Mathematical Functions

- A mathematical function is a mapping of members of one set, called the domain set, to another set, called the range set.
- A lambda expression specifies the parameter(s) and the mapping of a function in the following form:
  \[ \lambda(x) \ x \ x \ x \ x \]
  for the function \( \text{cube} (x) = x \times x \times x \)

Lambda Expressions

- Lambda expressions describe nameless functions.
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression:
  \[ \lambda(x) \ x \ x \ x \ x (z) \]
  which evaluates to 8.
Functional Forms

- A higher-order function, or *functional form*, is one that
  - either takes functions as parameters
  - or yields a function as its result
  - or both

Function Composition

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second
- Form: \( h \equiv f \circ g \)
  - which means \( h(x) \equiv f \left( g(x) \right) \)
- For \( f(x) \equiv x + 2 \)
  and \( g(x) \equiv 3 \times x \),
  \( h \equiv f \circ g \) yields \( (3 \times x) + 2 \)

Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters
- Form: \( \alpha \)
  Example:
  For \( h(x) \equiv x \times x \)
  \( \alpha(h, (2, 3, 4)) \)
  yields \( (4, 9, 16) \)

Fundamentals of Functional PL’s

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible
- The basic process of computation is fundamentally different in a FPL than in an imperative language
  - In an imperative language, operations are done and the results are stored in variables for later use
  - Management of variables is a constant concern and source of complexity for imperative programming
- In an FPL, variables are not necessary, as is the case in mathematics
Fundamentals of Functional PL's (cont'd)

- Referential Transparency – In an FPL, the evaluation of a function always produces the same result given the same parameters
- Tail Recursion – Writing recursive functions that can be automatically converted to iteration

LISP Data Types and Structures

- Data object types: originally only atoms and lists
- List form: parenthesized collections of sublists and/or atoms
  - e.g., (A B (C D) E)
- Originally, LISP was a typeless language
- LISP lists are stored internally as single-linked lists

LISP Interpretation

- Lambda notation is used to specify functions and function definitions. Function applications and data have the same form.
  - Example:
    - If the list (A B C) is interpreted as data it is a simple list of three atoms, A, B, and C
    - If it is interpreted as a function application, it means that the function named A is applied to the two parameters, B and C
- The first LISP interpreter appeared only as a demonstration of the universality of the computational capabilities of the notation
  - EVAL was itself in the form of an expression

Origins of Scheme

- A mid-1970s dialect of LISP, designed to be a cleaner, more modern, and simpler version than the contemporary dialects of LISP
- Uses only static scoping
- Functions are first-class entities
  - They can be the values of expressions and elements of lists
  - They can be assigned to variables and passed as parameters
Evaluation

- The interpreter is a read-evaluate-write infinite loop.
  - Parameters are evaluated, in no particular order
  - The values of the parameters are substituted into the function body
  - The function body is evaluated
  - The value of the last expression in the body is the value of the function

Primitive Functions

- Arithmetic: +, -, *, /, ABS, SQRT, REMAINDER, MIN, MAX
  - e.g., (+ 5 2) yields 7
- QUOTE - takes one parameter; returns the parameter without evaluation
  - QUOTE is required because the Scheme interpreter, named EVAL, always evaluates parameters to function applications before applying the function. QUOTE is used to avoid parameter evaluation when it is not appropriate
  - QUOTE can be abbreviated with the apostrophe prefix operator:
    ' (A B) is equivalent to (QUOTE (A B))

Function Definition: LAMBDA

- Lambda Expressions
  - Form is based on \( \lambda \) notation
    - e.g., (LAMBDA (x) (* x x))
      x is called a bound variable
- Lambda expressions can be applied
  - e.g., ((LAMBDA (x) (* x x)) 7)

Special Form Function: DEFINE

- A Function for Constructing Functions
  - Two forms:
    1. To bind a symbol to an expression
       e.g., (DEFINE pi 3.141593)
       Example use: (DEFINE two_pi (* 2 pi))
    2. To bind names to lambda expressions
       e.g., (DEFINE (square x) (* x x))
       Example use: (square 5)
- The evaluation process for DEFINE is different! The first parameter is never evaluated. The second parameter is evaluated and bound to the first parameter.
Output & Predicate Functions

• Output Functions
  - (DISPLAY expression)
  - (NEWLINE)

• Numeric Predicate Functions
  - #T is true and #F is false (sometimes () is used for false)
  - =, <>, >, <, >=, <=
  - EVEN?, ODD?, ZERO?, NEGATIVE?

Control Flow: \textbf{IF}

• Selection- the special form, IF
  \[(\text{IF predicate then}\_\text{exp else}\_\text{exp})\]
  - e.g.,
    \[(\text{IF } (<> \text{count } 0) \text{ (/ sum count) } 0)\]

Control Flow: \textbf{COND}

• Multiple Selection - the special form, COND
  General form:
  \[(\text{COND} \hspace{1cm} (\text{predicate}_1 \hspace{0.5cm} \text{expr} \hspace{0.5cm} \{\text{expr}\}) \hspace{1cm} \ldots)\]
  \[(\text{ELSE} \hspace{0.5cm} \text{expr} \hspace{0.5cm} \{\text{expr}\})\]
  - Returns the value of the last expression in the first pair whose predicate evaluates to true

Example of \textbf{COND}

\[(\text{DEFINE} \hspace{0.5cm} (\text{compare} \hspace{0.5cm} x \hspace{0.5cm} y) \hspace{0.5cm} \{
\begin{align*}
(\text{COND} \hspace{0.5cm} ((> x y) \hspace{0.5cm} \text{"x is greater than y"}) \hspace{0.5cm} ((< x y) \hspace{0.5cm} \text{"y is greater than x"}) \hspace{0.5cm} \text{(ELSE "x and y are equal")}) \\
\end{align*}
)\}
)\]
List Functions: **CONS and LIST**

- **CONS** takes two parameters, the first of which can be either an atom or a list and the second of which is a list; returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result. e.g., (CONS 'A '(B C)) returns (A B C)
- **LIST** takes any number of parameters; returns a list with the parameters as elements

List Functions: **CAR and CDR**

- **CAR** takes a list parameter; returns the first element of that list.
  - **Examples:**
    - (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.
  - **Examples:**
    - (CDR '(A B C)) yields (B C)
    - (CDR '((A B) C D)) yields (C D)

Predicate Function: **EQ?**

- **EQ?** takes two symbolic parameters; it returns #T if both parameters are atoms and the two are the same; otherwise #F
  - **Examples:**
    - (EQ? 'A 'A) yields #T
    - (EQ? 'A 'B) yields #F
- Note that if EQ? is called with list parameters, the result is not reliable.
- Also EQ? does not work for numeric atoms

Predicate Functions: **LIST? and NULL?**

- **LIST?** takes one parameter; it returns #T if the parameter is a list; otherwise #F
- **NULL?** takes one parameter; it returns #T if the parameter is the empty list; otherwise #F
  - Note that NULL? returns #T if the parameter is ()