Functional Languages

- Programming language modeled on mathematical functions
  - Mapping from domain to range
  - Applied to an element of domain and produce an element of range
  - recursion primary definitional tool
  - No side effects
  - Independent of context
  - No concept of state
    - No "variables"

Functions

- Name, parameters, mapping expression
- At evaluation, parameters bound to arguments
- Lambda expressions
  - Nameless function
- Functional form
  - High-order function
  - Either takes function as parameter and/or evaluates to a function
  - E.g.
    - Composition
    - Apply-to-all
**Languages**

- Pure functional
  - No variables, assignment or concept of state
  - No iteration (no variables)
  - Uses recursion instead
  - Programs are function definitions and applications
  - Referentially transparent
  - Deterministic
  - Same result every time independent of context
  - Most not pure
- Components
  - Primitive functions
  - Functional forms
  - Structure for data
  - Equivalence of program and data

\[ \text{cube}(x) = x \times x \times x, \text{ where } x \text{ is a real number} \]
\[ \text{cube}(2.0) = 2.0 \times 2.0 \times 2.0 = 8 \]

\[ \lambda(x) \times x \times x \]
\[ \lambda(x) \times x \times \lambda(x) \]
LISP

- Prototypical functional language
  - Not pure
    - Includes iteration and variables for efficiency
- Data structure
  - Atom
  - List
    - Nodes are pairs (data, pointer)
    - E.g.
      - Simple list
      - Nested lists
    - Representation
- Dynamic scoping

![LISP example](image-url)
**Universal Function**

- Function to evaluate other functions
- Requires common data/function representation
  - S-expression
- Cambridge Polish notation
- Lambda notation for function definitions
- EVAL
  - Became the LISP interpreter
  - Operational semantics

```
(function_name argument1 ... argumentN)
```

```
(+ 5 7)
(+ 3 4 7.6)
```

```
(function_name \( \lambda \text{args}. \text{expression} \))
```

**Scheme**

- Dialect of LISP
- Static scoping
- Functions are first-class entities
- Interpreted (REPL)
- Primitive numeric functions
- Function definition
  - Lambda expression
    - Nameless function
    - Bound variable(s)
  - DEFINE
    - Binds name to value or name to lambda expression
- I/O
  - By nature implies state or causes side-effects
- Numeric predicates
  - #T and #F (also ()

- Control flow
  - Two-way selector (IF)
  - Multi-way selector (COND)
  - Recursion

- QUOTE (')
  - Prevent evaluation (i.e., list is not a function evaluation)

- CAR, CDR, CONS
  - CAAR, CDR, etc., are predefined functional compositions
  - LIST

- Symbolic predicates
  - EQ?, EQV?, LIST?, NULL?

- Example functions
  - member, equalsimp, equal, append

- Functional forms
  - compose, map

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>(+ 3 7)</td>
<td>21</td>
</tr>
<tr>
<td>(+ 5 7 8)</td>
<td>10</td>
</tr>
<tr>
<td>(- 5 4)</td>
<td>-1</td>
</tr>
<tr>
<td>(- 15 7 2)</td>
<td>6</td>
</tr>
<tr>
<td>(- 24 (+ 4 3))</td>
<td>12</td>
</tr>
</tbody>
</table>
### 15.6

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>EVEN?</td>
<td>Is n an even number?</td>
</tr>
<tr>
<td>ODD?</td>
<td>Is it odd?</td>
</tr>
</tbody>
</table>

```
(IF predicate then_expression else_expression)

(DEFP (factorial n)
  (IF (= n 1)
    1
    (* n (factorial (- n 1))))
)

(DEFP (leap? year)
  (COND
    (ZERO? (MODULO year 400)) #T)
    (ZERO? (MODULO year 100)) #T)
    (MODULO year 4)))))
)

(QUOTE A) returns A
(QUOTE (A B C)) returns (A B C)

(CAR '(A B C)) returns A
(CAR '(A B C D)) returns (A B)
(CAR '*A) is an error because A is not a list
(CAR '('A)) returns A
(CAR '()) is an error
(CDR '(A B C)) returns (B C)
(CDR '(A B C D)) returns (C D)
(CDR 'A) is an error
(CDR '(A)) returns ()
(CDR '()) is an error

(DEFP (second a_list) (CAR (CDR a_list)))

(second '(A B C))
```
\begin{verbatim}
(DEFINE (member atm a_list)
  (COND
    [(NULL? a_list) #F]
    [(EQ? atm (CAR a_list)) #T]
    [ELSE (member atm (CDR a_list))])
)

(member 'B '(A B C)) #T
(member 'B '(A C D E)) #F

(DEFINE (equalp list1 list2)
  (COND
    [(NULL? list1) (NULL? list2)]
    [(NULL? list2) #F]
    [(EQP (CAR list1) (CAR list2))
      (equalp (CDR list1) (CDR list2))]
    [ELSE #F])
)

(equalp '(A B C) '(A B C)) #T
(equalp '(A B C) '(A B D)) #F

(DEFINE (append list1 list2)
  (COND
    [(NULL? list1) list2]
    [ELSE (CONS (CAR list1) (append (CDR list1) list2))])
)

(append '(A B C) '(D E F)) '(A B C D E F)
(append '(A B C) '(D E F)) '(A B C D E F)
\end{verbatim}
\begin{verbatim}
(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
((compose CAR CAR) '((a b) c d))

(DEFINE (third a_list)
  ((compose CAR (compose CAR CAR)) a_list))

(DEFINE (map fun a_list)
  (COND
   (NULL? a_list) '(())
   (ELSE (CONS (fun (CAR a_list))
                (map fun (CDR a_list)))))
)

(map (LAMBDA (n) (* n n n n)) '(3 4 2 6))
This call returns (27 64 8 216).
\end{verbatim}