Functional Programming — Scheme

Steven Zeil

Nov. 10, 2003

Contents

1 Scheme
   1.1 Data Types ........................................ 1
      1.1.1 atoms .................................... 1
      1.1.2 lists .................................... 1
   1.2 Expressions ....................................... 2
      1.2.1 General Form ................................ 2
      1.2.2 Quoting .................................... 2
      1.2.3 List Ops .................................... 3
      1.2.4 Conditionals ................................ 3
   1.3 Functions ......................................... 3
   1.4 Lexical Scope ..................................... 3
   1.5 Programming in Scheme ........................... 3

2 Implementing LISP
   2.1 Implementing Lists ................................ 4
      2.1.1 List Cells ................................ 5
   2.2 Garbage Collection ................................. 5

3 Functional Programming Influences on C++ .......... 5

Functional Programming

1. Overview
2. SML
3. Scheme
4. Implementing LISP
5. Functional Programming Influences on C++

1 Scheme

Scheme is a dialect of LISP (LISt Processing)

1. Data Types
2. Expressions

1.1 Data Types

Data in Scheme is divided into

1. atoms
2. lists

1.1.1 atoms

Atoms can be

• integers: 3, 0
• real numbers: 3.14
• symbols: 'x, 'abc

– unquoted, these are variables

Special symbols:
• #t, #f are booleans

1.1.2 lists

General form of compound data is the list:

\[ \langle \text{list} \rangle ::= (\{\langle \text{item} \rangle \}) \]

\[ \langle \text{item} \rangle ::= \langle \text{atom} \rangle | \langle \text{list} \rangle \]

Examples:

() (1) ('a 'b 'c) ('a ('a 'b 'c) 'c) (((('a))) 'b 'c)
1.2 Expressions

1.2.1 General Form

General form of code is the parenthesized prefix expression:

\[
\langle \text{expr} \rangle ::= \langle \text{operator} \rangle \{ \langle \text{item} \rangle \}
\]

\[
\langle \text{operator} \rangle ::= \langle \text{item} \rangle
\]

Examples:

\[(+ 5 3)\]
\[(* 2 (+ 5 3))\]

1.2.2 Quoting

Suppose we want the data “5 + 3” instead of the value 8.
To indicate that we want the + treated as an atom rather than as an operator, quote the list:

\[\text{quote } (+ 5 3)\]
\[\text{quote } (+ 5 3)\]

1.2.3 List Ops

Lists are manipulated with three basic operators:

- **cons**
- **car**
- **cdr**

**cons**

prepends an item onto a list

- \((\text{cons} 1 (2 3))\) produces \((1 2 3)\)
- \((\text{cons} (1) (2 3))\) produces \(((1) 2 3)\)

**car**

extracts the first element from a list

- \((\text{car} (1 2 3))\) is 1
- \((\text{car} ((1 2) (3 4)))\) is \((1 2)\)

**cdr**

returns the list of all except the first element

- \((\text{cdr} (1 2 3))\) is \((2 3)\)
- \((\text{cdr} ((1 2) (3 4)))\) is \(((3 4))\)

**null?**

tests a list to see if it is empty.

- \((\text{null? } ())\) is \#t
- \((\text{null? } (1 2))\) is \#f
- \((\text{null? } ())\) is \#f
1.2.4 Conditionals

(if P E₁ E₂) is the Scheme equivalent to SML's if-then-else

- (if (null? L) () (cdr L))

A more general form of conditional is

(cond (P₁ E₁₁) (P₂ E₂₁) ... (else Eₙ₁))

- The predicates Pᵢ are evaluated, one after another, until one is not #f.
- Then the corresponding Eᵢ is returned.

Test Operators

- (null? X) Is X empty?
- (pair? X) Is X a list (a cons pair)?
- (atom? X) Is X an atom?
- (number? X) Is X a number?
- (symbol? X) Is X a symbol?
- (equal? X Y) Are X and Y equal? (deep)
- (eq? X Y) Are X and Y equal? (shallow)
- (< X Y) Is number X < number Y?

1.3 Functions

Functions are declared via define:

(define (name ⟨formals⟩) ⟨expr⟩)

(define abs x)
  (if (> x 0) x (‐ x))

An alternate, and perhaps more interesting form, is:

(define name ⟨function-value⟩)
(define pi 3.14159)

Functions are 1st Class Objects

More generally, function values are written as lambda expressions:

- (lambda ⟨(formals)⟩ ⟨expr⟩)

(define abs)
  (lambda (x) (if (> x 0) x (‐ x)))

1.4 Lexical Scope

Like SML, we can bind names to constant values in a limited scope:

(let ((x₁ E₁) (x₂ E₂) ...) E)

(define a 2)
(define b 3)
(let ((a 4) (d 2)) (+ a b d))
(let ((a 4) (c (+ a b))) c)

What are the values of the let expressions?

1.5 Programming in Scheme

Start with a simple list manipulator:

append should join two lists.

(append (1 2) (3 (4))) should return
(1 2 3 4)

- Note that cons joins an item and a list:

  - (cons (1 2) (3 (4))) returns
    ((1 2) 3 (4))

(define (append x y)
  (cond ((null? x) y)
        (else (cons
               (car x)
               (append (cdr x) y)
               ))
             ))

Function

As in SML, much of the power of the language comes from the use of higher-order functions.

- (map f L) applies f to each element of L, collecting the result into a list.

  (map abs ’(2 4 7)) returns (2 4 7)
Implementing map
Could be defined as

\[
\text{define } (\text{map } f \text{ L} ) =
\begin{cases} 
(\text{null? L}) 
& \text{'}() \\
\text{else} (f \text{ (car L)}) 
& \text{cons } (\text{map } f \text{ (cadr L)})
\end{cases}
\]

- but map is actually predefined in Scheme
- Predefined map can apply to functions of different arity

\[
(\text{map } + \text{ '(2 -4 -7) (1 2 3)}) \text{ returns } (3 -2 -4)
\]

Another interesting h.o.f. is reduce

- \((\text{reduce } f \times \text{ (v1 v2 ... vn)})\) computes \((f \times (f \times (f \times \text{ (v1 v2 ... vn)})))\)
- For example, we can define a summation function \(\Sigma_i\) as

\[
(\text{define } (\text{sumAll } x) = (\text{reduce } + \text{ 0 x}))
\]

reduce is implemented as

\[
(\text{define } (\text{reduceX } f \text{ v}) =
\begin{cases} 
(\text{null? v}) \text{ (cadr v)} 
& \text{cond } (f \text{ (cadr v)}) \\
\text{else} (f \text{ (cadr v)}) 
& (\text{reduce } f \text{ (cadr v)})
\end{cases}
\]

\[
(\text{define } (\text{reduce } f \times x \text{ v}) =
\begin{cases} 
(\text{null? v}) \text{ x} 
& \text{cond } (f \times (\text{reduceX } f \text{ v}))
\end{cases}
\]

A vector dot product is defined as

\[
\vec{x} \cdot \vec{y} = \sum_i x_i \times y_i
\]

Can you use sumAll, reduce, and/or map to produce a dot produce function?

**Association Lists**

A common idiom in Scheme is the **association list**, or **a-list**

- a list of pairs, which map keys to values
- first element of each pair is usually a symbol

\[
(\text{define people} =
\begin{cases} 
(\text{edv } (((\text{name }\text{ "Ed"}) \text{ (id 123)))) \\
(\text{suev } (((\text{name }\text{ "Sue"}) \text{ (id 278)))) \\
(\text{billy } (((\text{name }\text{ "Bill"}) \text{ (id 380)))) \\
\text{))}
\]

\[
(\text{define project1 } '((\text{manager ed}) \text{ (staff sue bill))})
\]

\[
(\text{assoc } x \text{ A}) \text{ extracts the (first) pair keyed by x in the a-list A.}
\]

\[
(\text{define (manager project)} = (\text{cadr (assoc } \text{ 'manager project}))
\]

\[
(\text{define (managerName project)} = (\text{cadr (assoc } \text{ 'name (cadr (assoc (manager project) people))})
\]

2 Implementing LISP

LISP was originally envisioned as a LLL to implement a a List processing HLL.

It offers some interesting insights into implementation of FP.

1. **Implementing Lists**
2. **Garbage Collection**

2.1 Implementing Lists

In LISP/Scheme, typically two separate memory pools

- storage for atoms
Functional Programming — Scheme

- contains no pointers to other objects
- may be subdivided by kind/size of atom

- storage for lists

2.1.1 List Cells

A list is represented as a collection of cells:

• (car L) retrieves the pointer from the first part of the cell.
• (cdr L) retrieves the pointer from the second part of the cell.
• (cons H L) allocates a new cell, placing H and L in the two parts of the cell.

\[
\begin{align*}
\text{let} & \quad ((\text{edv } '((\text{name } \text{"Ed"}) (\text{id } 123))) \\
& \quad (\text{suev } '((\text{name } \text{"Sue"}) (\text{id } 278))) \\
& \quad (\text{billyv } '((\text{name } \text{"Bill"}) (\text{id } 380))) \\
& \quad '(((\text{ed , edv}) (\text{sue , suev}) (\text{bill , billyv})))
\end{align*}
\]

Association lists are used heavily in the implementation of LISP/Scheme.

- A special a-list, called the environment, contains the current list of bound variable names, associated with their values.
- Whenever the interpreter encounters a variable name, it evaluates it as (assoc name Environment).

2.2 Garbage Collection

- FPL’s use rely on shared data structures to make constructive manipulation efficient.
- Their implementations therefore make heavy use of pointers.
- Automatic storage management (garbage collection) is essential.

Some non-FP languages (e.g., Java, Modula 3) use automatic garbage collection as well.

- Often these languages feature reference semantics.
- Such languages usually do not have a delete command, so both garbage and dangling pointers are eliminated.

3 Functional Programming Influences on C++

The influence of the functional style can be seen in the new standard C++ library, which is filled with higher-order functions:

```cpp
list<Student> cs355Roster;
...
void printName(const Student& s) {
    cout << s.name() << endl;
}
...
void printRoster() {
    list<Student>::iterator start = cs355Roster.begin();
    list<Student>::iterator stop = cs355Roster.end();
    for_each(start, stop, printName);
}
```
list<Student> univ;
:
Student updateGPA(Student s) {
    Grades g = thisSemester.grades(s);
    s.gpa = computeGPA(s.gpa, s.hours, g);
    return s;
}

void reportCards() {
    list<Student>::iterator start = univ.begin();
    list<Student>::iterator stop = univ.end();
    transform (start, stop,
             start, updateGPA);
}

bool honors(const Student& s) {
    return s.gpa() > 3.5;
}

bool honors(const Student& s) {
    return s.gpa() > 3.5;
}
:

void selectHonors() {
    list<Student>::iterator start = univ.begin();
    list<Student>::iterator stop = univ.end();
    list<Student>::iterator toBeRemoved;
    toBeRemoved = remove_if (start, stop, honors);
    univ.erase (toBeRemoved, stop);
}

Note how not1 is used to generate a new function from an old one.

Objects can also simulate functions, and have the advantage of being fully 1st-class.

struct GPASelector
public unary_function<Student, bool>
{
    typedef double argument_type;
    double limit;
    GPASelector (double lim)
    {limit = lim;}

    bool operator() (const Student& s) {
        return s.gpa() > limit;
    };
}

void selectHonors(double gpa) {
    list<Student>::iterator start = univ.begin();
    list<Student>::iterator stop = univ.end();
    list<Student>::iterator toBeRemoved;
    toBeRemoved = remove_if (start, stop, gpa);
    univ.erase (toBeRemoved, stop);
}

not1 is a true h.o.f. It takes a function as a parameter and produces a new function (actually a simulating object).

template <class Predicate>
class unary_negate
    : public unary_function<Predicate::argument_name, bool>
{
    Predicate pred;
public:
    unary_negate (Predicate p): pred(p)
    {};
    bool operator() (typename Predicate::argument_name x)
    { return !pred(x); };
};

template <class Predicate>
UnaryFunction not1 (Predicate p) {
    return unary_negate<Predicate>(p);
};