 to the function and wait for the "4"
    - ++, -- (plus one with a single argument – "1" is removed as an argument.
  - » `( define curried-plus ( lambda ( a ) ( lambda ( b ) ( + a b ) ) ))`
    - `( ( curried-plus 1 ) 4 )  ;  chain here - one argument at a time.`
    - `( define plus-1 ( curried-plus 1 ) )`
    - `( plus-1 4 )`
  - » Idea: If you "fix" some arguments you get reduce the function arguments to only use the remaining arguments. Example:
    - y^x and fix y = 2 then you get the function of one variable 2^x.
- What is it really? An incomplete application of arguments to a function


turmeric


coriander


chile pepper


garlic

Examples: Currying

- `( define curried-plus ( lambda (a)  (lambda (b) (+ a b ) ) ) )`
- (curried-plus 3) : adds 3 to an argument b (not given yet)
  - » ((curried-plus 3) 4) => 7
- (define plus-three (curried-plus 3))
  - » (plus-three 4) => 7, (plus-three 5) => 8
- General purpose "function" that curries its (binary) arguments:
  - » `(define curry (  lambda ( f ) ( lambda (a ) (lambda (b) ( f a b )))))`
  - » `f` can be defined as addition '+' separately
    - `(define curried-plus (curry + ) )  -> ((curried-plus 3 ) 4 ) -> 7`
    - `(define curried-mult (curry * ) )  ->  ((curried-mult 3) 4) -> 12`

Currying

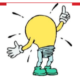
- Rewriting a function with multiple parameters as a composition of functions of one parameter
  - » plus = f(a, b) = a + b f(3, 2) = 5 (not curried)
  - » curried_plus = [f(b) => f(a) = a + b]
    - takes a single argument b and returns a function that takes a single argument 'b' and returns the results a + b
    - plus_one = curried_plus(1), and now
      - plus_one(5) returns 6 and plus_one(2) returns 3

Essential Scheme

```
Expression ::= ( Expression Expression* )
Expression ::= ( if Expression Expression
                    Expression )
Expression ::= ( define name Expression )
Expression := Primitive
Primitive  := number
Primitive  ::= + | - | * | / | < | > | =
Primitive  := … (many other )
```

Grammar is simple, just follow the replacement rules. What does it all mean?

Scheme: Functional programming



What is going on, really?

In General – 2 things (Evaluate and Apply):

- Evaluate the functions or the expressions then
- Apply the value of the first expression (a function) to the values of all the other expressions

Examples:

- (+ 655 58), (* 5 7 8), (-24 (* 4 3))

Evaluation: Expressions and Value

- Expression has a value (almost always)
- When an expression with a value is evaluated its value is produced
- How do we evaluate:
  - » primitives
  - » names
  - » applications (expression)

Evaluating: Primitives

- Primitives are *self evaluating*
  - » 2

    2
  - » #t

    #t
  - » +

    #<primitive:+>

Evaluating: Names

- Evaluates to the value associated with the name.

```
>(define two 2)
>two
2
```

Evaluating Applications

Evaluate:

all the sub expressions of the combination

- Apply the value of the first sub expression to the values of all the other sub expressions
  - » (expression expression expression)

Avoiding Evaluation

- Anything inside parenthesis are function calls (and therefore evaluated) unless quoted:
  - » `QUOTE` - takes one parameter; returns the parameter without evaluation, abbreviated ‘
  - » e.g., `'(A B)` is equivalent to (`QUOTE` (A B))
- ‘(a) returns a (it makes scheme think it is not something of value).
- (‘(a b c)) returns (a b c)

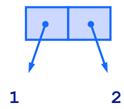
Dealing with Lists

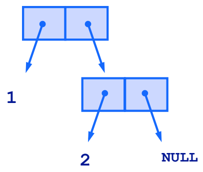
- LISP - Lots of Insipid Silly Parenthesis
- LISt Processing Language

- Lets talk about how to make lists…

CONS: CONStructs a pair

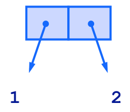
- (cons 1 2)
  - » (1 . 2)
- Creates a dotted pair, consisting of two atoms
- A list
  - » (1 . (2. nil)) -> (1 2)
- CONS builds a list from two parameters, the first is either an atom or a list, the second is usually a list.
  - » (cons ‘1 ‘()) -> 1





Splitting a Pair (car and cdr)

- (car (cons 1 2)) -> 1
- (cdr (cons 1 2)) -> 2



car extracts the first part of a pair

cdr extracts second part of a pair

Why “car” and “cdr”?

- Original (1950s) LISP on IBM 704
  - » stored cons pairs in memory registers
  - » car = “contents of the address part of register”
  - » cdr = “contents of the decrement part of the register (“could-er”)
- Think of them as the first and the rest (or head of list and tail of list)
  - » (define first car)
  - » (define rest cdr)

More examples

- `car` takes a list parameter; returns the first element of that list

  e.g., (car '(A B C)) yields A

  (car '((A B) C D)) yields (A B)
- `cdr` takes a list parameter; returns the list after removing its first element

  e.g., (cdr '(A B C)) yields (B C)

  (cdr '((A B) C D)) yields (C D)

  (cdr ‘A) is an error

Defining Threesomes

A triple is a pair where one of the pairs is a pair

```
(define (triple a b c )     (cons a ( cons b c )))
(define (triple-first  t)   (car t))
(define (triple-second t)   (car (cdr t )))
(define (triple-third t)    (cdr (cdr t)))
```

Lists

- List := (cons element list)
- A list is a pair where the second part is a list,
  - » ugh, how do we stop… this only allows infinitely long lists…
- A list is either
  - » a pair where the second pair is a list (cons Element List)
  - » or, empty (null)

Characteristics of “Pure” Functional Languages

- No side effects (e.g. no access to global variables)
- No assignment statements
- Often no variables
- Small concise framework
- Simple uniform syntax
- Recursive (that is how we get things done)
- Interpreted

Next Time

- **Tutorial on Scheme**