

Chapter 10

(7.8, 10.1 - 10.3)
Functional Programming

Side effects are fundamental to the Von Neumann Model of computing

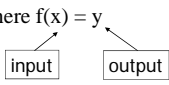
In (pure) functional, logic, and dataflow languages, there are no such changes

- These are called single-assignment languages

Function

A mapping of elements from one set (*domain*) to another set (*range*)

$f: X \rightarrow Y$ where $f(x) = y$



more precisely, $\forall x \in X, \exists! y \in Y: f(x) = y$

Example

$f: \mathbf{Z} \rightarrow \mathbf{Z}$ where $f(x) = x^2$ $g: \mathbf{Z} \rightarrow \mathbf{Z}$ where $g(x) = x + 5$

$\mathbf{Z} = \text{Integers}$

Composition of functions:

$f \circ g: \mathbf{Z} \rightarrow \mathbf{Z}$ where $f \circ g(x) = f(g(x)) = f(x+5) = (x+5)^2$

$g \circ f: \mathbf{Z} \rightarrow \mathbf{Z}$ where $g \circ f(x) = g(f(x)) = g(x^2) = x^2 + 5$

Functions

- Do not have side effects (functions do not modify their parameters)
- Parameters represent actual values from the domain (not memory locations)
- Do not have flow-of-control
- Can be a natural way to view programs

Functional Programming

- No assignment
- No state (no variables)
- Basic concepts of functional programming originated in LISP
 - Designed at MIT by John McCarthy (1960)
 - Started as a purely functional language

LISP (LIST Processing)

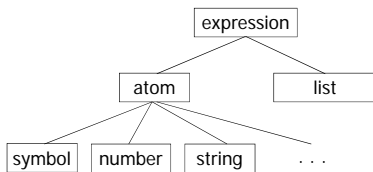
Simple (and unusual) syntax
 Data consists of *atoms* and *lists*
 Data and code have the same form
 Selection and recursion are the only control structures
 Usually interpreted (Read-Eval-Print loop)

Scheme

Based on LISP
 Developed at MIT around 1975 by Steele & Sussman
 Has added a number of imperative features

Scheme

Programs consist of *expressions*



Scheme Data

Atoms
 Usually numbers or symbols
 100 2.71828 x larger? end-of-list
 Two predefined symbols:
 #t #f

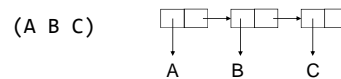
Scheme Data

Lists
 A sequence of expressions enclosed in parenthesis

(a b c)
 (
 (+ 4 9)
 (1 a (w x) 7 ((3) 5))

Scheme Lists

Internally, lists are implemented as singly linked lists of nodes



Scheme lists may be data or code

List as code

- Prefix form of a function
- First element is the function name
- Operands form the rest of the list
- All operands are evaluated before applying the function

Arithmetic Functions

```
(+ 4 9)
(* 3 8 2)
(* 4 (+ 5 6))
(+ (* 3 2) (- 7 1) (/ (+ 10 8) 6))
(expt 3 5)
(max 7 2 9 6)
(floor 5.678)
(odd? 25)
```

A Special Function

The first element of a list must evaluate to a function:

```
(1 2 3) ⇒ Error
```

quote - Returns its argument without evaluating it

```
(quote (1 2 3)) returns (1 2 3)
(quote x) returns x
```

quote is usually abbreviated by `'`

```
'(1 2 3) 'x
```

List Processing Functions

car - Takes one list as argument & returns the first item from its argument

cdr - Takes one list as argument & returns the argument list with the first item removed

Note: The names are acronyms for the IBM-704 registers used for these operations

List Processing Functions

```
(car '(a b c d))
(car '((a b) c d))
(cdr '(a b c d))
(cdr '((a b) c d))
(cdr '(5))
(car (cdr '(a b c d)))
(car (cdr '((a b) c d)))
(cdr (car '(a b c d)))
(cdr (car '((a b) c d)))
```

List Processing Functions

car and cdr can be abbreviated to at least 4 levels in Scheme:

```
(car (cdr '(1 2 3))) ⇒ (cadr '(1 2 3))
```

```
(car (cdr (cdr (car '((a b (c d) e) f g)))) ⇒ (caddar '((a b (c d) e) f g))
```