Programming in Lisp/Scheme

BY YOUJUN HU

Institute of Plasma Physics, Chinese Academy of Sciences, China
Email: yjhu@ipp.cas.cn

1 Introduction

`Lisp` is the second-oldest high-level programming language (after `Fortran`). Richard Stallman said in one of his articles: "The most powerful programming language is `Lisp`. If you don't know `Lisp` (or its variant, `Scheme`), you don't know what it means for a programming language to be powerful and elegant. Once you learn `Lisp`, you will see what is lacking in most other languages." This made me curious and motivated me to learn `Lisp` begining from September of 2015. "Powerful" in the world of Turing-complete programming languages means that you can do more with the language in a finite amount of time.

(Another big name in Lisp community is Paul Graham, who wrote many inspiring essays on Lisp[5]. I read most of his essays and found they provide new insights on many issues, not just programming.)

I primarily use `Guile`, a GNU implementation of the `Scheme`. I also use `Racket` (previously known as PLT scheme), another famous implementation. `Scheme` is mostly functional (but not purely functional). A program being functional means that the program accomplishes its task by evaluating various expressions. Being functional also means that a function itself is a type of value, which can be, for example, stored in a variable, passed as an argument to a function, returned from a function invocation, just like a traditional value such as an integer.

All function invocations (including basic arithmetic operations) in Lisp are based on the parenthezed-prefix-notation, i.e., `(operator operands1 operands2)`. For example `(+ 1 2)` is a function call. This notation is also used for other syntax structures (called special forms) in Lisp such as the `if` conditional structure:

```
(if my_test exp1 exp2)
```

A subexpression in the above can also be another function call (or special form), which will introduce another pair of round-brackets. This kind of nest can be of arbitary levels. As a result, Lisp programs are full of round-brackets, which makes Lisp source codes look messy and not so readable if without strong support from a text editor (e.g. automatic indenting).

The parentheses (round-brackets) are also used as the external representation of the list data structure in Lisp. Therefore, people often say that souce codes of Lisp take the same form as the list data structure.

Multiple whitespaces (including line-breaks) are equivalent to a single whithespace in Lisp.

2 Lisp/Scheme interpreter

On my Ubuntu desktop computer, I used `aptitude` to install Common-Lisp, `clisp`. Later I switched to `guile` and Racket. I will primarily use `guile` in this note.

2.1 Run Lisp code interactively

In a terminal, type `guile` to invoke the `scheme` interpreter:

```
yj@pic:~$ guile
guile> (display "Hello World! \n")
Hello World!
```

Users interact with a Scheme interpreter though a *read-evaluate-print loop* (*REPL*). Scheme waits for the user to type an expression, reads it, evaluates it, and prints the return value.

The readline support for `guile` command line is not loaded by default and can be loaded and activated by adding the following two lines of code in the init file `~/.guile`:

```
(use-modules (ice-9 readline))
(activate-readline)
```

2.2 Run Lisp scripts

2.2.1 Method 1

Create a file called `h.scm` with the following content:

```
;This line is a comment
(display "Hello, World! \n")
```

Then run the above script at command line:

```
$ guile -s h.scm
```

2.2.2 Method 2

Like Perl, Python, or any shell, Guile can interpret script files. A Guile script is simply a file of Scheme code with some extra information at the beginning which tells the operating system how to invoke Guile, and then tells Guile how to handle the Scheme code. Add the interpreter at the beginning of `h.scm`:

```
#!/usr/bin/guile -s
!#
```

and then make the file executable

```
chmod u+x h.scm
```

Then we can run it directly by `./h.scm` (assuming `h.scm` is in the current directory).

Note that there is an additional line starting with `!#`. This is because `#!..!#` indicates multiline comments (block comments) in `guile`. When bash sees `#!` at the first line of a file, it considers the name following `#!` as an interpreter and invoke the interpreter with the name of the present file as an argument. After `guile` gets control, it reads the file again from the begining. In this case, `guile` see `#!..!#`, which is block comment and so is ignored by `guile`. That is the reason why another line with `!#` is needed in the script.

3 Type of values

In dynamically typed languages such as Scheme, types are used to categorize values, rather than variables. The term "value" can be used exchangeably with "data", which is often used when making a comparison with "program". Scheme provides the following value types:

- Simple value types: boolean, number, char, symbol.

- Compound value types: string, vector, pair, procedure, port

Type of a value can be tested by the corresponding predicates:

```
boolean? number? char? symbol? string?
vector?  pair?  procedure? port?
```

Boolean: `#f` is logical false, `#t` is logical true. Although there is a separate boolean type, any Scheme value can be used as a boolean value for the purpose of a conditional test: all values count as true in such a test except for `#f`.

number–>complex–>real–>rational–>integer. Scheme numbers can be integers (eg, `42`), rationals (`22/7`), reals (`3.1416`), or complex (`2+3i`). In scheme, an integer is a rational, is a real, is a complex number, is a number. Predicates exist for testing the various kinds of numberness:

```
(number? 42)       =>  #t
(number? #t)       =>  #f
(complex? 2+3i)    =>  #t
(real? 2+3i)       =>  #f
(real? 3.1416)     =>  #t
(real? 22/7)       =>  #t
(real? 42)         =>  #t
(rational? 2+3i)   =>  #f
(rational? 3.1416) =>  #t
(rational? 22/7)   =>  #t
(integer? 22/7)    =>  #f
(integer? 42)      =>  #t
```

Integers need not be specified in decimal (base 10) format. They can be specified in binary by prefixing the numeral with `#b`. Thus `#b1100` is the number twelve. The octal prefix is `#o` and the hex prefix is `#x`. (The optional decimal prefix is `#d`.)

Character data are represented by prefixing the character with `#\`. Thus, `#\c` is the character `c`. Some non-graphic characters have more descriptive names, e.g., `#\newline`, `#\tab`. The character for space can be written `#\ `, or more readably, `#\space`. The character predicate is `char?`

Symbol is a sequence of characters that can not be confused with other values, namely, characters, booleans, numbers, compound data. Thus, `this-is-a-symbol`, `i8n`, `<=>`, and `$!#*` are all symbols; whereas `6`, `-i` (a complex number), `#t`, `"this-is-a-string"`, and `(barf)` (a list) are not. Symbols and strings are separate data types.

Symbols are also atomic, we cannot split them apart. The primary operation we perform on symbols is comparison (determining whether two symbols are the same).

A symbol denotes only itself. Unlike other simple values (booleans, characters, numbers), symbols are not self-evaluating. This design is because of the practice that a sequence of characters that is a symbol is reserved by Scheme as an identifier (rather than a value), and is evaluated to the value the identifier is bound (if it is bound, otherwise raises an error), rather than the symbol literal itself.

When we want to refer to something as a value (data) involved in a computation, rather than as a program (specifically, the name of some other value, an expression to be evaluated), we put an apostrophe (usually pronounced "quote") in front of it. In effect, by quoting something, we're telling Scheme to take it literally and without further interpretation or evaluation. You can quote many different data, not only limitted to symbols. For example:

```
(symbol? '(1 2 3)) ;=>#f
(symbol? 'sample)  ;=>#t
(symbol? '2)       ;=>#f
(integer? '2)      ;=>#t
```

In effect, an apostrophe introduces data i.e., values, which can be of any types. What follows the apostrophe and ends at a proper location (determined by Lisp syntax) is the data itself; apostrophe itself is not a part of the data. For the practical purpose of distinguishing between data and program for programmers, the apostrophe can be considered as a part of external representation of the data, as an indicator of data. For values of boolean, number, char and string, they are self-evaluating, and thus it is not necessary to quote them, but in order to have a sharp distinguishment between data and program, it is instructive to quote them.

To be unified with Lisp's parentheses prefix syntax, Lisp also introduces the quote special form:

```
(symbol? (quote sample))  ;=>#t
```

4 Compound data structure: string, vector, pair/list, procedure, port

4.1 String

string: a sequence of characters enclosed by double quotation markers is a string. Strings evaluates to themselves:

```
"hello"
=> "hello"
```

The characters in a given string can be individually accessed and modified. The procedure `string-ref` takes a string and a (0-based) index, and returns the character at that index:

```
(string-ref "Hello" 0)
=>  #\H
```

Other useful string methods include `string-append`, `make-string`, `string-set!`.

4.2 Vectors

Vectors are sequences like strings, but their elements can be anything, not just characters. Indeed, the elements can be vectors themselves, which is a good way to generate multidimensional vectors.

Scheme's representation of a vector value: a sharper sign `#` followed by the vector's contents enclosed in parentheses. eg.

```
#(0 1 2)
```

Here's a way to create a vector of the first five integers:

```
(vector 0 1 2 3 4)
```

```
=>  #(0 1 2 3 4)
```

In analogy with `make-string`, the procedure `make-vector` makes a vector of a specific length:

```
(make-vector 5)
```

The procedures `vector-ref` and `vector-set!` access and modify vector elements. The predicate for checking if something is a vector is `vector?`.

4.3 Pair and list

The following pair:

```
(a .  (b .  (c .  d)))
```

is equivalent to the following list:

```
(a b c d)
```

List predicat is `list?`

4.4 Procedure

This value is returned by a lambda expression (discussed later). Procedure values do not have external representations for readin syntax.

4.5 Port

Port value do not have external representations for readin syntax.

5 Expressions

Like other functional programming languages, basic elements of Scheme code are expressions. Expressions can be evaluated, producing a value. The most fundamental expressions are **literal expressions**:

```
#t  ;=>#t
23  ;=>23
```

This notation means that the expression `#t` evaluates to `#t`, that is, the value for "true", and that the expression `23` evaluates to a number object representing the number `23`.

Variable reference is another simple expression, e.g.,

```
(define a 10)
a    ;an expression evaluates to 10
```

Compound expressions are formed by placing parentheses around their subexpressions (multiple subexpressions are separated by white-spaces). The first subexpression identifies an operation; the remaining subexpressions are operands to the operation:

```
(+ 23 42)       ;=>65
(+ 14 (* 23 42)) ;=>980
(remainder 5 2) ; => 1
```

Compound expressions can be nested, as the second example shows. As these examples indicate, compound expressions in Scheme are always written using the same prefix notation: operator first then operands follow. As a consequence, the parentheses are needed to indicate structure. Another consequence is that, "superfluous" parentheses, which are often permissible in mathematical notation and also in many programming languages, are not allowed in Scheme. This is because adding a pair of superfluous parentheses to an expression corresponds to forming a new compound expression and letting Lisp treat the original expression as a sub-expression and use the value of this sub-expression as an operator and evaluate the compound expression (function invocation with no argument), the value of which is usually different from the original expression (if we are lucky to get a value at all).

Multiple sub-expressions of a compound expression are separated by white-space. In Scheme, line-ending is equivalent to whithe spaces, and multiple whitespaces or line-endings are equivalent to a single whitespace.

Not all expressions are valid programs. If you typed `(1 2)` at the Scheme listener, you will get an error because `(1,2)` is a list, which is valid data but is not an expression that can be evaluated to give an value.

Compound expressions are a subcategory of combinations (in Guile) or forms (in R5RS). This ditinguishment is necessary becasue, in scheme, some combinations are not considered to be expressions. One example is the variable binding and initialization structure:

```
(define x 23)
```

which is not expression. While definitions are not expressions, definitions and compound expressions exhibit similar syntactic structure:

```
(define x 23)
(* x 2)
```

While the first line contains a definition, and the second an expression, this distinction depends on the bindings for `define` and `*`. At the purely syntactical level, both are forms, and form is the general name for a syntactic part of a Scheme program. In particular, `23` is a subform of the form `(define x 23)`.

Scheme evaluates a list form by examining the first element, or head, of the form. If the head evaluates to a procedure, the rest of the form is evaluated to get the procedure's arguments, and the procedure is applied to the arguments. If the head of the form is a special form, how the remaing sub-form are evalued are determined by that special form. Some special forms are `begin`, `define`, and `set!`. `define` introduces and initializes a variable. `set!` assigns a new value to a variable. `begin` causes its subforms to be evaluated in order, the result of the entire form being the result of the last subform.

Empty combination () is considered to be an invalid program in Guile and Racket (it is still a valid data, i.e., empty list) because it is missing the procedure expression. In GNU clisp, empty list is a valid program, which evaluates to `NIL` (boolean value false).

All function calls (including basic arithmetic operations) in Lips are based on the parenthezed-prefix-notation, i.e., `(operator operands1 operands2)`. Many predefined operations of Scheme are provided not by syntax, but by variables whose values are procedures. The + operation, for example, which receives special syntactic treatment in many other languages, is just a regular identifier in Scheme, bound to a procedure that adds number objects. The following are some arithmetic operations:

```
(+ 2 3)  ;=> 5
```

```
(- 2 3)  ;=> -1
(* 2 3)  ;=> 6
(/ 2 3)  ;=> 2/3
(expt 2 3) ;=> 8
(quotient 5 2) ; => 2
(remainder 5 2) ; => 1
(/ 35 5) ; => 7
(/ 1 3) ; => 1/3
(exact->inexact 1/3) ; => 0.3333333333333333
(+ 1+2i  2-3i) ; => 3-1i
```

A list in lisp can be treated as program or data. If a list is given to a lisp interpreter, by default, the list is treated as program and lisp evaluates each elements of the list. To prevent a list from being evaluated (i.e, to treat them literally), we can use the `quote` special form. For example

```
guile>(quote (1 2 3))
(1 2 3)
guile> (quote foo)
foo
```

`quote` is called a "special form" because, unlike most other lisp operations, it doesn't evaluate its argument.

There is a shortcut (syntax sugar) which provides an alternative form of calling the `quote` procedure: instead of using the standard list form: a single quotation marker before an expression have the effect of preventing evaluation. For example

```
guile> '(f 2 3)
(f 2 3)
guile> 'foo
foo
```

6 Variables (pointers), values (objects), data type

6.1 Variable names

In Scheme, there are very few restrictions on what kind of sequence of characters can be used as a variable name. Any character sequence that can not be confused with Scheme's value types can be used a variable name.

The restriction on variable names in Lisp can be put in the following way: All literals that are symbols (i.e., the predicate `symbol?` return true) can be used as identifiers (variable names, names of locations, names of values).

6.2 Define variables and assignments

Scheme has latent as opposed to manifest types. Types are associated with objects(also called values) rather than with variables. (Some authors refer to languages with latent types as untyped, weakly typed or dynamically typed languages.) Other languages with latent types are Python, Ruby, Smalltalk, and other dialects of Lisp. Languages with manifest types (sometimes referred to as strongly typed or statically typed languages) include Fortran, C/C++, C#, Java, Haskell, and ML.

`Lisp` is a language with dynamically typed variables, which means the type of a variable can change during runtime and hence there is no type declaration for variables. More accurately, typing is associated with the value that a variable assumes rather than the variable itself. A variable (name) is a pointer pointing to a location where a data object (value) is stored.

In `guile`, a variable (a name) is bound to a value (data object) by using `define`

```
(define a 123)
(define a "hello")
```

More accurately speeaking, `define` bind a name with a location (if the name is not yet bound) and assign a value to that location. When a variable is bound to a location, its value can also be modified by using `set!`, e.g.,

```
(set! a  456)
```

For a bound variable, using `define` and `set!` are equivalent to each other. If we use `set!` on a unbound variable, the lisp will raise error message, reminding us that a variable must be first bound to a location (using `define`) before we can set its value.

6.3 Variable scope

`Scheme` uses lexical (static) scope. (Originally Lisp used dynamic scoping, Emacs lisp can swich between lexical and dynamic scoping by seting an option.) In lexical scope, each use of a variable is associated with a lexically apparent binding of that variable.

Scheme is a statically scoped language with block structure. To each place in a top-level program or library body where an identifier is bound, there corresponds a region of code within which the binding is visible. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a lambda expression, for example, then its region is the entire lambda expression. Every mention of an identifier refers to the binding of the identifier that establishes the innermost of the regions containing the use. If a use of an identifier appears in a place where none of the surrounding expressions contains a binding for the identifier, the use may refer to a binding established by a definition or import at the top of the enclosing library or top-level program (see chapter 7). If there is no binding for the identifier, it is said to be unbound.

In dynamic scoping, a function may reference local variables defined in the lexical scope of the calling unit, which means, if a programmer declares a variable within the lexical scope of a function, it is available to subroutines called from within that function. Originally, this was intended as an optimization; lexical scoping was still uncommon and of uncertain performance. Somebody asked RMS (Richard Stallman) why it was dynamically scoped when he was implementing emacs lisp and his exact reply was that lexical scope was too inefficient. Dynamic scoping was also meant to provide greater flexibility for user customizations. However, dynamic scoping has several disadvantages. It can easily lead to bugs in large programs, due to unintended interactions between variables in different functions.

With the constructs for local binding (let, let*, letrec, and letrec*), the Scheme language has a block structure like most other programming languages since the days of ALGOL 60. Readers familiar to languages like C or Java should already be used to this concept.

The most basic local binding construct is `let`. syntax: let bindings body

bindings has the form

```
( (variable1 init1) ...)
```

that is zero or more two-element lists of a variable and an arbitrary expression each. All variable names must be distinct. A `let` expression is evaluated as follows. All init expressions are evaluated. New storage is allocated for the variables. The values of the init expressions are stored into the variables.

The expressions in body are evaluated in order, and the value of the last expression is returned as the value of the let expression.

Local binding: 'me' is bound to "Bob" only within the (let ...)

```
(let ((me "Bob")  (you "Tom") )  me)  ;; => "Bob"
```

The other binding constructs are variations on the same theme: making new values, binding them to variables, and executing a body in that in the newly extended lexical context.

All objects created in the course of a Scheme computation, including procedures and continuations, have unlimited extent. No Scheme object is ever destroyed. The reason that implementations of Scheme do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they can prove that the object cannot possibly matter to any future computation. Other languages in which most objects have unlimited extent include C#, Java, Haskell, most Lisp dialects, ML, Python, Ruby, and Smalltalk

7 Conditionals

"A conditional is an if-then-else construct. We take these for granted now. They were invented in the course of developing Lisp. (Fortran at that time only had a conditional goto, closely based on the branch instruction in the underlying hardware.)[5].

The conditionals in Scheme take the following forms:

```
(if test consequent alternate)
(if test consequent)
```

where `test`, `consequent`, and `alternate` are expressions. An `if` expression is evaluated as follows: first, `test` is evaluated. If it yields a true value, then `consequent` is evaluated and its values are returned. Otherwise `alternate` is evaluated and its values are returned. If `test` yields a false value and no `alternate` is specified, then the result of the `if` expression is unspecified.

Example:

```
(if (= 5 (+ 2 3)) "equal" "inequal")   ;==>"equal"
```

`if` is a special form because it does not automatically evaluate **all** of its arguments:

```
(if (= 0 1)  (/ 1 0)  (+ 2 3))          ;==> 5
```

Note that the `(/ 1 0)` is not evaluated. Another interesting example:

```
((if #f + *) 3 4)                  ;===>  12
```

In scheme standard[2], `if` construct is the primitive construct. There is a derived conditional, the `cond` construct:

```
(cond (test1 expr1)
      (test2 expr2)
```

```
  ...
(testN  exprN))
```

As soon as `cond` find a test that evaluates to true, then it evaluates the corresponding expression and return its value. The remaining tests and expressions are not evaluated. If none of the tests evaluate to ture, then the return value of `cond` is not defined. To get an "else" part , we provide a test that is gurrantee to be true as the last clause of the `cond` structure.

```
(cond (test1 expr1)
      (test2 expr2)
        ...
      (#t  exprN))
```

Consider the fact "else" part is often used, `guile` provide a syntax sugar which uses `else` as a keyword:

```
(cond (test1 expr1)
      (test2 expr2)
        ...
      (else  exprN))
```

If none of the tests evaluate to ture then `cond` evaluate the `else` part and return its value (the `else` part can be left off, but it's not good style, since this may make return value undefined).

Logical operators: `and`/`or`. For example:

```
(and #f #t)   ;=> flase
(or  #f #t)   ;=> true
```

8 Lambda expression

The lambda expression define a (anonymous) function and, when it is evaluated, return that function object as the returned value. Let us examine an example:

```
(lambda (x y) (+ x y))
```

which will return a function object that takes two arguments and what the function does is to sum the two arguments.

Note that function/procedure is just another type of value and the importance of `lambda` is that it can return this type of value. Using the word "lambda" instead of "funtion" or "procedure" is due to Lisp's history. In practice, when we see the word `lambda` in source codes, we can safely translate (in our mind) the word to "function".

The returned function by the above `lambda` call is unnamed. How do we use the returned function? We can use it wherever a function name can appear. For example

```
((lambda (x y) (+ x y)) 2 3)
```

or, equivalently, binding the function objet to a variable:

```
(define foo (lambda (x y) (+ x y)))
```

and then calling the function using the variable name:

```
(foo 2 3)
```

In the latter case, we define a unnamed function by using `lambda` and then bind it to the name `foo`. This amounts to that we are defining a named function. Since defining named functions is a very common task, `scheme` provides this special version of `define` that doesn't use `lambda` explicitly:

```
(define (foo x y) (+ x y))
```

which can be considered as an abbreviation for `(define foo (lambda (x y) (+ x y)))`, i.e., defining a unnamed function and then binding it to the variable `foo`.

8.1 Pass by value or reference?

Similar to C, `Scheme` function call uses pass by value (rather than by reference) with pointer semantics. Let us examine an example:

```
(define (func x)
        (define x 4)
        (* x 2))
(define a 3)
(func a)
(display a)     ;=> 3
```

8.2 General syntax of lambda expression:

`(lambda <formals> <body>)`

where `<formals>` have one of the following forms:

- `(variable_1 variable_2 . . . variable_n)` The procedure takes a fixed number of arguments; when the procedure is called, the values of actual arguments will be stored in newly allocated locations to which the formal arguments are bound.
- `variable` The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments is converted into a newly allocated list, and the list is stroed in a fresh location to which `variable` is bound.
- `(variable_1 ... variable_n . variable_n+1)` If a space-delimited period precedes the last variable, then the procedure takes n or more arguments, where n is the number of formal arguments before the period. The value stored in the binding of the last variable will be a newly allocated list of the actual arguments left over after all the other actual arguments have been matched up against the other fromal arguments.

8.3 Closure

What is important about `lambda` expression is that it returns an enviroment (e.g., values of the free variables in the lambda expression) to perform calculations and this environment can be changed and is remembered (persitent variables) by the closure. Let us see an example of a serial number generator:

```
(define make-serial-number-generator
  (lambda ()
    (let ((current-serial-number 0))
      (lambda ()
        (set! current-serial-number (+ current-serial-number 1))
        current-serial-number))))

(define entry-sn-generator (make-serial-number-generator))
(define entry-sn-generator2 (make-serial-number-generator))

(display (entry-sn-generator)) ;==>1
(newline)
(display (entry-sn-generator))  ;==>2
(newline)
(display (entry-sn-generator2)) ;==>1
(newline)
(display (entry-sn-generator2))  ;==>2
(newline)
```

This example create a procedure with a variable binding that is private to the procedure, like a local variable, but whose value persists between procedure calls.

When make-serial-number-generator is called, it creates a local environment with a binding for current-serial-number whose initial value is 0, then, within this environment, creates a procedure. The local environment is stored within the created procedure object and so persists for the lifetime of the created procedure.

Note that make-serial-number-generator can be called again to create a second serial number generator that is independent of the first. Every new invocation of make-serial-number-generator creates a new local let environment and returns a new procedure object with an association to this environment.

In summary, closure is the capture of an environment, containing persistent variable bindings, within the definition of a procedure. This is rather similar to the idea in some object oriented languages of encapsulating a set of related data variables inside an "object", together with a set of "methods" that operate on the encapsulated data. The following example shows how closure can be used to emulate the ideas of objects, methods and encapsulation in Scheme.

```
(define (make-account)
  (let ((balance 0))
    (define (get-balance)
      balance)
    (define (deposit amount)
      (set! balance (+ balance amount))
      balance)
    (define (withdraw amount)
      (deposit (- amount)))

    (lambda args
      (apply
        (case (car args)
```

```
      ((get-balance) get-balance)
      ((deposit) deposit)
      ((withdraw) withdraw)
      (else (error "Invalid method!")))
    (cdr args)))))
```

Scheme was one of the first languages to support procedures as objects in their own right. Procedures can be created dynamically, stored in data structures, returned as results of procedures, and so on. Other languages with these properties include Common Lisp, Haskell, ML, Ruby, and Smalltalk.

One distinguishing feature of Scheme is that continuations, which in most other languages only operate behind the scenes, also have "first-class" status. First-class continuations are useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines.

In Scheme, the argument expressions of a procedure call are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not. C, C#, Common Lisp, Python, Ruby, and Smalltalk are other languages that always evaluate argument expressions before invoking a procedure. This is distinct from the lazy-evaluation semantics of Haskell, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure.

The power of lambda expression is better shown when they are used as an anonymous function inside another function.

9 Recursion

Let us see some simple examples of recursion.

```
(define (factorial n)
  (if (= 0 n)
      1
      (* n (factorial (- n 1)))))
```

which is the classic factorial function.

```
;;  (double-each '(1 3 4))  => (2 6 8)
(define (double-each s)
  (if (null? s)
      '()
      (cons (* 2 (car s))
            (double-each (cdr s)))))
```

which doubles each number in a list.

Rules for writing recursive functions:

1. Know when to stop (the base case)

2. Decide how to take one step towards the base case

3. Phrase the solution in terms of one step, and a smaller version of the original problem.

For numbers, the base case is usually a small integer constant, and a smaller version of the problem is something like `n-1`.

For lists, the base case is usually the empty list, and a smaller version of the problem is usually the rest (i.e., `cdr`) of the list. Here is a template for most recursive functions:

```
(define (fn args)
  (if base-case
      base-value
    (compute-result (fn (smaller args)))))
```

Implementations of Scheme must be properly tail-recursive, which means that the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure. Thus with a properly tail-recursive implementation, iteration can be expressed using the ordinary procedure-call mechanics without runtime penality, so that special iteration constructs are useful only as syntactic sugar.

9.1 Loop

Scheme has no expressions designed for looping at a general level. The only easy way to do this is recursion, which forces a particular mindset but is actually a natural way for iteratation. The following example illustrates a Scheme script that iterates from 0 to 9, then prints done. This example uses what in Scheme is called tail recursion. Note that the function being defined is invoked at the end of the block. This recursion is called tail recursion. In traditional languages, this recursion eats away at the stack to maintain a history of the calls; in Scheme, it's different. The last call (the tail) simply invokes the function without any procedure call or stack maintenance overhead. This is often stated in the following words: Implementations of Scheme must be properly tail-recursive

```
(let countup ((i 0))
  (if (= i 10) (begin (display "done") (newline))
               (begin  (display i)   (newline)   (countup (+ i 1)))))
```

Scheme is "properly tail recursive", meaning that tail calls or recursions from certain contexts do not consume stack space or other resources and can therefore be used on arbitrarily large data or for an arbitrarily long calculation.

10 Higher order function

`Scheme` supports functions as *first class citizens*, which means `Scheme` supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures. A programming language is said to have first-class functions if it treats functions as first-class citizens.

Scheme gets much of its expressiveness and capacity for abstraction by supporting an **applicative** programming style using *higher order functions* – functions that take other functions as arguments, or that return functions as the result. All other functions are **first-order** functions. In mathematics **higher-order** functions are called **operators** or **functionals**. The differential operator in calculus is a common example, since it maps a function to its derivative, also a function.

First-class functions are a necessity for the functional programming style, in which the use of higher-order functions is a standard practice. A simple example of a higher-ordered function is the map function, which takes, as its arguments, a function and a list, and returns the list formed by applying the function to each member of the list. For a language to support map, it must support passing a function as an argument. Let us see an example:

```
(map (lambda (x) (* x 2)) '(1 2 3))
```

which doubles all elements in a numerical list. Let us define our own version of `map`, `my-map`, which is a higher order function:

```
(define (my-map fun alist)
(if (null? alist)
      '()
      (cons (fun (car alist)) (my-map fun (cdr alist))))
)
```

There are certain implementation difficulties in passing functions as arguments or returning them as results, especially in the presence of non-local variables introduced in nested and anonymous functions. Historically, these were termed the funarg problems, the name coming from "function argument". In early imperative languages these problems were avoided by either not supporting functions as result types (e.g. ALGOL 60, Pascal) or omitting nested functions and thus non-local variables (e.g. C). The early functional language Lisp took the approach of dynamic scoping, where non-local variables refer to the closest definition of that variable at the point where the function is executed, instead of where it was defined. Proper support for lexically scoped first-class functions was introduced in Scheme and requires handling references to functions as closures instead of bare function pointers, which in turn makes garbage collection a necessity.

A function object is also known as a closure.

Memory management is one of Scheme's strong points. Unlike languages like C, programmers of scheme do not have to deal with complicated memory management. Scheme uses garbage collection, so you do not have to worry about. This means that the Scheme environment frees memory for you when the last reference to an object is destroyed. Simply stop referring to an object (e.g. re-assigning a variable) and the value is marked as free and garbage collected. Languages like Java also do this. Garbage collection is a very powerful feature. The source of many bugs (in languages that you must manually manage memory in) is memory management. Why have every program deal with memory management manually when you can do it all in one place? Of course, many people will disagree, but whether garbage collection is the answer is usually something that is true for some apps but false for others.

1.7 Procedure calls and syntactic keywords

Whereas `(+ 23 42)` and `((lambda (x) (+ x 42)) 23)` are all examples of procedure calls, `lambda` and `let` forms are not. This is because `let`, even though it is an identifier, is not a variable, but is instead a syntactic keyword. A form that has a syntactic keyword as its first subform obeys special rules determined by the keyword. The `define` identifier in a definition is also a syntactic keyword. Hence, definitions are also not procedure calls.

The rules for the `lambda` keyword specify that the first subform is a list of parameters, and the remaining subforms are the body of the procedure. The rules for the `let` keyword specify that the first subform is a list of binding specifications, and the remaining subforms constitute a body of expressions.

Procedure calls can generally be distinguished from these special forms by looking for a syntactic keyword in the first position of an form: if the first position does not contain a syntactic keyword, the expression is a procedure call. (So-called identifier macros allow creating other kinds of special forms, but are comparatively rare.) The set of syntactic keywords of Scheme is fairly small, which usually makes this task fairly simple. However, it is possible to create new bindings for syntactic keywords

5.8.7 The R5RS syntax-rules System

R5RS defines an alternative system for macro and syntax transformations using the keywords `define-syntax`, `let-syntax`, `letrec-syntax` and `syntax-rules`. The main difference between the R5RS system and the traditional macros is how the transformation is specified. In R5RS, rather than permitting a macro definition to return an arbitrary expression, the transformation is specified in a pattern language that (1) does not require complicated quoting and extraction of components of the source expression using caddr etc. (2) is designed such that the bindings associated with identifiers in the transformed expression are well defined, and such that it is impossible for the transformed expression to construct new identifiers. The last point is commonly referred to as being hygienic, i.e., the R5RS syntax-case system provides hygienic macros.

In Guile, the syntax-rules system is provided by the `(ice-9 syncase)` module. To make these facilities available in your code, include the expression `(use-syntax (ice-9 syncase))`. For example:

```
#!/usr/bin/guile -s
!#
(use-syntax (ice-9 syncase))
(define-syntax def
  (syntax-rules ()
    ( (def    f (p ...)  body)
      (define (f p ...)  body) )
    )  )

(def f(x) (+ x 1))
     (display (f 3))
     (newline)
```

Pattern Language: The syntax-rules pattern language.

Define-Syntax: Top level syntax definitions.

Let-Syntax: Local syntax definitions.

11 I/O

Reading

Writing

File ports

Automatic opening and closing of file ports

String ports

12 System interface

Useful Scheme programs often need to interact with the underlying operating system.

12.1 Checking for and deleting files

`file-exists?` checks if its argument string names a file. `delete-file` deletes its argument file. These procedures are not part of the Scheme standard, but are available in most implementations (e.g. `gule`). These procedures work reliably only for files that are not directories. (Their behavior on directories is dialect-specific.)

12.2 Calling operating-system commands

The system procedure executes its argument string as an operating-system command.1 It returns true if the command executed successfully with an exit status 0, and false if it failed to execute or exited with a non-zero status. Any output generated by the command goes to standard output.

```
(system "ls")
;lists current directory

(define fname "spot")

(system (string-append "test -f " fname))
;tests if file ‘spot’ exists

(system (string-append "rm -f " fname))
;removes ‘spot’
```

The last two forms are equivalent to

```
(file-exists? fname)
(delete-file fname)
```

12.3 Getting environment variables

The `getenv` procedure returns the setting of an operating-system environment variable. Eg,

```
(getenv "HOME")
=>  "/home/yj"
```

reference: https://ds26gte.github.io/tyscheme/

13 A complete Example of using scheme in numerical simulation

Let us consider using Runge-Kuta method to integrate a system of ordinary differential equations (ODEs). First let us construct some basic facilities (some data structures and operations on them) which will be used in our simulation. We use a Scheme vector to store a state of the system. If we have a system of 3 ODEs, then the vector will be of size 3, with one element of the vector standing for the value of one variable of the 3 ODEs.

```
(define (generate-vector size proc)
  (let ((ans (make-vector size)))
    (letrec ((loop
              (lambda (i)
                (cond ((= i size) ans)
                      (else
                       (vector-set! ans i (proc i))
                       (loop (+ i 1)))))))
      (loop 0))))

(define (elementwise f)
  (lambda vectors
    (generate-vector
     (vector-length (car vectors))
     (lambda (i)
       (apply f
              (map (lambda (v) (vector-ref v i)) vectors))))))

(define add-vectors (elementwise +))

(define (scale-vector s)
  (elementwise (lambda (x) (* x s))))
```

Here `vectors` is a list of vectors.

Next, define the core procedure which uses runge-Kutta method to push a given system one step forward:

```
(define (runge-kutta-4 f h)
  (let ((*h (scale-vector h))
        (*2 (scale-vector 2))
        (*1/2 (scale-vector (/ 1 2)))
        (*1/6 (scale-vector (/ 1 6))))
    (lambda (y) ;; y is a system state
      (let* ((k0 (*h (f y)))
             (k1 (*h (f (add-vectors y (*1/2 k0)))))
             (k2 (*h (f (add-vectors y (*1/2 k1)))))
             (k3 (*h (f (add-vectors y k2)))))
        (add-vectors y
                     (*1/6 (add-vectors k0
                                        (*2 k1)
                                        (*2 k2)
                                        k3)))))))
```

Here `h` is time-step size, `f` is a function that calculates the time derivates for a given system state. Specifically, function invocation `(f y)`, with the `y` being a vector standing for the system state, returns the time derivates.

Next, let us consider a specific system, a damped oscillator:

$$\frac{dv_C}{dt} = -\frac{i_L}{C} - \frac{v_C}{RC}, \tag{1}$$

$$\frac{di_L}{dt} = \frac{v_C}{L}. \tag{2}$$

The above time derivatives are implemented as:

```
(define (damped-oscillator R L C)
  (lambda (state)
    (let ((Vc (vector-ref state 0))
          (Il (vector-ref state 1)))
      (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
              (/ Vc L)))))
```

Finally, let us implement the time iteration:

```
(define (integrate-system system-derivative initial-state h nsteps)
  (let ((next (runge-kutta-4 system-derivative h))
        (old_state  initial-state)
        (new_state  initial-state))
    (letrec ((countdown (lambda (i)
                          (if (= i 0) 'liftoff
                              (begin
                                (set! new_state (next old_state))
                                (display (vector-ref new_state 0))
(newline)
                                (set! old_state new_state)
                                (countdown (- i 1)))))))
      (countdown nsteps))))
```

And invoke the above procedure for a specific damped oscillator with $R = 10000$, $L = 1000$, $C = 0.01$, initial state vector `#(1 0)`, time-step size $h = 0.01$, number of time-steps $n_{\text{step}} = 5000$:

```
(integrate-system (damped-oscillator 10000 1000 .001) '#(1 0) 0.01 5000)
(display "finish")
(newline)
```

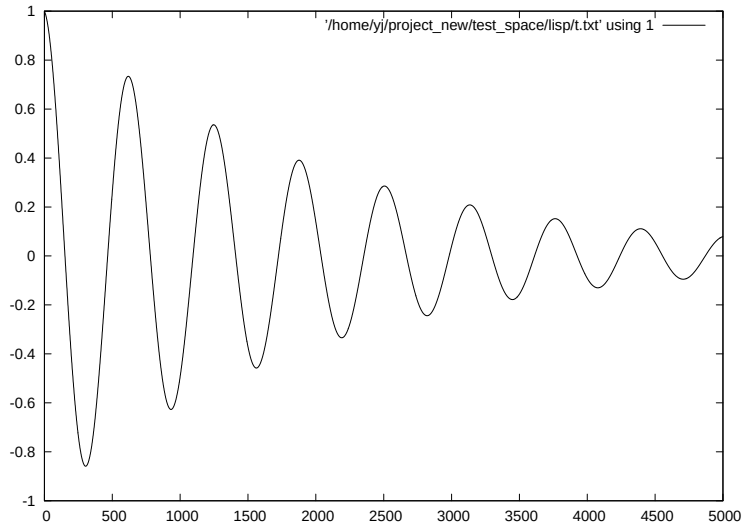
The above example is a revised version of the example given in R7RS[3]. Store the above codes in a file and run it in a linux shell:

```
$ guile -s rk_yj.scm > t.txt
```

Then plot the result stored in `t.txt` by using `GNUplot`:

```
   This is a TeXmacs interface for GNUplot.

GNUplot] plot '/home/yj/project_new/test_space/lisp/t.txt' using 1 w l
```



GNUplot]

14 Readability of **LISP** source code

Each function call in LISP source code is organized as a (nested) list. Since each list is delimited by a pair of parentheses (), for a nested list with several depth, the number of parenthesis will increase to a level that makes many people consider LISP as weird/unreadable at first glance. The reason is obvious: by intuition, people do not distinguish a nested structure by parentheses. People's intuition tends to distinguish logical structure through the spatial structure, e.g. line break and indent. This is the reason why Python uses indent to delimit structure, which increase the readability of the source code.

There is no difficulty for a computer to recognize the structure whatever delimiting marks a language chooses to use. Source code editors of LISP can transform the structure defined by the parenthesis to the appropriate indent loved by human's eyes and brain (the messy parentheses are still there, just do not bother with them and occasionally use them to get useful information for the structures.)

As mentioned above, Lisp source code is a list, which is a one-dimensional structure, and line-breaks are equivalent to white spaces. We can add new lines appropriately to provide vertical spacial structure in order to enhance readability.

two things will vastly improve your experience with Emacs and Guile:

The first is Taylor Campbell's Paredit. You should not code in any dialect of Lisp without Paredit. (They say that unopinionated writing is boring—hence this tone—but it's the truth, regardless.) Paredit is the bee's knees.

When developing these notes, I read the following materials: [4][1]

https://people.eecs.berkeley.edu/~bh/ssch0/preface.html

A misc

, (2) program and data are equivalent, which makes it easy to write Scheme programs that process/produce other programs, e.g. compilers, structure editors, debuggers, etc.

Lisp code is easy to write, but hard to read by pepole other than the code author (due to using macoros)

Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running

Metaprogramming can be used to move computations from run-time to compile-time, to generate code using compile time computations, and to enable self-modifying code. The language in which the metaprogram is written is called the metalanguage. The language of the programs that are manipulated is called the attribute-oriented programming language. The ability of a programming language to be its own metalanguage is called reflection or "reflexivity". Reflection is a valuable language feature to facilitate metaprogramming.

Metaprogramming was popular in the 1970s and 1980s using LISP languages. LISP hardware machines were popular in the 1980s and enabled applications that could process code.

A large number of programmers would have a tendency to learn a new programming language whenever they get a chance. Typically, decent programmers can pick up a new Language and write nontrivial programs in that language within a few days. This high efficiency of learning a new language is achieved by explicitly asking ourselves several questions about general language features:

static type or dynamic type or mixed?

what is the primitive type (e.g., real and integer numbers) and compound type (e.g., arrays and lists),

what is the syntax for name binding (i.e., variable/function definition)?

what is the syntax of calling functions?

lexical scope or dynamic scope? (most languages adopts lexical scope),

the flow control: what is the syntax for conditional structures and loop structures?

With these questions in mind, we can quickly find the answers by searching online or a handbook of that language. Then we can write codes and test the syntax with a compiler/interpreter.

https://schemers.org/Documents/Standards/R5RS/HTML

Scheme has latent (as opposed to manifest) types. Types are associated with values (also called objects) rather than with variables. (Some authors refer to languages with latent types as weakly typed or dynamically typed languages.) Other languages with latent types include python and javascript. Languages with manifest types (sometimes referred to as strongly typed or statically typed languages) include Fortran, and C.

Arguments to Scheme procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the procedure needs the evaluation result or not. ML, C, and APL are three other languages that always pass arguments by value. This is distinct from the lazy-evaluation semantics of Haskell, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure.

Scheme's model of arithmetic is designed to remain as independent as possible of the particular ways in which numbers are represented within a computer. In Scheme, every integer is a rational number, every rational is a real, and every real is a complex number. Thus the distinction between integer and real arithmetic, which is important to many programming languages, does not appear in Scheme. In its place is a distinction between exact arithmetic, which corresponds to the mathematical ideal, and inexact arithmetic on approximations. As in Common Lisp, exact arithmetic is not limited to integers.

Note that the sequence of characters `(+ 2 6)` is not an external representation of the integer 8, even though it is an expression evaluating to the integer 8; rather, it is an external representation of a three-element list, the elements of which are the symbol + and the integers 2 and 6. Scheme's syntax has the property that **any sequence of characters that is an expression is also the external representation of some object.** This can lead to confusion, since it may not be obvious out of context whether a given sequence of characters is intended to denote data or program, but it is also a source of power, since it facilitates writing programs such as interpreters and compilers that treat programs as data (or vice versa).

An identifier may name a type of syntax, or it may name a location where a value can be stored. An identifier that names a type of syntax is called a syntactic keyword and is said to be bound to that syntax. An identifier that names a location is called a variable and is said to be bound to that location. The value stored in the location to which a variable is bound is called the variable's value. By abuse of terminology, the variable is sometimes said to name the value or to be bound to the value. This is not quite accurate, but confusion rarely results from this practice.

Certain expression types are used to create new kinds of syntax and bind syntactic keywords to those new syntaxes (these expression types are called keyword binding constructs.), while other expression types create new locations and bind variables to those locations (these expression types are called binding constructs).

The most fundamental of the variable binding constructs is the lambda expression, because all other variable binding constructs can be explained in terms of lambda expressions. The other variable binding constructs are let, let*, letrec, and do expressions.

In Lisp, list delimiter is chosen to be parentheses, i.e., round brackets (for comparison, square brackets are chosen in python). The whitespace is chosen as the saperator between different elements of a list (cf., commas are chosen by python and whitespace is ignored by the interpreter). In lisp, a newline is equivalant to a whitespace. The following is a lisp list:

```
(1 2 3 a b c)
```

For comparison, the following is a python list:

```
[1, 2, 3, a, b, c]
```

Fortran's new array constructor (introduced in Fortran 2003) can also adopt the python style, e.g.

```
integer :: a(3), i
a=[1, 2, 3]
a=[(i, i=1,3)] !implied do-loop
```

The standard form of fortran array constructor is `(/ ... /)`, which is hard to remember and read.

The so-called "lambda calculus" is a formal system for expressing computation based on function abstraction and application using variable binding and substitution. Althouth the name contains "calculus", the lambda calculus has nothing to do with the calculus in mathematics (i.e., integration and differential). Then why the name "calculus"? Is the name here to frighten newbies? Maybe paritally yes. The remaining part, I guess, is related to the applicative programming style using *higher order functions* – functions that take other functions as arguments. In mathematics **higher-order** functions are called **operators** or **functionals**. The differential operator in calculus is a common example, since it maps a function to its derivative, which is also a function. Due to this similarity with the differential operators in calculus, this formal computational system is called "lambda calculus", where lambda can be understood as "function".

The lambda calculus can be called as the smallest programming language of the world. It gives the definition of what is computable. Anything that can be computed by lambda calculus is computable. It provides a theoretical framework for describing functions and their evaluation.

Do not get frightened by the fancy/mysterious nonmenclatures. Calm down and assume that they probably refer to a very simple thing that you already understand. The author use them maybe because he/she want to impress readers. They are simple concepts in disguise.

`Scheme` would still be useful as a notation for expressing computational methods even in the absence of a mechanical implementation.

One of the great simplifications of `Scheme` is that a procedure is just another type of value, and that procedure values can be passed around and stored in variables in exactly the same way as, for example, strings and lists.

a function that, when invoked, return a function.

a function that return a function upon invocation.

History

Alonzo Church formalized lambda calculus, a language based on pure abstraction, in the 1930s. Lambda functions are also referred to as lambda abstractions, a direct reference to the abstraction model of Alonzo Church's original creation.

Lambda calculus can encode any computation. It is Turing complete, but contrary to the concept of a Turing machine, it is pure and does not keep any state.

Functional languages get their origin in mathematical logic and lambda calculus, while imperative programming languages embrace the state-based model of computation invented by Alan Turing. The two models of computation, lambda calculus and Turing machines, can be translated into each another. This equivalence is known as the Church-Turing hypothesis.

Functional languages directly inherit the lambda calculus philosophy, adopting a declarative approach of programming that emphasizes abstraction, data transformation, composition, and purity (no state and no side effects). Examples of functional languages include Haskell, Lisp, or Erlang.

By contrast, the Turing Machine led to imperative programming found in languages like Fortran, C, or Python.

The imperative style consists of programming with statements, driving the flow of the program step by step with detailed instructions. This approach promotes mutation and requires managing state.

The separation in both families presents some nuances, as some functional languages incorporate imperative features, like OCaml, while functional features have been permeating the imperative family of languages in particular with the introduction of lambda functions in Java, or Python.

Python is not inherently a functional language, but it adopted some functional concepts early on. In January 1994, map(), filter(), reduce(), and the lambda operator were added to the language.

Operation on List Data structure.

List is one the built in data type in `Scheme`. Lists in `Scheme` can contain items of different types:

```
(1 1.5 x (a) “hello”)
```

The intrinsic functions that create or operate on a list: `list`, `car`, `cdr`, `cons`, and `append`.

```
(list 1 2 3)        ==>create a list

(car (list 1 2 3))  ==> choose the first element of a list

(car '(1 2 3))      ==> the same as the above

(cdr '(1 2 3))      ==> create a list by excluding the first element
(cons 'foo '(1 2 3))  ==> add a new cell to a list:
(append '(1 2) '(3)) ===>concatenate two or more lists==>(1 2 3)
```

The process that led to the R6RS standard brought a split in the Scheme community to the surface. The implementors that wrote R6RS considered that it was impossible to write useful, portable programs in R5RS, and that only an ambitious standard could solve this problem. However, part of the Scheme world saw the R6RS effort as too broad, and as having included some components that would never be adopted by more minimalistic Scheme implementations. This second group succeeded in taking control of the official Scheme standardization track and in 2013 released a more limited R7RS, essentially consisting of R5RS, plus a module system. Guile supports R7RS also.

As a Scheme program runs, values of all types pop in and out of existence. Sometimes values are stored in variables, but more commonly they pass seamlessly from being the result of one computation to being one of the parameters for the next.

Scheme programmers prefer to avoid assignment statements because assignment is a mutating operation, which is not prefered in (pure) functional programming pardigam. In most cases, values pass seamlessly from being the result of one computation to being one of the parameters for the next, rather than being stroed in an temporary variable.

Part of what we mean when we talk about "creating a variable" is in fact establishing an association between a name, or identifier, that is used by the Scheme program code, and the variable location to which that name refers. Although the value that is stored in that location may change, the location to which a given name refers is always the same.

Why is not Lisp booming now that there is an AI boom in 2010s?

The reason is very simple, Lisp was created for the school of Symbolic AI, and the AI that is growing in popularity right now, is a totally different school, the school of Machine Learning, which is a highly numerical domain (numerical statistics). You could say that Symbolic AI looks for a deductive approach while Machine Learning an Inductive approach.

Lisp is definitely no longer the language for AI, because AI itself has moved into a highly numerical domain which has traditionally been more of stronghold of C/C++. With current emphasis on GPU computing for maximizing compute power, C/C++ is the right vehicle to build AI systems because GPU computing is still a very low-level exercise.

Additionally, for programmers seeking a higher level of abstraction (by using libraries), Python has become the language of choice, because of its accessibility and widespread support, and the necessary packages and bindings becoming available for compute intensive numeric operations.

DrScheme automatically indents according to the standard style when you type Enter in a program or REPL expression. For example, if you hit Enter after typing (define (greet name), then DrScheme automatically inserts two spaces for the next line. If you change a region of code, you can select it in DrScheme and hit Tab, and DrScheme will re-indent the code (without inserting any line breaks). Editors like Emacs offer a Scheme mode with similar indentation support.

Line breaks and indentation are not significant for parsing Scheme programs, but most Scheme programmers use a standard set of conventions to make code more readable. For example, the body of a definition is typically indented under the first line of the definition. Identifiers are written immediately after an open parenthesis with no extra space, and closing parentheses never go on their own line.

Bibliography

[1] Alan Borning. Programming Languages . https://courses.cs.washington.edu/courses/cse341/03wi/scheme/, 2013. [Online; accessed 16-Feb-2019].

[2] Richard Kelsey et al. Revised5 Report on the Algorithmic Language Scheme. https://schemers.org/Documents/Standards/R5RS/HTML/r5rs.html, 1998. [Online].

[3] Steven Ganz et al. Revised7 Report on the Algorithmic Language Scheme. https://small.r7rs.org/attachment/r7rs.pdf, 2013. [Online].

[4] Free Software Foundation. GNU guile manual. https://www.gnu.org/software/guile/manual/, 2019. [Online].

[5] Paul Graham. What made lisp different. http://www.paulgraham.com/diff.html.