The Scheme Programming Language

- Scheme is a functional language
- Scheme is a dialect of LISP.
- It was designed in the mid 1970s to be a teaching language.
- Scheme is a small language with a very simple syntax.

Functional Programming Languages

- based on the theory of mathematical functions
- value-based rather than state based
- applicative – result obtained by applying a sequence of functions to the input
- recursion replaces iteration
- most functional languages are interactive

Functions

A mathematical function maps members of a domain (the possible inputs) to a range (the possible outputs). The mapping must be unique.

A function definition consists of

- signature – specifies domain (inputs) and range (outputs)
- mapping rule – specifies how to determine the value of the range corresponding to each value in the domain

Once a function has been defined, it can be *applied* to any element in the domain.

Components of a functional language

- set of data objects – e.g. numbers of various types, aggregate data types
- set of built-in functions for manipulating data objects – e.g. operators for numeric data
- set of functional forms for creating new functions – composition

Some functional languages

- LISP – has a number of dialects
  - first functional language; initially purely functional
  - there are several variants (e.g. common LISP, Scheme)
  - dynamic types and scope
  - list data structure
  - introduced garbage collection
- ML
  - functional base with additions
  - list data structure
  - strongly typed, static scope
  - type inferencing
- Haskell – a purely functional language

Scheme Programs

- A Scheme program is a sequence of definitions and expressions executed in order.
- An expression can be either an atom or a list.
  - A list is a sequence of expressions enclosed by parentheses.
- Anything following a semicolon, **;** (up to the end of the line) is considered a comment

Atoms – literals

- numbers – just type them in
- characters – `#\a` or `#\space`
- strings of characters
- boolean values – `#t` and `#f`
- symbols (like variables) – a name (can't start with a digit) which is associated with a value of some kind

Any atom has an inherent value.

numbers in Scheme

- Allowed Values: Scheme supports the usual numeric types like integer and real as well as rational and complex
- Operations:

`+ - * / mod`	usual arithmetic operations
number?	predicate to test for number
zero?	predicate to test for zero
`< > <= >=`	comparison operators
`=`	equality testing for numbers
`abs sqrt sin ...`	standard mathematical functions

- Literal Representation:

  `12 -2 25.4 6.7e-4`

Note: Both `+` and `*` are variatic in Scheme, that is, they can take any number of arguments.

```
(+ 1 2 3 4) ; adds up the numbers 1 through 4
```

```
(* 1 2 3 4 5) ; is 5!
```

Lists

- A list is a sequence of expressions enclosed by parentheses and separated by spaces.

  A literal list has a single quote character (') in front of it.

  The interpreter will attempt to evaluate any list that is not literal.

  Scheme expressions are in prefix format – the first element of the list is assumed to be *applicable*, *i.e.* it should be a function. All remaining elements are assumed to be arguments.

  ```
  (+ 1 5)
  (* (+ a b) (- a b))
  (sqrt 25)
  ```

- All arguments are evaluated and the function is applied to the results.

Special Forms in Scheme

There are some kinds of operations that cannot be done using the standard evaluation process. For these, there needs to be a special syntax, called a *special form*. The most important of these are listed below.

- and or – short-cut evaluation doesn't work if these are regular functions
- define – give a name to a value or function
- if, cond – conditional evaluation
- lambda – create a function
- let, letrec – local binding of names to values

and & or

The logical operators **and** and **or** are special forms to allow for short-cut evaluation

both can take any number of parameters (variatic like + and *)

```
(and test1 test2 ...testn)

(or test1 test2 ... testn)
```

define

```
(define variable expression)
```

Sequencing:

1. evaluate expression
2. bind value of expression to variable

if

```
(if test-expr then-expr else-expr)
```

Sequencing:

1. evaluate test-expr
2. if true, evaluate then-expr
3. otherwise, evaluate else-expr

cond

```
(cond
    (test1 conequent1)
    (test2 conequent2)
    ...
    (else alternate))
```

Omitting the else results in an unspecified return value for an expression that executes that case.

This is equivalent to

```
(if test1 consequent1
    (if test2 consequent2
        ...
            alternative)...))
```

Scheme References

The Scheme Programming Language by R. Kent Dybvig – on reserve at the library and on-line at

`http://www.scheme.com/tspl2d/`

The Structure and Interpretation of Computer Programs by Harold Abelson, Gerald Jay Sussman with Julie Sussman uses Scheme to illustrate the principles of computer programming (on reserve)

The Little Schemer by Daniel P. Friedman and Matthias Felleisen (library has)

Running scheme on onyx

- MIT Scheme is installed on onyx. To run it, simply type

  `scheme`

- The Scheme interpreter is interactive. You type in an expression and it prints out the value of the expression. (Or an error message if you did something wrong. :-)

- For long sequences of expressions or complicated expressions, you will probably want to type the code into a text file. Then, you can load the file by typing

  `(load "defs.scm")`

  where defs.scm is the name of the file to be loaded.

- You can also start up the interpreter and read stuff from a file by the command

  `scheme -load "defs.scm"`

- To exit scheme, type `CTRL-c q` or `(exit)`

Information about MIT Scheme

MIT Scheme Reference by Chris Hanson and others

`http://sicp.ai.mit.edu/Spring-2001/`
`    manuals/scheme-7.5.5/doc/scheme_toc.html`

From onyx, there is a local version at

`/usr/local/manuals/schemeref/schemeref-divided/scheme_toc.html`

Running Scheme from inside vi

You can run Scheme from inside vi by placing the following macro definition in your `.exrc` (or `.vimrc`) file

```
map @ :!scheme -load % ^M
```

The `^M` is entered as CTRL-v CTRL-m. This will automatically load the file being edited into the scheme interpreter when the **@** key is pressed. Remember to save your file before running the macro.