Jumping right in

A Scheme procedure

(define increment
  (lambda (n)
    (+ n 1))
)
A call to the procedure

(increment 21)

The Spirit of Lisp-like Languages

We shall first define a day of symbolic expressions in terms of ordered pairs and lists. Then we shall define fundamental functions and procedures built from them by composition, conditional expressions and recursive definitions in an extensive data of functions of which we shall give a number of examples. We shall then show how these functions can themselves be expressed as symbolic expressions and we shall give a simple function interpreter that allows us to compile from the expressions for a given function its value for given arguments. Finally, we shall define some functions with functions as arguments and give some useful examples.


Pure Functional Languages

Fundamental concept: application of mathematical functions to values

1. Referential transparency: The value of a function application is independent of the context in which it occurs
   - value of (f a b c) depends only on the values of f, a, b and c
   - It does not depend on the global state of computation
   - all vars in function must be parameters

2. The concept of assignment is not part of functional programming
   - no explicit assignment statements
   - variables bound to values only through the association of actual parameters to formal parameters in function calls
   - function calls have no side effects
   - thus no need to consider global state

3. Control flow is governed by function calls and conditional expressions
   ⇒ no iteration
   ⇒ recursion is widely used

4. All storage management is implicit
   ⇒ needs garbage collection

5. Functions are First Class Values
   ⇒ can be returned as the value of an expression
   ⇒ can be passed as an argument
   ⇒ can be put in a data structure as a value
   ⇒ Unnamed functions exist as values

A Functional Program

A program includes:

1. A set of function definitions
2. An expression to be evaluated

Eg, in Scheme:

1] (define (increment n)
   (if (> n 0)
     (+ n 1))

[Value: increment

1] (increment 21)

[Value: 23]

1] (define (test)
   (lambda (x)
     (+ x x))

[Value: test]

1] (test 1)

[Value: 2]

1] (define (uniform (a b))
   (lambda (x)
     (if (> x a)
       b
       0))

[Value: uniform]

1] (uniform 1 2)

[Value: 1]
Formal Root: λ-Calculus

- Defined by Alonzo Church, a logician, in 1930s as a computational theory of recursive functions.
- λ-Calculus is equivalent in computational power to Turing machines.
- Recall: what's a Turing machine? Turing machines are abstract machines that emulate computation as a series of state transitions driven by symbols on an input tape (which leads naturally to an imperative style of programming based on assignments).
- How is λ-calculus different?
  - λ-Calculus emphasizes typed expressions and functions (which naturally leads to a functional style of programming).
  - No state transitions.

λ-Calculus (cont.)

λ-Calculus is a formal system for defining recursive functions and their properties.

- Expressions are called λ-expressions.
- Every λ-expression denotes a function.
- A λ-expression consists of 3 kinds of terms:
  - Variables: z, x, etc
  - ‘‘ denotes arbitrary variables.
  - Abstractions: λx.E where x is some variable and E is another λ-term.
  - Applications: (E1 E2) where E1 and E2 are λ-terms. Applications are sometimes called combinations.

λ-Calculus (cont.)

Formal Syntax in BNF

<λ-expr> ::= <variable> | λ<variable><λ-expr> | (√<λ-expr> <λ-expr>)
<variable> ::= x | y | z | ..."

Or more compactly:

E := λx.E1 E2
V := x | y | z | ...

Where y is an arbitrary variable and x is an arbitrary λ-expression call x the head of the λ-expression and z the body.

λ-Calculus: Functional Forms

A higher-order function (functional form):
- Takes functions as parameters
- Yields a function as a result.
E.g.: Given
f(x) = x + 2, g(x) = x^2 + x then
h(x) = f(g(x)) and
h(x) = (3 * x) + 2
h() is called a higher-order function.

Types of Functional Forms:

Construction form: E.g.,
g(x) = x + x, h(x) = 2 * x, s(x) = x / 2
[g,a,b] (4) = (1,6,2)

Application form: E.g.
h(x) = x + x
y(x) = 3 * x
y(b, (2,3,4)) = (4,9,16)

λ-Calculus: Is it really Turing Complete?

Can we represent the class of Turing computable functions?

Yes, we can represent:
- Boolean and conditional functions
- Numerical and arithmetical functions
- Data structures: ordered pairs, lists, etc.
- Recursion, but, doing so in λ-calculus is tedious.
- Need syntactic sugar to simplify tasks.
- λ-calculus most suitable as an abstract model of a programming language rather than a practical programming language.

Both Turing machines and λ-calculus are idealized, mathematical models of computation.

Scheme: A Functional Programming Language

1958: Lisp
1975: Scheme (based on the same)
1980: Common Lisp ("CL")

Lisp, Scheme, and CL are described on following pages.

Some features of Scheme:
- Generational semantics based on the λ-calculus, t.p. the need for garbage collection is minimized at the time that the garbage (i.e., evaluated and returned).
- arbitrary data structures w/ continuations,
- functions as first-class values
- automatic garbage collection

LISP

- Functional language developed by John McCarthy in 1968.
- Semantics based on λ-calculus
- All functions operate on lists or atomic symbols (called "S-expressions")
- Only five basic functions: list functions cons, car, cdr, equal, atom and one conditional construct: cond
- Uses dynamic scoping
- Useful for list-processing applications
- Programs and data have the same syntactic form: S-expressions
- Used in Artificial Intelligence
**SCHEME**

- Developed in 1975 by G. Sussman and G. Steele
- A version of LISP
- Consistent syntax, small language
- Closer to initial semantics of LISP
- Provides basic list processing tools
- Allows functions to be first-class objects
- Provides support for lazy evaluation
- Lexical scoping of variables

**COMMON LISP (CL)**

- Implementations of LISP did not completely adhere to semantics
- Semantics redefined to match implementations
- COMMON LISP has become the standard
- Committee-defined language (1980s) to unify LISP variants
- Many defined functions
- Simple syntax, large language

**Commonalities between LISP and SCHEME**

- Expressions are written in prefix, parenthesized form:
  - `(function arg1 arg2 ... argn)`
  - `(+ 4 5)
  - `(+ (* 3 4 5) (+ 3 4))`

- In order to evaluate an expression:
  1. Evaluate function to a function value
  2. Evaluate each arg, in order to obtain its value
  3. Apply the function value to these values

**S.expressions**

Common structure for both procedures and data. In SCHEME, functions are called procedures.

When an expression is evaluated, it creates a value or list of values that can be embedded into other expressions. Therefore programs can be written to manipulate other programs.

```
<expression> ::= <variable>
| <expression> <operator> <expression>
| <cond-expression>
| <let-expression>
| <letrec-expression>
| <defun>
| <macro-definition>
| <type>
| (())
| (a b c) d
| (a b c) (d e (f g))
| (1 2)
```

Lists have nested structure.

---

**Built In Procedures**

- `eq`: identity of atoms
- `null`: is list empty?
- `car`: selects first element of list
- `cdr`: selects rest of list
- `(cons element list)`: constructs lists by adding element to front of list
- `quote` of `'`: produces constants

**Built In Procedures**

- `'( )` is the empty list
- `(car '(a b c))`
- `(cdr '(a b c))`
- `(cadr '(a b c d))`
Things you should know about cons, pairs and lists

<table>
<thead>
<tr>
<th>Other (Predicate) Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicates procedures return #t or () (true, false),</td>
</tr>
<tr>
<td>* = + / numeric operators, e.g.,</td>
</tr>
<tr>
<td>(+ 6 3) = 9, (+ 6 5) = 11, (+ 6 3) = 1.6666666</td>
</tr>
<tr>
<td>* = &lt; &gt; = number comparison ops</td>
</tr>
<tr>
<td>Run-time type checking procedures</td>
</tr>
<tr>
<td>- All return Boolean values #t and ()</td>
</tr>
<tr>
<td>- (number? 6) is #t</td>
</tr>
<tr>
<td>- (zero? 0) is #t</td>
</tr>
<tr>
<td>- (symbol? 'a) is #t</td>
</tr>
<tr>
<td>- (list? '(a b)) is #t</td>
</tr>
<tr>
<td>- (null? '()) is #t</td>
</tr>
</tbody>
</table>

The pair or cons is the most fundamental of Scheme's structured object types.
A list is a sequence of pairs; each pair consists of the next pair in the sequence.
The car of the last pair in a proper list is the empty list, otherwise the sequence of pairs from the root to the last pair is the list, and any pair whose cdr is a proper list is a proper list.
An improper list is printed in dotted-pair notation with a prefix list preceding the first element of the list. A pair whose cdr is not a list is often called a dotted pair e.g., '(a b). The procedure cons actually builds pairs, and there is no reason that the cdr of a pair must be a list, as illustrated on the previous page.

The procedure list is similar to cons, except that it takes an arbitrary number of arguments and always builds a proper list.

Example: (list 'a 'b 'c) → (a b c)

READ-EVAL-PRINT Loop

Example:

```
1. Evaluate cons to obtain a procedure
2. Evaluate 'a to obtain a itself
3. Evaluate (cons 'b 'c) to obtain (a b c)
4. Evaluate (cons 'b 'c) to obtain (a b c)
5. Apply the cons procedure to a and (b c d) to obtain (a b c d)
6. Print the result of the application: (a b c d)
```

Quotes Inhibit Evaluation

```
;; Same as before:
1. (cons 'a (cons 'b 'c d))
;Value 1: (a b c d)
2. (cons 'a (cons 'b 'c d))
;Value 2: (a b c d)
```

Quotes vs. Eval

```
;; Same things evaluate to themselves:
1. (list 'a (quote 'b ) (c d))
;Value 4: (1 2 #t ()
2. (quote 'b ) (c d))
;Value 5: (1 2 #t ()
```

Eval Activates Evaluation

```
;; They can also be quoted:
1. (list 'a (quote 'b ) (c d))
;Value 6: (1 2 #t ()
2. (quote 'b ) (c d))
;Value 7: (1 2 #t ()
```

Can also be used to define procedures.

READ: Read input from user

```
READ: Read input from user:
```

```
Example:
```
```
Procedure Definition

Two syntaxes for definition:
1. (define (if (condition) <result1> <result2>)
   (define (square x) (+ x x))
2. (define <name> (if <condition> <true-case> <false-case>)
   (define mean (lambda (x y) (/ (+ x y) 2)))

Lambda procedure syntax enables the creation of anonymous procedures. More on this later!

Conditional Execution: if

(if <condition> <result1> <result2>)
1. Evaluate <condition>
2. If the result is a "true value" (i.e., anything but 0 or nil), then evaluate and return <result1>
3. Otherwise, evaluate and return <result2>

(define (abnormal x)
  (if (= x 0) x (+ x x))
)

(define (repeat-first-of-list)
  (if (eq? (car list) (cdr list)) '())
)

Conditional Execution: cond

(cond (if <condition> <result1>)
      (if <condition> <result2>)
      ...
      (if <condition> <resultn>)
      (else <else-result>)) ; optional else

1. Evaluate conditions in order until obtaining one that returns a true value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return <else-result>

(define (abnormal x)
  (if (= x 0) x (+ x x))
)

(define (repeat-first-of-list)
  (if (eq? (car list) (cdr list)) (cdr list))
)

Conditional Execution: cond

1. Evaluate conditions in order until obtaining one that returns a true value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return <else-result>

Better atom? procedure

Any list is a pair (dotted pair with CAR and CDR), except the empty list (which is both list and atom),

(define (atom? x)
  (if (symbol? x) 't
      (if (number? x) 't
          (if (char? x) 't
              (if (string? x) 't
                  (if (null? x) 't
                      (else 'f)
                  )
              )
          )
      )
  )
)

Conditional vs. Boolean Expressions

Write a procedure that takes a parameter x and returns #t if x is an atom, and false otherwise, using cond:

(define (atom? x)
  (cond ((symbol? x) 't)
        ((number? x) 't)
        ((char? x) 't)
        ((string? x) 't)
        ((null? x) 't)
        (else 'f))
)

Conditional vs. Boolean Expressions

Now write atom? without using cond:

(define (atom? x)
  (if (symbol? x) 't
      (if (number? x) 't
          (if (char? x) 't
              (if (string? x) 't
                  (if (null? x) 't
                      (else 'f)
                  )
              )
          )
      )
  )
)

Five Steps to a Recursive Function

1. Strategy: How to reduce the problem?
2. Header: What is the function header? Use a name phrase for the function name,
3. Spec: Write a method specification in terms of the parameters and return, reduce expressions;
4. Base Case:
   - What is the answer so simple that we know it without recursing?
   - What is the answer in these base case(s)?
5. Recursive Case:
   - Decompose the input: the other case(s) is in terms of the answer on smaller inputs,
   - Simplify if possible
   - Write code for the recursive case(s),
Recursive Scheme Procedures:

Sum-N

Parameter: integer \( n \geq 0 \),

Result: sum of integers from 0 to \( n \)

\( \text{return} \) \( \text{(sum-n n)} \)

\( \text{(define (sum-n n)} \)

\( \text{(cond} \) \( \)

\( \text{\( (0 \to (\text{+ n 0)} \)} \)

\( \text{\( (\text{\text{else}} \to \text{(sum-n -1 n))} \)} \)

\( \text{\( )} \)

This is called ‘tail-recursion’

Note: There is a built-in \text{length} procedure,

Recursive Scheme Procedures:

Length

\( \text{(define (length x)} \)

; In to range

\( \text{(length 'a b c d)} \)

\( \text{[entering # trophies procedure 5 length]} \)

\( \text{arg1: (a b c)} \)

\( \text{[entering # trophies procedure 5 length]} \)

\( \text{arg2: (b c)} \)

\( \text{[entering # trophies procedure 5 length]} \)

\( \text{arg3: (c)} \)

\( \text{[entering # trophies procedure 5 length]} \)

\( \text{arg4: (d)} \)

\( \text{0 < < # trophies procedure 5 length} \)

\( \text{arg5: (5)} \)

\( \text{1 < < # trophies procedure 5 length} \)

\( \text{arg6: (1)} \)

\( \text{2 < < # trophies procedure 5 length} \)

\( \text{arg7: (2)} \)

\( \text{3 < < # trophies procedure 5 length} \)

\( \text{arg8: (3)} \)

; Value: 3

Length (cont.)

Recursive Scheme Procedures:

Abs List

\( \text{* (abs-list 'f 2 24 40)} \)

\( \text{=> (1 2 3 4 0)} \)

\( \text{* (abs-list 'f)} \)

\( \text{=> ()} \)

\( \text{(define (abs-list list)} \)

\( \text{)} \)

Recursive Scheme Procedures:

Append

\( \text{(append '(1 2) '(3 4 5)) = (1 2 3 4 5)} \)

\( \text{(append '(1 2) '(3 4 5) \) = (1 2 3 (4) 5)} \)

\( \text{(append '(1 2) '(3 4)) = (1 2 3 4 5)} \)

\( \text{(append '(1 2) '(3 4) \) = (1 2 3 4 5)} \)

\( \text{(append '(1) \) = (1)} \)

\( \text{(define (append x y)} \)

\( \text{)} \)

Note: There is a built-in \text{append} procedure,