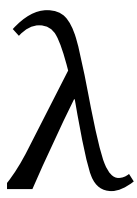

A Tour of Scheme in Gambit

Many thanks to Per Eckerdal, Christian Jaeger, Marc Feeley, Gottfrid Svartholm Warg.

Revision 1.

© 2008 Mikael More

Contents

Introduction

This is a brief introduction to The Scheme Programming Language, rather a how-to or guided tour than a reference or school book[1]. It's about giving you an intuitive feel about it, and about answering your most basic practical questions.

The language is very well standardized. There are a few different common standards, those are R5RS, R6RS, R4RS and IEEE. We'll work with R5RS. It is the strongest in combining being widespread, simple and new.

Scheme is maybe the language with the highest number of industry-quality environments.[2] They vary in many different aspects. They are like engines, there's small engines, big engines, engines that burn gasoline, those that burn hydrogen, they have different cylinder counts, there's fast, etcetra. You get the point. And there's manifoldness with a reason: their uses are manifold too. Cars, jet planes, hobby boats.

We will go with Gambit Scheme. It executes code equally fast to or faster than the mainstream development environments, it has 20 years on its neck, it is robust, and, as a system it is very mature, that is, it is very well thought through. Also, it has good debugging features, it's convenient to use, it can execute a million threads of execution in parallel on a desktop computer, and its feature set should be considered more can complete.

The big differences between Scheme environments are about whether they compile, compile incrementally or interpret, how they do error handling, work with the network, the operating system, libraries external to the Scheme environment, binary data, and to some extent characters beyond the range of A-Z (that is unicode characters).

Who made it

Like any good story, Scheme's starts in ancient times, back in the 1960s-70s, in the Golden Age of Computer Science, when philosophers, mathematicians, bohemians, entrepreneurs, militaries – people from universities, enterprises, garages, and war rooms, all were playing with our civilization's first revisions of interactive digital equipment.

Lisp was originally invented by John McCarthy in 1958 while on MIT. Inspired by Lambda calculus, he showed that one could build a Turing-complete language for algorithms using just a few instructions. While McCarthy hadn't realized a computer

[1] If you want a reference, go get it. There's An Introduction to Scheme and its Implementation on http://www.federated.com/~jim/schintro-v14/schintro_toc.html , also there's the Computer Science book Structure and Interpretation of Computers on http://mitpress.mit.edu/sicp/ which is based on Scheme, there's video to it out there as well, then there's The Scheme Programming Language on http://www.scheme.com/tspl3/ , as well as many other good books.

[2] Bigloo, Chez, Chicken, Gambit, Gauche, Ikarus, MIT, MzScheme, PLT, SISC, Tinyscheme, etc.

could actually execute this, Steve Russel read the paper, did so, and made a computer implementation of it. Russel also made the famous Spacewar! game. McCarthy is a Professor Emeritus of Computer Science at Stanford.

Gerald Jay Sussman and his former student Guy L. Steele lay the ground for Scheme in the late 1970s through while questioning current programming language implementations, writing a series of papers showing that programming languages can be implemented efficiently without constraining the programmers by arbitrary rules. Sussman also coauthored the well-known Computer Science book Structure and Interpretation of Computer Programs, and is one of two cofounders of The Free Software Foundation. Steele also coauthored the first three editions of The Java Language Specification.

Lisp pretty quickly became famous for its use in Artificial Intelligence research. Though, while it is good at describing algorithms to a computer, it does not perform anything intelligent in itself.

Scheme's standard, and its environments, have been made progress to over time. Scheme is one of the currently two major Lisp dialects, the other being Common Lisp.

Gambit started in 1988, as a Scheme for the Motorola 68K series of processors, used by very many computers of that day, including Macintosh desktops. It was given support to use the C programming language for intermediary step between Scheme and executable binary format in 1994, thus making it cross-platform. It has been released under both commercial and free licenses over the years, and was released open source in 2007.

Marc Feeley of Université de Montréal has led its development all from the beginning.

Get it up running

There's download links and nice instructions on Gambit's homepage www.iro.umontreal.ca/~gambit/ . It works on Mac OS X, Windows and Linux.

If you're on Windows, like with any console program, the interface won't be very fashionable, so perhaps you would consider for instance installing Debian Linux www.debian.org on a VirtualBox www.virtualbox.org and then run Gambit remotely on it via Putty www.putty.org .

How to interact with it

Typically you run the gsc program. Gsc stands for Gambit System Compiler. That is the full-fledged version. There's also the lightweight gsi, that can't generate C code and make binaries of it. The difference is 4MB to 3MB, so there's generally no reason not to use gsc. This is how it looks to start:

```
Gambit v4.3.0

>
```

And it gives you a prompt, you can type in things into it. The Lisp term for this is the Read-Eval-Print Loop. Read means you type something into the keyboard (namely, an expression, we'll get to that later), Eval means it performs what you typed, Print is that it displays the results, and Loop is that it keeps doing it.

Here's some basic guidance for how to navigate it:

If you ever want to quit, type `,q` and press enter, or press ctrl+c.

If you have ever generated an error (and you will, all of the time), the REPL will go into a special error mode. Here's how it looks:

```
> lets-see-what-happens-if-i-type-in-just-anything!
*** ERROR IN (console)@7.1 -- Unbound variable: lets-see-what-
happens-if-i-type-in-just-anything!
1>
```

In quoted REPL user interactions like this, the input from the user is bolded.

From this error mode, you generally want to do two things:

Get a ”backtrace”, that is the same thing as a stack dump. You do that with `,b` and enter.

Get back to the normal mode, you do that with ctrl+d.

There are Very sophisticated tools for error analysis available. For instance, there's advanced code and variable introspection. You may want to study that in the Gambit manual someday.

What's neat about the REPL is that it lets you execute quite any Scheme code right at the spot. Even if the code runs 5-500% slower in this interpreted mode than in compiled mode, usually you don't notice this speed difference at all. Rather, it's an incredible effectivity booster, because you can try ideas directly as they come to your mind.

This functionality does not exist in C, C++, Java, C#, Actionscript. It does exist in Python and Ruby, and it can easily be built in Javascript.

How to edit Files

Many Schemed developers appreciate Emacs with the Slime plugin.

There are other ways to edit Scheme code as well.

Eclipse with SchemeWay is one way.

VIM with syntax highlighting and its code formatting could hypothetically work as well.

I heard Lisp is full of parentheses, what about it?

In languages such as C, C++, Java, Pascal, Actionscript, Javascript, Ruby, Python etc., there are ample of different syntaxes for describing things such as procedure declarations, variables, code blocks, expressions, classes etc. . Let's look at two strips of code from Pascal and C (don't even read the code, just watch the general structure):[3]

```
Program NameOnList(Input,Output);
type
     ListType = ^NodeType;
     NodeType = record
                     First:String;
                     Rest :ListType
                end;
var
     List :ListType;
     Name :String;
function Member(Name:String; List:ListType):String;
     Begin
          if List = nil then
               Member := 'no'
          else if Name = List^.First then
               Member := 'yes'
          else
               Member := Member(Name,List^.Rest)
     End;
Begin
     Readln(Name);
     Writeln(Member(Name,List))
End.
```

```
#include <stdio.h>
typedef struct listCell * list;
struct listCell {
     int first;
     list rest;
};
bool guest(int x, list l) {
     if (l == NULL)
          return false;
```

[3] Code examples borrowed from teach-scheme.org/Talks/ts-proj.pdf .

```
        else if (x == (l -> first))
             return true;
        else
             return guest(x,l -> rest);
   }
   int main(int argc, char ** argv) {
        list l1,l2,l3 = NULL; int x;
        l1 = (list) malloc(sizeof(struct listCell));
        l2 = (list) malloc(sizeof(struct listCell));
        l2 -> first = 3; l2 -> rest = l3;
        l1 -> first = 2; l1 -> rest = l2;
        scanf("%d",&x);
        printf("%d\n",member(x,l1));
   }
```

While it should first be said the code examples above in no way map one-to-one to Scheme code, the following comparison still makes sense: All the bolded text in the examples above are replaced by one parenthesis, a comma, space, or by nothing at all in Scheme.

This means that we do get a lot of parentheses all over the place. And when we structure them with efforts of code indentation similar to what we are used to in C, C++, C#, Java, Javascript, Ruby, Python, etc., and we have the regular syntax highlighting we're used to from other languages as well, and, we have the feature of highlighting of the opposite parenthesis that we have in many other languages' editors as well, the structural clarity of Scheme code is always very good.

It should be said that Scheme code is less verbose, or to put it the other way around, more concise, than that of many other programming languages. Since the syntax is so compact, it can be learned by heart easily.

The standard Library

The R5RS standard library is found bundled with the language definition on its homepage, that should be schemers.org/Documents/Standards/R5RS/HTML/ . The Gambit-specific standard library is found in its manual on its homepage, that should be www.iro.umontreal.ca/~gambit/doc/gambit-c.html .

Additional core functions are typically described in Scheme Request for Implementation announcements, and available as regular libraries (if you don't know what a library is, read: as pluggable utilities; as accessory code). The spirit in which the SRFIs have been brought about is similar to that of the Request For Comments of the common Internet standards HTTP, FTP, etc.. SRFIs are listed on srfi.schemers.org .

The common Convention on Variable and Procedure Names

For instance, Java uses to have `identifierNamesWrittenLikeThis`, and the first letter is capital for class names.

In Scheme, `you-write-identifier-names-like-this`. More characters are allowed for identifier names than in most languages: You can use `* - + : . ' #` in them. As a general rule, you can have antyhing in a identifier name, except parentheses, semicolons (for), double quotes and quasiquotes, commas.

A procedure that modifies the contents of any of its parameters usually have names ending with `!` or `-set!`.

How to do Comments

Commenting out single lines is made using `//` in C, C++, Java, Javascript, PHP, Actionscript.

```
// my comment
```

In Scheme the equivalent is `;` .

```
; my comment
```

Scheme does not natively supports multiple-line comments, but it has that through an extension, and those are written `|| my comment ||` . (Compare C, C++, Java, Javascript, PHP, Actionscript `/* my comment */`).

Generally, you can also comment out expressions by putting `'` right before them, even though that's kind of a dirty hack. You'll understand why it works and why it's dirty when you study lists later.

Common Tasks in Scheme as compared to some other Languages

Call Procedures

Say that you have a procedure `foo`.

PHP, Javascript, C, C++, Pascal: `foo();`
Scheme, Common Lisp: `(foo)`

Say that you have a procedure `print`, and that you want to pass the value 5 to it.

PHP, Javascript, C, C++, Pascal: `foo(5);`
Scheme, Common Lisp: `(foo 5)`

Say that you have a procedure `print`, and that you want to pass the value "Hello world" to it.

PHP, Javascript, C, C++, Pascal: `foo("Hello world");`
Scheme, Common Lisp: `(foo "Hello world")`

Say that you have a procedure `foo`, and that you want to pass the values 5 and 6 and 7 to it.

PHP, Javascript, C, C++, Pascal: `foo(5,6,7);`
Scheme, Common Lisp: `(foo 5 6 7)`

Say that you have a procedure `foo`, and a variable `bar` that you want to pass to it.

PHP, Javascript, C, C++, Pascal: `foo(bar);`
Scheme, Common Lisp: `(foo bar)`

Declare Variables

C, C++, Pascal requires you to declare a type specifically for every variable. PHP, Javacript, Scheme, Common Lisp does not. Say that we want to declare the integer variable five containing the value 5, and the string variable hello containing the string value "Hello".

C, C++:	`int five = 5;`	`char[] hello = "Hello";`
Pascal:	`var five:Integer = 5;`	`var hello:String = "Hello";`
PHP:	`$five = 5;`	`$hello = "Hello";`
Javascript:	`var five = 5;`	`var hello = "Hello";`

Scheme differs a little bit from C, PHP, Javascript here, in that you can define variables in different ways.

First, there's the `define` procedure, using which you declare variables in the global namespace (that is, at the top level of the code, that is, not inside any declaration such as a procedure declaration. When the REPL is in its default mode, it's in the global namespace.).

```
(define five 5) (define hello "Hello")
```

`define`s in the global namespace are always executed in the order from first to last (usually that equals top to bottom). You'll learn how to define a function pretty soon, that will look like `(lambda (parameters) code)`. Before `code`, though, you can stick in `define`s:

```
(lambda (parameters)
  (define five 5)
  (define hello "Hello")
  etc.
  code)
```

However, these are not necessarily executed from first to last, their order of execution is undefined. You'll learn to write blocks of code pretty soon. You can define local variables at the beginning of any block of code. You do this using `let`, `let*` and `letrec`. `let` is the basic one.

```
(let ((five 5)
      (hello "Hello"))
     code)
```

You could really replace let here with `let*` or `letrec`, and it would do the same thing. What's special about the latter two is that while `let` does not allow a declaration in it to reference the own declaration or any other declaration in the `let`, `let*` allows each declaration to reference any earlier declaration, and, `letrec` allows each declaration to reference itself. Some Scheme environments have a `letrec*` that combines `let*` and `letrec`.

```
(let    ((five 5    ) (hello "Hello")) code)
```
Valid. (Example 1)

```
(let*   ((five 5    ) (hello "Hello")) code)
```
Valid. (Example 2)

```
(letrec ((five 5    ) (hello "Hello")) code)
```
Valid. (Example 3)

```
(let    ((five 5    ) (hello five   )) code)
```
Not valid. (Example 4)

```
(let*   ((five 5    ) (hello five   )) code)
```
Valid. (Example 5)

```
(letrec ((five 5    ) (hello five   )) code)
```
Not valid. (Example 6)

```
(let    ((five hello) (hello "Hello")) code)
```
Not valid. (Example 7)

```
(let*   ((five hello) (hello "Hello")) code)
```
Not valid. (Example 8)

```
(letrec ((five hello) (hello "Hello")) code)
```
Not valid. (Example 9)

```
(let    ((five 5    ) (hello hello  )) code)
```
Not valid. (Example 10)

```
(let*   ((five 5    ) (hello hello  )) code)
```
Not valid. (Example 11)

```
(letrec ((five 5    ) (hello hello  )) code)
```
Valid. (Example 12)

Examples 4 and 7-11 will throw an error on evaluation. In example 6, hello would rather than the value 5 get the value of `#!unbound`.

To `code` in example 5 above, `hello` will be 5. To `code` in example 12 above, `hello` will be `hello`. Perhaps that doesn't fill any practical purpose, it's just a self-referencing object, but, that's what it will do.

Write Blocks of Code

You can start a block of code quite anywhere. In that respect, Javascript, PHP and Actionscript are like Scheme.

Code blocks are started by `let`, `let*` or `letrec`, `lambda` (that we will learn about soon) or by `begin`.

Unlike `let`/`let*`/`letrec`, `begin` is not about declaring variables, only about declaring that you want to start a code block.

```
(begin
  code
  more code
  etc.)
```

Define Procedures

You define a procedure by (lambda (parameters) code), where code is its code, and parameters is its parameters declaration. The latter is a list of the parameters the procedure should take.

Scheme:	Compare Javascript and PHP:
`(lambda ( ) code)` takes no parameters.	`function( ) { code };`
`(lambda (a b ) code)` takes two parameters.	`function(a,b ) { code };`
`(lambda (a b c) code)` takes three parameters.	`function(a,b,c) { code };`

Also, a function can take be defined to take at least a specified number of parameters. This is defined through first defining the parameters that must be passed, then a separate dot, and then a variable that will get the rest of parameters passed, if any. If no parameters must be passed, then omit the dot, as well as the starting and ending parentheses of the parameters declaration. The rest of parameters will be passed as a list, you'll learn what a list is in the context of Scheme later.

`(lambda (a . rest) code)` will take at least one parameter.
`(lambda (a b . rest) code)` will take at least two parameters.
`(lambda all-parameters code)` will take zero or more parameters.

Now to add to it, there's a special convention called the DSSSL (you can just read that name and forget it right off, no problem), that's a convention on how to define parameters, in addition to the one described above. I don't know really why someone invented this. However, they don't interfere with each other, and the latter adds some really nice features. But before we jump onto that, let's look at a special syntax for how we define procedures in the global namespace.

So, by now, you know that you can do `(define five 5)` in the global namespace, and then `five` contains 5. And, you know that `(lambda () code)` is how you define a function that does code and doesn't take any parameters. Thus, not too complicated math brings you to the conclusion that

```
(define foo (lambda () code))
```

will define that function under the name “foo” in the global namespace. Now this is pretty neat, because that means if you run the procedure `foo`, and you know already you do that by `(foo)`, then you will execute the procedure.

Now just to get you to something real concrete, let's stick to the revelation that there's a `print` procedure that takes an arbitraty number of parameters. So `(print "Hi")` will print “Hi” on your screen. To add even more magic to it, there's a special `\n` sequence that you may have learned in other languages, that means newline. Thus, `(print "Hi\n")` will print “Hi”, and then move the cursor to the next line. If you do:

```
(define foo (lambda (bar) (print "Bar is " bar ".\n"))
```

Then running

```
(foo "open")
```

will print

```
Bar is open.
```

to your console. Also, there is a `write` procedure. What differs it from `print` is that it only takes one parameter for output, and, that it's made to output the parameter in a way that is technically specific, in contrast to print that's made to print pretty. We'll get to that later, when speaking about *serialization*. So enough for practical examples for now.

There's a *syntactic sugar* in Scheme that says that `(define X (lambda (Y) Z))` is equal do `(define (X Y) Z)`. That means that the define above is exactly the same thing as

```
(define (foo bar) (print "Bar is " bar ".\n"))
```

What you get out of this is that you may find this more comfortable and concise than the previous way.

So let's get back to the DSSSL stuff. It's about extending `parameters` of the `(lambda (parameters) code)` syntax like this:

First, you are free to add any obligatory parameters, just like we said previously.

Then, you may add optional parameters. You define these by `#!optional`. Then, for each parameter that you want to define, do this: If you want it to have a default value, write a parenthesis, the parameter's name, a space, its default value, and an ending parenthesis. If you won't want a default value for it, just write its name. Example:

```
(define (foo a #!optional b (c 5))
        (print "a is " a ", b is " b ", c is " c ".\n"))
```

Example uses:

```
> (foo 0)
a is 0, b is #f, c is 5.

> (foo 1 2)
a is 1, b is 2, c is 5.

> (foo 4 5 6)
a is 4, b is 5, c is 6.
```

The PHP equivalent of this example is:

```
function foo($a,$b,$c = 5) {
     echo "a is $a b is $b c is $c \n"; }
```

```
foo(0);
foo(1,2);
foo(4,5,6);
```

Then, you may add parameters that are optional and identified by a unique name, a little bit like you pass parameters to console programs: The Unix “ls” command, that's for listing files, optionally takes the `--hide` and `--width` parameter. So while `ls *` would display all files in a directory to you, `ls * --hide=apattern --width=15` would list you all files in the current directory, but with the files matching the pattern “apattern” filtered away, and no row of its console output will have more than 15 columns. You get the point.

Declaring parameters to work like this is done by first writing `#!key`, and then declaring the parameters like in `#!optional`. Here's an example:

```
(define (a b #!key c (d 7) (e 8))
        (print "b is " b ", c is " c ", d is " d ", e is " e
".\n"))
```

Here's example uses:

```
> (a 0)
b is 0, c is #f, d is 7, e is 8.

> (a 1 c: 2)
b is 1, c is 2, d is 7, e is 8.

> (a 1 e: 5)
b is 1, c is #f, d is 7, e is 5.

> (a 1 c: 2 d: 3 e: 4)
b is 1, c is 2, d is 3, e is 4.
```

Then, in order to get a behavior where an unlimited number of parameters are stuck into a variable, there's a `#!rest` as well. It is followed by the name of the variable that should get them. Here's an example, it uses `write` that we mentioned above:

```
(define (bar a #!rest b)
        (print "a is ")
        (write a)
        (print ", b is ")
        (write b)
        (print ".\n"))
```

Example uses:

```
> (bar 1)
a is 1, b is ().

> (bar 2 3)
a is 2, b is (3).
```

```
> (bar 4 5 6)
a is 4, b is (5 6).
```

So this is what DSSSL is about: first parameters that must always be there, then `#!key` for named optional parameters, then `#!optional` for unnamed optional parameters, then `#!rest` for the rest.

They can be combined, for instance like this:

```
(define (x #!optional (a 4) #!key (b 5) c)
        (print "a is ") (write a) (print ", b is ") (write b)
        (print ", c is ") (write c) (print ".\n"))

(define (y #!key (i 5) #!rest j)
        (print "i is ") (write i) (print ", j is ") (write j)
        (print ".\n"))
```

Example uses:

```
> (x)
a is 4, b is 5, c is #f.

> (x 1)
a is 1, b is 5, c is #f.

> (x 2 b: 3)
a is 2, b is 3, c is #f.

> (x 4 b: 5 c: 6)
a is 4, b is 5, c is 6.

> (x 7 c: 8)
a is 7, b is 5, c is 8.

> (y 9)
i is 5, j is (9).

> (y 10 i: 11)
i is 5, j is (10 i: 11).

> (y i: 12 13)
i is 12, j is (13).
```

Your current understanding of what power there is to parameter passing will mature first when you know about lists. Though this was OK for a primordial example.

Variable Scope

At any location in your code, you must know which of your variable declarations it is that the computer will understand you to have addressed, when you pass their names.

Different programming languages come with different solutions to this problem.

One solution is that all variables' names must be unique in every scope. This is true for for instance C, C++ and Pascal.

```
int foo;        Valid.
{
    int bar;    Valid.
    int foo;    Not valid, there's already a variable foo defined.
}
```

This is how Scheme does it:

The general rule is that all variables declared until now in the current code block, and previously in the code in any code block “above” the current code block in the parse tree, are accessible.

```
(define one 1)

; one is accessible from here

(define (a-procedure)
  ; one is accessible from here as well
  (let ((two 2))
    ; one is accessible from here as well.
    ; two is accessible from here.
    )

  ; though two is not accessible from here.

  (let* ((three 3))
    ; two is not accessible from here neither, though,
    ; one is accessible from here.
    ; three is accessible from here.
    ))

; one is accessible from here, but not two nor three.
```

In case there's an overlap of names, the declaration that is most close is used.

```
(define var 1)
; var is 1
(let ((foo 2))
  ; var is 1
  (define (bar)
```

```
(letrec ((var 3))
  ; var is 3
  )
; var is 1
)
```

Of course, if you would mutate the contents of any variable (that's using `set!`, more on that later), the variable that's changed is looked up according to the same scheme that they were read above.

Toss around Procedures just like you do with Variables (First-Class Procedures)

Procedures being first-class means you can use them as variables.

Procedures are first-class in Scheme, including the operators `+ - / *`.

Procedures (though not operators) are first-class in C#, D, Javascript, PHP as well. C, C++ and Pascal could be forced into a functionally equivalent behaviour, but that's through using aspects of the language seldom used, by good reasons[4]. Java has nothing to come up with similar to this.

Procedures being first-class implies a lot. Make sure that you understand this put in the context of closures and continuations. However, those are both lessons to come.

Here's how to store them in variables:

```
> (define plus +)
> (define minus -)
> (define go-with-plus #t)
> (define method (if go-with-plus plus minus))
> (method 1 2)
3
> ((if go-with-plus plus minus) 4 5)
9
```

Here's how to put them in a list, and apply a procedure to each one of them:

```
> (define (test a b)
    (for-each
     (lambda (p)
       (print p " " a " " b " is " (p a b) ".\n"))
     (list + - * / < >)))
> (test 2 3)
#<procedure #11 +> 2 3 is 5.
#<procedure #12 -> 2 3 is -1.
#<procedure #13 *> 2 3 is 6.
```

[4] Again that would be through using pointered types. Usually that imples a mess of code, and the compiler won't give you warnings like you are used to.

```
#<procedure #14 /> 2 3 is 2/3.
#<procedure #15 <> 2 3 is #t.
#<procedure #2 >> 2 3 is #f.
> (test 10 20)
#<procedure #11 +> 10 20 is 30.
#<procedure #12 -> 10 20 is -10.
#<procedure #13 *> 10 20 is 200.
#<procedure #14 /> 10 20 is 1/2.
#<procedure #15 <> 10 20 is #t.
#<procedure #2 >> 10 20 is #f.
```

Here's an example on tossing them around:

```
> (define combiner +)
> (define (foo method1 method2 param1 param2)
    (combiner (method1 param1 param2) (method2 param1 param2)))
> (foo * + 4 5)
29
> (foo + + 2 1)
6
> (foo + + 10 20)
60
> (foo (lambda (a b) (+ a 3)) - 10 20)
3
```

Did you follow that? Let's take a look at the first example, `(foo * + 4 5)`. `foo` is executed so that `method1` is `*`, `method2` is `+`, `param1` is 4 and `param2` is 5. `method1` is executed with `param1` and `param2` as parameters, that means `(* 4 5)` is evaluated, and we have 20. Then, `method2` is evaluated with `param1` and `param2` as parameters, that means `(+ 4 5)` is evaluated, that gives 9. Finally, `combiner` is evaluated with 20 and 9 as parameters, that is `(+ 20 9)` is evaluated, and `foo` returns 29.

The Variable Types

Scheme has booleans, numbers, strings, characters, symbols, lists, pairs, procedures, continuations, vectors, ports. Additionally, Gambit adds hashtables, structures, u8vectors, uninterned symbols, a special “foreign type”, and some other types as well.

Each of them have typechecker procedures for them, that is, there is a `string?` procedure as well as a `table?` procedure, etc. .

```
> (string? "Hi")
#t

> (string? 2)
#f

> (procedure? print)
#t
```

Boolean

True is `#t`, false is `#f`.

Compare in Java, Pascal, C, C++, C#, D, PHP, Javascript, Pascal it's true / false.

In C, C++, PHP and Javascript, you can treat integer values as booleans. The rule is that zero is false, and everything else is true. For instance, you can do `int i = 1; bool b = i;` in C and C++. And, in all of them you can do `if (1) { code }`, and code will be executed, because the expression evaluated to true.

Scheme both does this and does not do this. This is what Scheme does:

`#f` is a unique value. It is the only value that evaluates to false. All other values are true.

Thus, `(print (if 5 "True!\n" "False\n"))` prints “True!”, and `(print (if #f "True!\n" "False\n"))` prints “False!”.

The main procedures for boolean operations are `not`, `and` and `or`.

Number

Scheme's number system is very sophisticated, it supports integer, decimal, rational and complex values in all sizes.

This is far beyond what any of C, C++, Java, Javascript, PHP have native support for.

Examples:

```
> 5                    Integer
5
> 2.5                  Decimal
2.5
> 10/11                Rational
10/11
> 3+4i                 Complex
3+4i
> (+ 3+4i 2)
5+4i
> (+ 10/11 3)
43/11
> (- 2.5 0.5)
2.
> (inexact->exact 2.)
2
```

```
> (exact->inexact 2)
2.
> (exact->inexact 43/11)
3.909090909090909
```

String

In Scheme a String is "A character sequence just like this", just like in Java, Javascript, PHP, C, C++. Please note that while Strings can be started and ended by apostrophe in PHP and Javascript, this is not the case in Scheme.

Character

`#\xHH` or `#\UHHHHHHHH` where H is the hexadecimal value for the character. There is also a set of unique characters such as `#\space` and `#\tab`. The procedure `integer->char` converts a unicode character into its character representation. `(make-string 1 #\!)` makes the string `"!"`.

The character type's equivalent in C, C++ and Pascal is char . It has no equivalent in Javascript or PHP.

Symbol

This is a datatype pretty unique to Lisp.

A symbol is a special type of string. What's unique about them technically is that while a string gets into the system, memory is allocated for it, when you compare two strings, they are compared character by character, and when a strings is no longer used, it is garbage collected.

When a symbol gets into the system, a lookup is made in a global symbol table, in which each symbol has a unique object reference. If the symbol doesn't exist, a slot is allocated for it. When symbols are compared, their object reference identifiers are compared only. Symbols are never garbage collected, so don't use them for values you have more than a well-defined set of.

Though, the technical difference is solely a consequence of the concept of symbols. Symbols have a key role in code generation, and usually in information processing. You'll learn more about this soon.

List and Pair

Lists are at the heart of Lisp.

Effectively, lists are linked lists of pairs, and pairs is a structure datatype with two slots in it, the head (the “car”) and the tail (the “cdr”). The names car and cdr are there because of antique reasons.

Using these linked lists, you can do linear data structures, cyclically formed data structures, tree data structures, and mixes of these.

What is powerful about Lisp lists is that this abstraction, that is, this datatype, is so simple!

That might sound like a fancy argument, but that's the way it is. Neither C, C++, Pascal, PHP, Python, Ruby on Rails, Actionscript, Javascript, Java, you name it have it.

What is powerful about it is not this simple datatype in itself. I suppose it's about as useful in itself as one hydrogen atom is. The funny thing is building molecules using it, and analyzing or creating the structures at racing speed using any of the very many rich libraries that are available, that are built on it.

A Java equivalent of a Pair would be

```
public class Pair {
     Object car, cdr;
}
```

We will look closer at how lists are used to do rapid development and in information processing.

Procedure

Procedures are “first-class”, that means they are counted as a data type, and that you can pass them around as variables. This has already been covered with a separate section.

Continuation

We'll get to what these are about later.

Vector

These are arrays with an integer index value for lookup. Comparable to Java, C, C++, Javascript, Actionscript arrays, though they can contain any variable type.

Ports:

Scheme comes with an I/O system that's based on “ports”. They are analogue to file handles in C, Pascal, PHP, and with File objects in Java.

The main distinction among ports are the binary and those for characters, the latter do character encoding conversion while the further don't. They're all in the Gambit manual.

Hashtable

While not mathematically kosher enough to do it all the way into the R5RS spec, Gambit provides hashtables.

You create them with `(make-table)`, set their content with `(table-set! table key value)`, get their content with `(table-ref table key)`, and there's some special rules for tweaking their contents' relation with the garbage collector.

Its equivalent in Javascript would be objects, where you can always do `object[key] = value;` , for instance `var v = {}; v["hi"] = "First string";` . The same thing is in PHP, `$object[$key] = $value;` .

Structure

C and Pascal provide structures. They would look something like

```
struct Book {
     title:String;
     isbn:String;
};
```

Or as in Java

```
class Book {
     String title;
     String isbn;
}

Book b = new Book();
b.title = "An Introduction to Scheme and its
Implementation";
```

Or as in Javascript

```
function book() { return {title: "", isbn: ""}; }
var b = book();
v.title = "An Introduction to Scheme and its
Implementation";
```

Gambit provides `define-type`:

```
(define-type book title isbn)
(define b (make-book "" ""))
(book-title-set! b
                 "An Introduction to Scheme and its
Implementation")
(print (book-title b))
```

When using structures for more than long time, check out the `id:` parameter of `define-type`.

U8vector

This is a special array type, where the elements are unsigned 8-bit integers. The C and Pascal analogue to this would be array of byte. There's a separate chapter on handling binary data later that addresses u8vectors.

Uninterned Symbols

Uninterned symbols are symbols that garbage collect. Generated using `(gensym)` or `(make-uninterned-symbol)`.

Foreign Type

Gambit supports integration with C/C++ libraries. Data structures that you import from the C world will appear in the Scheme environment as foreign function object instances. You generally only inspect them using procedures tailored for the respective foreign structure type. More about that in the Gambit manual.

Compare Values

Scheme comes with `=`, `eq?`, `eqv?` and `equal?`.

`=` compares numbers only, and throws errors if it got other parameters.

`eq?` and `eqv?` compares value for numbers, and compares object reference for objects.

`equal?` compares the full contents of the compared values.

```
> (= 1 2)
#f
> (= 1 1)
#t
> (eq? 'a 'b)
```

```
#f
> (eq? 'c 'c)
#t
> (eq? 1 1)
#t
> (eq? #t #t)
#t
> (eq? "a" "a")
#f
> (define variable "a")
> (eq? variable variable)
#t
> (equal? "a" "a")
#t
> (equal? #t #t)
#t
```

The reason `(eq? "a" "a")` gives `#f` is because they have different object references.

Compare Java:

`1 == 1`	True, compared by value.
`new Integer(1) == new Integer(1)`	False, compared by object reference.
`"a" == "a"`	False, compared by object reference.
`"a".equals("a")`	True
`new Integer(1).equals(new Integer(1))`	True
`String variable = "a";`	
`variable == variable`	True, compared by object reference.

There's a little bit more details to it than this, see the R5RS specs.

Typecasting (that is, convert Variable Types)

First, let's look about how we relate to variable types when declaring variables:

> In C, C++, Pascal, you define a variable type with the definition of every variable, be it a global, local, or parameter variable. The variables cannot contain variables of other types (generally[5]).

C:
```
int v;
v = 5;
```
Valid

[5] The only exception is if you create pointered types. This is not usual to do in Pascal and C++. In C it is, but, its usage will generally disable the compiler from being able to warn you about invalid type use, so it should be used with care there as well.

```
        v = "Hi";       Not valid
Pascal: var v:Integer;
        v := 5;         Valid
        v := "Hi";      Not valid
```

In Java, you define a variable type with the definition of every variable, but if it's a class instance, you can actually typecast them freely between each other, though the typecast needs to be defined manually.

```
{
     Integer   a = new Integer(5);
     ArrayList b = (ArrayList) a;    Valid
     ArrayList c = a;                Not valid
     int d = 5;
     d = (int) b;           Not valid (because d is not a class
instance)
     c = (ArrayList) d;     Not valid (")
}
```

In PHP and Javascript, there are no types on variable declarations.

```
PHP:        $whatever_variable = 123;
Javascript: var whatever_variable = 234;
```

In Scheme, there are neither any types on variable declarations.

```
(define whatever-variable 345)
(let ((whatever-variable 456)) (print whatever-variable))
```

Then, let's look at how strict we are about variable types in evaluations:

C and C++ interconvert number types automatically between each other. Boolean values are counted as numbers. String ↔ number is not automatic, neither number ↔ object or string ↔ object.

Pascal and Javascript are like C and C++ except that number → string conversion is made automatically.

Java is like Pascal and Javascript, but booleans are variable type separate from numbers.

Scheme is like Java, but it does not do number ↔ string conversion automatically.

`(+ 2 "three")`	Not valid
`(string-append "two" 3)`	Not valid
`(string-append "two" (number->string 3))`	Valid
`(+ 2 (string->number "3"))`	Valid

So Scheme's offer is that while the way you d data is very flexible and unrestricted (in the sense that you don't ever declare variable types), internally, it is very strict about variable types.

Conditional Flows of Execution and Loops

C, C++, C#, Java, Javascript, PHP, Python, Actionscript give us quite the same assortment of syntactic constructs using which to put up conditions in your code to steer its flow. These are:

```
if (this evaluates to true)
     run this;
else
     run this;

while (this evaluates to true)
      run this;

do
     run this;
until (this evaluates to true);

for (starters, do this, then; while this evaluates to true; do
this)
     and then this, and then continue at step two;

goto here;
[ whatever ]
here:

switch (this value) {
     case like-this:
          do-this;
     case like-this:
          do-this;
     etc.
     default:
         do-this;
}
```

.. and `break;` and `continue;`.

Of these, Scheme supports `if`, and `do`, in two variants.

The way Scheme does control flow is very separate from the examples above.

Out of Scheme's vantage point, the constructs above seem like simple playthings: they are utilities that fit a quite narrow usage spectra.

What Scheme does is to translate these problems into a question of Scheme-style recursion.

Before looking into recursion, let's look at the conditional operations we have, and use to use:

First, we have `if`. It works like this:

```
(if this do-this otherwise-this)
```

Examples:

```
> (if #t "True" "False")
"True"
> (if (not #t) "True" "False")
"False"
> (if (zero? 0) "True" "False")
"True"
> (if #t
    (print "True\n")
    (print "False\n"))
True
> (if (number? "Hi")
    (begin
      (print "Tru")
      (print "e\n"))
    (begin
      (print "Fal")
      (print "se\n")))
False
```

Second, we have `cond`:

```
(cond (if-this-is-true do-this-then-break)
      (if-this-is-true do-this-then-break)
      etc.
      (else do-this))
```

Examples:

```
(define v 4)
(cond ((= v 3) (display "V is 3.\n"))
```

```
        ((= v 4) (display "V is 4.\n"))
        (else (display "V is neither 3 nor 4.\n")))
```

Also there's `do`, but I never saw it in use.

Also note how we can use the boolean operator procedures `and` and `or` to control flow.

`and` works like this: it evaluates each of its parameters. If during the evaluation process it gets one false return value, it returns false. Otherwise, the value of the last parameter will be returned.

`or` works like this: it evaluates each of its parameters. Upon the first evaluation that does not return false, it returns the return value. If no non-false return value is had during all evaluations, it returns false.

Examples:

```
> (or #f 1 2)
1
> (or 1 #f 2)
1
> (or 1 2 3)
1
> (or #f 2 3)
2
> (and #f 1 2)
#f
> (and 1 2 #f)
#f
> (and 1 2 3)
3
```

Check our what this does to procedure evaluation:

```
> (define (m a)
    (print "m was invoked for " a ".\n")
    (if (> a 3) "a is more than 3!" #f))
> (or (m 2) (m 3))
m was invoked for 2.
m was invoked for 3.
#f
> (or (m 2) (m 3) (m 4) (m 5))
m was invoked for 2.
m was invoked for 3.
m was invoked for 4.
"a is more than 3!"
> (or (m 5) (m 4) (m 3) (m 2))
m was invoked for 5.
"a is more than 3!"
> (and (m 2) (m 3) (m 4) (m 5))
```

```
m was invoked for 2.
#f
> (and (m 5) (m 4) (m 3) (m 2))
m was invoked for 5.
m was invoked for 4.
m was invoked for 3.
#f
> (and (m 5) (m 4))
m was invoked for 5.
m was invoked for 4.
"a is more than 3!"
```

We should see that the way Scheme does this is very powerful, so let's get to recursion.

Recursion (and Tail Call Optimization)

C, C++, C#, Java, Javascript, D, Actionscript, PHP generally don't support tail call optimizations. Tail call optimizations can be explained like this:

```
function b() {
     do_some_things();
     return result_of_some_other_things();
     note_this_will_never_be_executed();
}

function a() {
     b();
     do_further_things();
}
```

When we get to `return result_of_some_other_things()` in `b()` during a call from `a()`, it is completely obvious that there will be no further execution within `b()` before the return to `a()`. Generally, when a function call is made, a frame is allocated in the program's stack.

But in this special case – that is, a call is made, but returning to the current frame would be meaningless because everything it would do would be to return to the frame under it – we can actually skip the stack allocation when calling, and instead replace the current frame with the frame we thought about allocating. That would be equal to making what you may recognize as a `goto` or a `longjmp` instead of a `call`.

So what do we get out of this? Well, what happens is that when we know our underlying environment has this particular optimization or should we say feature built in, that is, that making tail calls are cheap, the way we program can be adapted in alignment to this.

That is, in the general programming languages, recursion is a little bit expensive. In Scheme, tail recursion is not. Scheme compilers have terrific optimizations related to benefiting from tail call optimizations. We take it as far as building all of our loops on it.

This is the general structure of an infinite loop:

```
(let loop ()
  code
  (loop))
```

If you want to put a condition on it, make it to the invocation of `(loop)` .

```
(let loop ((i 5))
  (print "i is " i ".\n")
  (if (> i 0) (loop (- i 1))))
```

Output:

```
i is 5.
i is 4.
i is 3.
i is 2.
i is 1.
i is 0.
```

What did this code do? It allocates the local variable i, sets it to 5, prints out “i is 5.”. It checks if i is more than zero, which it is. Therefore, it substracts one from i, gets 4, and invokes loop with this as parameter, so now i is 4. It prints out “i is 4.”, etc.

With this way of working, we have won a great deal of flexibility. You can have loops that jump between each other, that have very complex rules for how to loop, etc.

Make Evaluations

C, C++, C#, Java, Javascript, Actionscript, PHP uses infix notation for evaluations. It looks about like this:

```
a = b * c + d - e / f;
```

Evaluations in Scheme are made with prefix notation. That is, all evaluations are made through that you take a pair of parentheses, then put the procedure you want to call first, and then the parameters then. Thus,

```
a = b * c * (d + e + (f - (g / h)));
```

equals

```
(set! a (* b c (+ d e (- f (/ g h)))))
```

The pros about the Scheme way here is that there's no need to keep track of operator

precedence in evaluations, and that there's no need to write operators multiple times when applying them in a sequence (it's `a + b + c + d` vs. `(+ a b c d)`).

Please note that `+`, `-`, `*`, `/` etc. are just regular procedures like `print` and `number->string`.

Note that everything evaluates, everything gives a return value. There is a silent `#!void` return value returned by certain functions, for instance `print`.

The Scheme equivalent of the assignment operator `=` in C, C++, etc. is `set!`.

Javascript:
```
var v = 3;
v = 5;
v = v + 2;
```

Scheme:
```
(define v 3)
(set! v 5)
(set! v (+ v 2))
```

..or:
```
(let ((v 3))
     (set! v 5)
     (set! v (+ v 2)))
```

Functional programming can be summarized in: Don't mutate, and don't have a state. Mutate is to change the contents of a variable, and that's exactly what `set!` does. Even though it might be beneficial in some cases, in general, rather have programming code take input data, and generate new structures for output data, all of the time. More about this in a dedicated section later.

Handle Structures of Data of all Kinds (linear, circular or tree-structured. That is LISP Lists.)

So until now we have said that lists are linked lists of the pair data type. Now let's look a bit closer on lists.

This is how you create a pair: `(cons 0 1)`, 0 will become the car and 1 the cdr.

```
> (cons 0 1)
(0 . 1)
```

Parenthesis, value, dot, value, parenthesis is how pairs are printed out. Lists can be created in multiple ways. For starters, let's look at the list procedure.

```
> (list 0 1 2 3)
(0 1 2 3)
```

This structure is constructed by three pairs. Here is how we would construct the same list, if we made it pair by pair.

```
> (cons 0 (cons 1 (cons 2 '())))
(0 1 2)
```

We get the car of a list with the `car` procedure, and the cdr of a list with the `cdr` procedure.

```
> (car (list 1 2 3))
1
> (cdr (list 1 2 3))
(2 3)
```

There is a set of procedures that combine a sequence of `car`s and `cdr`s:

```
> (car (list 1 2 3 4))
1
> (cdr (list 1 2 3 4))
(2 3 4)
> (car (cdr (list 1 2 3 4)))
2
> (cadr (list 1 2 3 4))
2
> (caddr (list 1 2 3 4))
3
> (cadddr (list 1 2 3 4))
4
> (cddr (list 1 2 3 4))
(3 4)
> (cdddr (list 1 2 3 4))
(4)
> (cddddr (list 1 2 3 4))
()
```

There is also `list-ref` to pick an entry from a list

```
> (list-ref (list 'first 'second 'third) 1)
second
> (list-ref (list 'first 'second 'third) 2)
third
```

There's also `length`, to count list length

```
> (length (list 7 8 9))
3
```

In addition to these, there are ample of list handling functions to do things such as searching, sorting, filtering, splicing, etcetra in the R5RS specification, in SRFI 1 and in SRFI 95.

We have still merely touched the surface on what we can do with lists.

Now let's get deeper on how to define lists. In addition to list, there's quote and

quasiquote.

List is a procedure that makes a list of the parameters you pass it.

```
> (list (+ 1 2) (+ 3 4) (string->number "8"))
(3 7 8)
```

quote is about conserving everything you pass it as a list.

```
> '(list (+ 1 2) (+ 3 4) (string->number "8"))
(list (+ 1 2) (+ 3 4) (string->number "8"))
```

What was that? Well, it's a list consisting of the symbol `'list`, then `'(+ 1 2)`, then `'(+ 3 4)`, then `'(string->number "8")`. Say that we would take a look at the list `'(+ 3 4)` in particular, then its first entry is `'+`, then `3`, then `4`. That's it. So quote is about conserving the list structure you pass it.

```
> (define a '(list (+ 1 2) (+ 3 4) (string->number "8")))
> (list-ref a 0)
list
> (list-ref a 1)
(+ 1 2)
> (list-ref a 2)
(+ 3 4)
> (define b (list-ref a 2))
> (list-ref b 0)
+
> (list-ref b 1)
3
> (list-ref b 2)
4
```

We'll get more into the particular use of lists to describe Scheme code in the eval and macro sections that are yet to come.

Quasiquote is quote but with a modification: `,` and `,@` , is about stopping the conservation to insert something from external. `,@` is a `,` , but with the content merged into the list structure.

```
> (define a (list 1 2 3))
> '(this is a fine list)
(this is a fine list)
> '(this is ,a fine list)
(this is ,a fine list)
> `(this is a fine list)
(this is a fine list)
> `(this is ,a fine list)
(this is (1 2 3) fine list)
> `(this is ,@a fine list)
```

```
(this is 1 2 3 fine list)
```

Now let's look at some common R5RS list procedures:

`for-each` takes two parameters, one procedure that should take a value as a parameter, and then a list. `for-each` invokes the procedure for each of the entries in the list. `map` is a `for-each` that collects the procedure's return value into a list.

```
> (for-each print '("hi " "ho\n"))
hi ho

> (for-each
    (lambda (v) (print "Got value: ") (write v) (print "\n"))
    '("value one" "second value" "third value"))
Got value: "value one"
Got value: "second value"
Got value: "third value"

> (map string->number '("1" "7" "12"))
(1 7 12)
```

`apply` takes one procedure and one list as parameters. What it does is to pass the contents of the list as parameters to the procedure.

```
> (apply + '(5 10 20))
35

> (apply string-append '("a " "list" " of strings."))
"a list of strings."
```

Other procedures you may want to study are `append`, `assoc` / `assq` / `assv`, `member` / `memq` / `memv`.

Scheme Code is Lists (that is S-Expressions, or S-Exps for short)

Scheme internally represents all of your code as lists. That is, when you enter the expression

```
(+ 2 (* 3 4))
```

into it, it will relate to this internally as a list of three elements, where the first element is `'+`, the second element is 2, and the third element is `'(* 3 4)`.

Lists that contain code are called s-expressions.

All Scheme code is s-expressions.

All Scheme code is represented internally in any Scheme environment as s-expressions during any code processing work. Code processing work is about performing interpretation and compilation. They are preceded by a third step, macro expansion. More about that in the macro section.

The term we use to address the totality of code in a Scheme environment is the parse tree. So the parse tree is the total amount of s-expressions in your Scheme environment.

And like s-expressions is a hierarchical data structure, the parse tree is as well, so you could really adderss any part of the parse tree as “here, the parse tree is”, or “what's in the parse tree right here right now”. Usually the word “code” does more than well for Scheme code, though the “parse tree” term might be beneficial to describe how Scheme relates to code internally, and how macros work, which we get to in the macros section.

Evaluate S-Expressions whenever (that is eval)

One more or less known aspect of Javascript is the ability to run Javascript code with a string for source of programming code:

```
eval("Window.alert(\"Test box\");");
```

Scheme has `eval`, and you will see that it's at the heart of the language.

`eval` takes an s-expression for parameter. The list should obviously be valid Scheme code.

```
> (eval '(display "hi\n"))
hi
```

This means that you can construct programming code during execution time and run it. And you can download it from the Internet, or load it from files, or from a database, and run it. Whenever.

Code executed by `eval` does not have access to your global environment. Thus, if you want it to get input data, make it return a procedure that you can pass the input data.

```
> (define fu (eval '(lambda (input) (+ input 1))))
> (fu 5)
6
> (fu 10)
11
```

There's an example combining deserialization and `eval` in the serialization section later.

Handle XML

There is a 1:1 mapping of the text data format XML with Scheme's list structures called SXML. There's a widespread library to deal with this called SSAX-SXML.

So let's just zoom out conceptually, briefly. XML is perhaps the most popular data format for structured data there is. LISP is perhaps the most powerful programming language there is when it comes to list processing. If you can make LISP eat XML, that means that you have the most powerful XML processing there is. And that is exactly what SXML is about.

Here's some XML and its SXML equivalents:

```
<br />                                  (br)

<p>Hi ho</p>                            (p "Hi ho")

<img src="hi.jpeg" />                   (img (@ (src "hi.jpeg")))

<table width="50%" height="100">        (table (@ (width "50%")
(height "100"))
     <tr><td>Hi</td></tr>                 (tr (td "Hi")))
</table>

<html><body>Hi</body><html>             (html (body "Hi"))

<!-- Comment -->                        (*COMMENT* " Comment ")
```

And that's pretty much the core of the standard. Well, there's namespace support also. You find out more about that and other SXML features on SSAX-SXML's homepage.

Here's how to load:

```
> (define a (xml-string->sxml "<data><book id=\"1\"><title>My
book</title><isbn>123</isbn></book><book id=\"2\"><title>Another
book</title><isbn>234</isbn></book></data>"))

> (define b (xml-string->sxml "<html><body><p>Text</p><p>More
text</p><img src=\"flower.jpeg\" /></body></html>"))

> a
(*TOP* (data (book (@ (id "1")) (title "My book") (isbn "123"))
             (book (@ (id "2")) (title "Another book") (isbn
"234"))))

> b
(*TOP* (html (body (p "Text")
                   (p "More text")
                   (img (@ (src "flower.jpeg"))))))
```

It's exciting to see how well this integrates with Scheme, in a way that produces extremely concise code. Here's how we do xpath:

```
> ((make-xpath "/data/book/title/text()") a)
("My book" "Another book")
> ((make-xpath "/data/book/isbn/text()") a)
("123" "234")
> ((make-xpath "//p/text()") b)
("Text" "More text")
```

Let's generate some XHTML statically:

```
> (define c '(*TOP* (html (body (p "text") (p "more text") (p
"even more text")))))
> c
(*TOP* (html (body (p "text") (p "more text") (p "even more
text"))))
> (define xml (sxml->xml c #t))
> (print xml "\n")
<html>
  <body>
    <p>text</p>
    <p>more text</p>
    <p>even more text</p>
  </body>
</html>
```

Let's generate some dynamically as well:

```
> (define c
  `(*TOP*
    (html
      (body
      "Here's some books my children read recently: "
      (ul
        ,@(map
          (lambda (t)
            `(li
              (@ (style "color: blue;"))
              ,(string-append "The book \"" t "\".")))
          '("Peter Pan" "Le Petit Prince" "Pippi
Longstocking")))
      (hr)
      "More about it later!"))))
> c
(*TOP* (html (body "Here's some books my children read recently:
"
                   (ul (li (@ (style "color: blue;"))
                           "The book \"Peter Pan\".")
                       (li (@ (style "color: blue;"))
                           "The book \"Le Petit Prince\".")
                       (li (@ (style "color: blue;"))
```

```
                                    "The book \"Pippi Longstocking\"."))
                          (hr)
                          "More about it later!")))
> (define xml (sxml->xml c #t))
> (print xml "\n")
<html>
  <body>Here's some books my children read recently: <ul>
    <li style="color: blue;">The book "Peter Pan".</li>
    <li style="color: blue;">The book "Le Petit Prince".</li>
    <li style="color: blue;">The book "Pippi Longstocking".</li>
  </ul><hr/>More about it later!</body>
</html>
```

Now you have the tools to make XML transformations kicking XSLT to the moon.

Further reading is Oleg Kiselyov's SXML page on http://okmij.org/ftp/Scheme/xml.html , and Peter Bex' “Authoring dynamic websites with SXML” available at http://sjamaan.ath.cx/docs/scheme/sxslt.pdf .

Serialize and de-serialize Objects

Scheme features serialization of s-expressions to text and back.

Serialization is through write and deserialization is through read.

```
> (define variable `(1 "test string" 1.0 11/25 (recursion
(recursion (recursion a-symbol another-symbol)))))
> variable
(1
 "test string"
 1.
 11/25
 (recursion (recursion (recursion a-symbol another-symbol))))
> (define as-string (call-with-output-string '() (lambda (p)
(write variable p))))
> (print as-string "\n")
(1 "test string" 1. 11/25 (recursion (recursion (recursion a-
symbol another-symbol))))
> (define read-variable (call-with-input-string as-string read))
> read-variable
(1
 "test string"
 1.
 11/25
 (recursion (recursion (recursion a-symbol another-symbol))))
```

The only data structure that cannot be deserialized this way is structure values from `define-type`.

Here's an example of how to combine deserialization and `eval`:

```
> (define (read-expression-from-console)
    (print "Please enter an expression and press enter.\n")
    (read (current-input-port)))
> (define expression (read-expression-from-console))
Please enter an expression and press enter.
(+ 2 3 4)
> expression
(+ 2 3 4)
> (list-ref expression 0)
+
> (begin (print "The output from the expression is: ") (display
(eval expression)) (print ".\n"))
The output from the expression is: 9.
```

Gambit features serialization to binary format, that can serialize closures and continuations, including their state. More about that in the section on procedure, closure, continuation serialization.

Write Blocks of Code with a State (that is Closures)

Until now, all variables we have been working with have been in the global namespace (that is, global variables), and variables that we use within the scope of a block of code (that is, local variables).

```
(define global-variable "a value")
(let ((local-variable "a value"))
     code)
```

Now let's take our variable use to the next dimension.

```
> (define (container value)
    (lambda ()
      (string-append "This container contains " value ".")))
> (define apple (container "an apple"))
> (define pie (container "a pie"))
> (apple)
"This container contains an apple."
> (apple)
"This container contains an apple."
> (pie)
"This container contains a pie."
> (pie)
"This container contains a pie."
> (apple)
"This container contains an apple."
```

There's no closures in C, C++, Java. There are in Javascript, Actionscript, Ruby, Python, D. In Javascript this code would look like:

```
function container(value) {
     return function() {
          return "This container contains " + value + ".";
     };
}
var apple = container("an apple");
var pie = container("a pie");
apple();
apple();
pie();
pie();
apple();
```

What did we just do here? We trapped variables into code. You need to spend a lot of time experimenting with this feature, because it is very powerful. Among other things, you can implement coroutines and object orientation kind of features using closures.

Counter example:

```
> (define (counter)
    (let ((value 0))
      (lambda ()
        (set! value (+ value 1))
        value)))
> (define first-counter (counter))
> (define second-counter (counter))
> (first-counter)
1
> (first-counter)
2
> (first-counter)
3
> (second-counter)
1
> (second-counter)
2
> (first-counter)
4
> (second-counter)
3
> (second-counter)
4
```

Simple container example:

```
> (define (container)
    (let ((value "None yet"))
      (cons
        (lambda (v) (set! value v))
        (lambda () value))))
> (define first (container))
> (define first-set! (car first))
```

```
> (define first-get (cdr first))
> (define second (container))
> (define second-set! (car second))
> (define second-get (cdr second))
> (first-get)
"None yet"
> (first-set! "Hi!")
> (first-get)
"Hi!"
> (second-set! (first-get))
> (first-set! 123)
> (first-get)
123
> (second-get)
"Hi!"
```

Simple coroutine-style example:

```
> (define-type routine name procedure)
> (define plus (make-routine "plus" +))
> (define minus (make-routine "minus" -))
> (define (perform p1 routine p2)
    (print p1 " " (routine-name routine) " " p2 " is "
           ((routine-procedure routine) p1 p2)
           "\n"))
> (perform 5 plus 10)
5 plus 10 is 15
> (perform 7 minus 2)
7 minus 2 is 5
```

It can be said that closures pave the way for poor man's object orientation. There are object systems built atop closures. An example is Meroon.

Subdivide Code into Modules

In Gambit, a good way to subdivide code is by using the X module system.

The layout is pretty straight-forward:

create a .scm-file . Start it with `(module name)` where name is the name.

Then, declare what other modules to use with the `(use)` procedure. For instance:

```
(use lib/srfi1 lib/http-client)
```

You do always have access to the `(use)` procedure from the REPL, so you can load any module you want whenever that way.

`(use)` loads all the modules you specified to it, and all of their dependencies as well.

Though before loading a module, it checks if there is a compiled version of it. If there is, and there is a source file that's newer than the binary version, it recompiles the module. If there is a compiled version, it choses it rather than the source version.

More sophisticated module handling features are found in the build module. To get access to it, `(use build)`.

`(compile-all-modules!)` or `(cam!)` for short, compiles all modules that are not compiled. Passing it #t for parameter will make it continue compiling subsequent modules even if there are compilation errors.

`(module-clean! '[module name as a symbol])` erases the compiled version of a module.

`(module-compile '[module name as a symbol])` /re-/compiles a module.

Object oriented Programming

There is quite plenty of buzz about object orientation as “the modern way to design applications” or “the best way to develop software”. Oh my God.

Object orientation, single- and multiple-inheritance, is one particular abstraction. Particular abstractions fit particular problems. Thus, there is an interval of problems that object orientation is either suboptimal or directly unfit for. Just like screwdrivers.

While I must admit that I do see advantages with common object orientation systems, I think these do not compare in any way with the capacities of everything evaluates, closures, macros, continuations. Common Lisp is an interesting language in that it has an extensive object orientation system called CLOS, while being a Lisp.

How to abbreviate Code (that is how to write Code that writes Code, or Macros)

There's an interval of cases when the ways to program we described until now where the code could still be more concise. It helps you a lot in creating DSLs.

Macros are rules for code transformation that are applied to code before the Scheme environment starts to execute it.

Scheme implementations always come with `define-syntax`, which is a hygienic code pattern matching search and replace mechanism. Hygienic means doesn't interfere with variables that the macro wasn't given exclusive access to through having them passed to it as parameters.

Most Scheme implementations ship with `define-macro` as well. They are not hygienic.

It's a search and replace mechanism at the level of the parse tree. We'll take a look at it here, both because it provides a powerful way of solving programming problems, and also because it provides us an interesting application of lists.

The use of macros multiplicates the contents of the macro on each use case, so application code size increases much more than when using procedures. Also, `define-macro` wipes out line numbering information from any debug output of macro invocations. So, use macros solely where the more basal ways of solving problems prove suboptimal.

Before getting to how macros work, let's clarify our definitions on how procedures work.

Declaring a procedure in the global namespace or right after a `let`/`let*`/`letrec` local variables declaration can be defined like this:

```
(define (name parameter1 parameter2 etc.)
  code)

(print "The procedure returned " (name (+ 1 2) (+ 3 4)) ".\n")
```

When a procedure is invoked, all of its parameters are evaluated in sequential order (in this case, it's `(+ 1 2)` first, and then `(+ 3 4)`, giving 3 and 7, respectively). After this, the procedure is invoked with the parameter values. The invocation is about executing the code, where the parameters (here, parameter1, parameter2, etc.) are local variables containing the parameter values. The invocation is generally ended by the completion of the code and the return of the procedure, generating a return value that is given to the caller.

All of this is made during run-time. That is, if you have an application that contains the code above, and you compile it, and you run it, then it will be executed when you run it. Not at the time you compile it.

Now let's get to how macros work.

```
(define-macro (name parameter1 parameter2 etc.)
  code)

(print "The procedure returned " (name (+ 1 2) (+ 3 4)) ".\n")
```

When a macro is invoked, its code is executed in a way that its parameters (here, parameter1, parameter2, etc.) are local variables containing the code of the parameters in symbol format (that is, parameter1 contains `'(+ 1 2)` and parameter2 contains `'(+ 3 4)`). The invocation is about executing the code, and is generally ended by the completion of the code and the return of the macro, generating a return value, that the Scheme environment uses as a replacement for the current content of the macro invocation in the parse tree. This is done at parse time. That is, if you compile code, it's

made at the time that you compile, and not at the time of execution of the compiled variant, and if you invoke a macro in an interpreter (that is, in the REPL, or by using `(eval)`), it's made right before it starts executing the code in the interpreter.

Whoa!

So now we've worked out the concept briefly. Let's see how it looks in practice.

Example A

Code:

```
(begin
  (define (experimental-procedure . params)
    (display " In code! ")

    (begin
      (map (lambda (v)
             (display " -- ")
             (display (symbol->string v))
             (display " ** "))
           params))

    "Ending code!")

  (define d 'letter-d)
  (define e 'letter-e)
  (display "a\n")
  (display (experimental-procedure d e e))
  (display "b\n")
  )
```

Output at compile time:

(none)

Output at run time:

```
a
 In code!  -- letter-d **  -- letter-e **  -- letter-e ** Ending
code!b
```

Explanation:

Upon pre-processning (which is made at compile-time):

The code is compiled as usual.

Upon runtime:

Variableres `d` and `e` are defined to `'letter-d` and `'letter-e`.

`a`[newline] is printed to the console.

`d`, `e`, `e` are evaluated to `'letter-d`, `'letter-e`,`'letter-e`.

`experimental-procedure` is invoked so that params is fed with these as a list, that is, `'(letter-d letter-e letter-e)` .

`" In code! "` is printed.

For each and one of the parameters, the following is done:

> `" -- "` is printed, and the respective symbol is printed. `" ** "` is printed. The return value from display is returned, which is `#!void` .

map returns, with the return value `(list #!void #!void #!void)`. The return value is garbage collected.

`experimental-procedure` returns with the return value `"Ending code!"`.

The return value, `"Ending code!"`, is printed.

`b`[newline] is printed.

Example B

Code:

```
(begin
  (define (experimental-procedure2 . params)
    (display " In code! ")

    `(begin
       ,@(map (lambda (v)
                `(begin
                   (display " -- ")
                   (display ,(symbol->string v))
                   (display " ** ")))
              params)

    "Ending code!"))

  (define d 'letter-d)
  (define e 'letter-e)
  (display "a\n")
```

```
(write (experimental-procedure2 d e e))
(display "b\n")
)
```

Output at compile time:

(none)

Output at run time:

```
a
 In code! (begin (begin (display " -- ") (display "letter-d")
(display " ** ")) (begin (display " -- ") (display "letter-e")
(display " ** ")) (begin (display " -- ") (display "letter-e")
(display " ** ")) "Ending code!")b
```

Explanation:

Upon pre-processning (which is made at compile-time):

The code is compiled as usual.

Upon runtime:

The variables `d` and `e` are defined to `'letter-d` and `'letter-e`.

`a`[newline] is printed to the console.

`d`, `e`, `e` are evaluated to `'letter-d`, `'letter-e`,`'letter-e`.

`experimental-procedure2` is executed in a way that `params` is fed with these as a list, that is `'(letter-d letter-e letter-e)` .

" `In code! "` is printed.

`experimental-procedure2` generates a return value, which is a list, which looks like this: `'(begin X "Ending code!")` , where X is the return value merged, of the following function, executed for each and one of the parameters in `params`:

> Return the list `'(begin (display " -- ") (display Y) (display " ** "))` , where Y is replaced with the respective symbol.

The return value from `experimental-procedure2` is printed.

`b`[newline] is printed.

Example C

Code:

```
(begin
  (define-macro (experimental-macro . params)
    (display " In code! ")
    `(begin
       ,@(map (lambda (v)
                `(begin
                   (display " -- ")
                   (display ,(symbol->string v))
                   (display " ** ")))
              params)

    "Ending code!"))

  (define d 'letter-d)
  (define e 'letter-e)
  (display "a\n")
  (display (experimental-macro d e e))
  (display "b\n")
  )
```

Output at compile time:

```
 In code!
```

Output at run time:

```
a
 -- d **  -- e **  -- e ** Ending code!b
```

Explanation:

Upon pre-processning (which is made at compile-time):

The parse tree contains the code as shown above.

The pre-processor starts to traverse the code.

When it encounters `(experimental-macro .. )` , it matches it with the function `experimental-macro`.

The specified parameters `d`, `e`, `e` are taken straight off, without being evaluated first, and are passed to experimental-macro according to the parameter definition, that is in the form of a list. In this case: `'(d e e)`.

`" In code! "` is printed.

Experimental-macro generates a return value,which is a list, which looks like this: `'(begin X "Ending code!")` , where X is the return value merged, by the following function, executed for one and each of the parameters in `params`:

> Return the list `'(begin (display " -- ") (display Y) (display " ** "))` , where Y is replaced with the respective symbol.

`'(experimental-macro d e e)` in the parse tree is replaced with the result from experimental-macro, which is `'(begin (begin (display " -- ") (display "d") (display " ** ")) (begin (display " -- ") (display "e") (display " ** ")) (begin (display " -- ") (display "e") (display " ** ")) "Ending code!")` .

Right now, the contents of the parse tree, that is, the resulting code, is:

```
(begin
  (define-macro (potatis-makro . params)
    (display " In code! ")
    `(begin
       ,@(map (lambda (v)
                `(begin
                   (display " -- ")
                   (display ,(symbol->string v))
                   (display " ** ")))
              params)

    "Ending code!"))

  (define d 'letter-d)
  (define e 'letter-e)
  (display "a\n")
  (display
    (begin
      (begin (display " -- ") (display "d") (display " **
"))
      (begin (display " -- ") (display "e") (display " **
"))
      (begin (display " -- ") (display "e") (display " **
"))
      "Ending code!"))
  (display "b\n")
  )
```

The code is compiled as usual.

Upon runtime:

The variables `d` and `e` are defined to `'letter-d` and `'letter-e`.

`a`[newline] is printed to the console.

The following is executed: `(display " -- ") (display "d") (display " ** ") (display " -- ") (display "e") (display " ** ") (display " -- ") (display "e") (display " ** ")` so their respective strings are printed.

`"Ending code!"` is returned.

`display` prints the return value, that is `"Ending code!"` is printed.

`b`[newline] is printed.

Exception handling

Gambit has exception handling. R5RS does not have that built-in. R6RS does.

It works like this: You can raise anything

```
> (raise 3)
*** ERROR IN (console)@243.1 -- This object was raised: 3
1>

> (raise "hi")
*** ERROR IN (console)@245.1 -- This object was raised: "hi"
1>
```

Typically you may want to use `error` for raising errors. `error` in turn builds an error structure, and passes it to `raise`.

```
> (error "boo")
*** ERROR IN (console)@246.1 -- boo
1>
```

Generally you want to catch exceptions with `with-exception-catcher`.

```
> (define (divide a b)
    (with-exception-catcher
      (lambda (e) #f)
      (lambda () (/ a b))))
> (divide 3 2)
```

```
3/2
> (divide 3 0)
#f
```

There are procedures to get more information about the exception.

Handle binary Data

Binary data is generally handled best using u8vector vectors and -ports in Gambit.

Create, manipulate, read u8vector:

```
> (define v (make-u8vector 5))
> v
#u8(0 0 0 0 0)
> (u8vector-set! v 3 2)
> v
#u8(0 0 0 2 0)
> (u8vector-ref v 2)
0
> (u8vector-ref v 3)
2
```

Pipe output to u8vector:

```
> (call-with-output-u8vector
    '(char-encoding: UTF-8)
    (lambda (p) (display "Test!   " p)))
#u8(84 101 115 116 33 32 32 32)
```

Serialize to u8vector:

```
> (call-with-output-u8vector
    '(char-encoding: UTF-8)
    (lambda (p) (write "Test!   " p)))
#u8(34 84 101 115 116 33 32 32 32 34)
```

Deserialize from u8vector:

```
> (call-with-input-u8vector '#u8(34 84 101 115 116 33 32 32 32
34) read)
"Test!   "
```

There's s8 vectors, and 16-, 32- and 64-bit variants as well.

Interface C or C++ Code

C is interfaced through `c-lambda`, `c-define`, `c-initialize`, `c-define-type` and `##c-declare`. Sometimes code using these is referred to as Foreign Function Interface code.

These are only handled by the compiler, trying to execute them in interpreted mode fails. They are well described in the Gambit manual.

`c-lambda` works similarly to lambda in that it takes a list of arguments and then code. It works like `(c-lambda ([list of arguments that should be passed to C, each described by its variable type]) [return value type] [C code])`. The argument values are accessed as `___argX` or `___argX_voidstar` where X is their index number from C. The return value is passed into `___result` or `___result_voidstar`.

`c-declare` takes one string parameter, which is a strip of C or C++ code that should be inlined in the C code Gambit outputs. `c-initialize` is like c-declare though the code is executed on startup. `##c-declare` is like c-declare but does not need to be put at the top level of the code.

`c-define` is about defining a procedure in Scheme code that is available from C and Scheme.

`c-define-type` is about making a Scheme representation of a C data type. You can declare a C function that will be executed upon the garbage collection of each instance of the type.

Here's a `number->string` implementation for integers using the C standard library:

```
(c-declare "#include <stdio.h>")

(define integer->string
  (c-lambda (integer) UTF-8-string #<<e
    char[21] s;
    sprintf(&s,"%i",___arg1);
    ___result_voidstar = &s;
e
    ))
```

These mechanisms provide you with the tools for integrating your Scheme code very tightly with external routines, like libraries such as libzip, Cairo or OpenSSL, or operating system mechanisms such as the Windows GUI API:s.

Code formatting

C, C++, C#, D, Java, Javascript, Ruby, Python, Pascal, Actionscript tend to have an

integer multiple for indentation depth, and it's 4-5 spaces each, i.e.

```
// This is the root level
function a() {
     // Indented one level
     if (true) {
          // Indented two levels
          {
               // Indented three levels
          }
     }
}
```

There is good reason for this: Indentation is made for each started block of code. Since generally code has maximum five levels of blocks of code, this indentation works great.

Scheme code is a bit of a different story, since 90% of all opening parentheses technically start a block of code. Just look at this:

```
(let loop ((a b))
  (define (c)
    (d e))
  (if f
      (and g
           (let ((h i))
             (set! j k)
             (loop ((if l m n) (o) (- p (or q r))))))
      (loop (or s (if (> t (u v)) x y)))))
```

Therefore, indentation of Scheme code is a little bit less obvious, and a little bit more free-style, than in most other languages. Indentation is generally two characters.

```
(define (a)
  (b c)
  (begin
    (c d)
    (e f))
  (if g
    (h)
    (j)))
```

The big exception is when parameters to procedures are distributed over multiple lines. Then, indentation is made so that the parameters start at the same column.

```
(a-procedure parameter1
             parameter2
             parameter3)
```

This makes Scheme code's indentation more varied than most languages'.

Now let's focus on Scheme only

The Heart of Functional Programming (that is, create anew instead of mutating and having a state)

The moral of functional programming is: Don't mutate, don't keep a state.

Variable mutations is change a variable's content. Unlike some functional programming languages, Scheme supports mutations, like the general programming languages C, Java etc.. These are made with `set!`.

State is about having one or more variables that describe the current state of execution.

Instead of mutating, generate values anew.

There's ample of litterature on this subject.

Here's a simple example of the difference between imperative style and functional programming:

Imperative:

```
(let ((a 5) (b 6))
     (let ((res (+ a b)))
          (set! b 7)
          (set! res (+ res (* a b)))
          (println res)))
```

Functional:

```
(println ((lambda (a b)
            ((lambda (res b)
               (+ res (* a b)))
             (+ a b) 7))
            5 6))
```

Multi-dimensional Flows of Execution and avoid Event-driven programming (that is First-Class Continuations)

Scheme features a rare language construct, that is one of the most powerful there are, and that makes ground for software complexity in a rare way: First-class continuations.

In non-continuations based programming, each thread of execution is occupied with entering a function, evaluating, or returning, in a linear way.

A continuation is one possible future of program execution. Using first-class continuations, you get one possible future of program execution stored in a variable. Like with any variable, you can have multiple of them. And, the continuations (i.e. different possible futures) can invoke each others.

In non-continuations based programming, a return statement is always executed at most once per entrance into a function.

```
function a() {
     things();
     return something;
}
```

But in first-class continuations-based programming, since you can store possible futures in variables, and since you can invoke these whatever many times you want to, a function can return more times than it returns. Also, functions can enter more times than they return, without the application terminating.

Continuations has nothing to do with threading. Even if there's some conceptual similarities, they are separate concepts.

Here's how continuations work: They are created using `call/cc` (that's `call-with-current-continuation` abbreviated). It takes one procedure for parameter, and this procedure will in turn be invoked with one procedure for parameter. That last procedure is the continuation. The possible future the continuation addresses is the return of the `call/cc`. The parameter that's passed to it, is the value that will be returned by `call/cc`. If the procedure passed as a parameter to `call/cc` returns without invoking the continuation, it will return as normal.

Here is an example that does not display any features unique to call/cc, but still gives an example of how to use it:

```
> (call/cc (lambda (k) 123))
123
```

Now let's start playing with the continuation.

```
> (define continuation #f)
> (print "The continuation returned "
         (call/cc (lambda (k) (set! continuation k) "normally"))
         ".\n")
The continuation returned normally.
> (continuation 1)
The continuation returned 1.
> (continuation 2)
The continuation returned 2.
> (continuation)
The continuation returned #<unknown>.
```

See? In the procedure above, we stored the future of returning from `call/cc` into the variable `continuation`. Then each time we invoke continuation with a parameter, that future will be invoked with that value for return value from `call/cc` in place, and thus, this `print` evaluation will proceed, with that value for second argument. It's not more difficult than that.

The difference between having continuations support in a programming environment or not, is about as big as not having a garbage collector, or threads.

There is a lot of depth about what you can do with continuations. It's a very powerful language construct. For instance, Scheme's exception handling is implemented using continuations.

A very good example of how continuations can spare you from event driven programming is in continuations-based web frameworks. For an example, see Weblocks common-lisp.net/project/cl-weblocks/ .

In non-continuation-based languages, web pages as provided by a web server are usually individual files. If they are not, they are data entries in a table somewhere, which is roughly the same thing. The way these pages link to each other goes like this:

- Page A defines a link to page B, through specifying the URL of page B.
- When page B gets a request to it, there is HTTP header information that tells that the source was page A, so in this way it can act accordingly.

Take a web shop for example, there's a product list page, a debiting information page, a debiting processing page, a debiting failed page, and a debiting finished page.

In conventional web frameworks, in order to ensure that the user's flow of browsing pages is correct, each page needs to contain checking code. For instance, when entering the debiting processing page, did the user really come from the debiting information page? If not, what do?

The problem with this design approach is that as the number of pages increases, the

amount of checking logics required to cover the entire interval of possible actions from the user's side increases exponentially. And in the general programming languages, that means something about an exponential increase in code size.

So what's the problem with it? Well, it requires the programmer to design logics that target the complete interval of unintended behavior specifically.

The situation is quite equal in many regular pieces of application programming: we need to describe to the computer what the pattern is of how user interface components may be interacted with in a user interface, and what should happen in all possible cases. We need to describe to the computer what the pattern is of how the methods of an object should be invoked, and what should happen in all possible cases.

If this is how event-driven programming works:

```
(define (products-page user-comes-from)
  (if [user comes from the right place for this page]
      [do what this procedure was intended for. Return page
contents.]
      [error handling. return some kind of page contents.]))

(define (card-processing-page user-comes-from)
  (if [user comes from the right place for this page]
      [do what this procedure was intended for. Return page
contents.]
      [error handling. return some kind of page contents.]))

(define (payment-failure-page user-comes-from)
  (if [user comes from the right place for this page]
      [do what this procedure was intended for. Return page
contents.]
      [error handling. return some kind of page contents.]))

(define (payment-success-page user-comes-from)
  (if [user comes from the right place for this page]
      [do what this procedure was intended for. Return page
contents.]
      [error handling. return some kind of page contents.]))
```

Then this is how continuations-based web programming works:

```
(define (entire-payment-process user-comes-from)
  (define (products-page)
    (let ((user-action (show [products page contents])))
      (if (eq? user-action 'purchase)
        (processing-page)
        [handle other types of conduct])))

  (define (processing-page)
    (let ((processing-result 'not-done))
```

```
      (thread-start!
        (make-thread! (lambda () (make-debit [debiting
information])))))
      (let loop ()
        (show [processing page contents])
        (if (eq? processing-result 'not-done) (loop)))
      (if processing-result
        (payment-success-page)
        (payment-failure-page))))

  (define (payment-failure-page)
    (show [payment failure page])
    (products-page))

  (define (payment-success-page)
    (show [payment success page]))

  (products-page))
```

Did you see what we did? We assigned the task of handling the complete set of undefined intervals into the `show` procedure.

I would say that usage of continuations to invert event-driven programming into one block of code that defines all existing defined behaviors is of great mental benefit to the programmer. It's about communicating with the computer in a language much more close to our own (where our own is: I want a products page. It should list the products the user chose, and give the user the choice to proceed to purchase. If chosing to purchase, it should do the payment processing meanwhile showing a progress meter, and then lead to the failure or success pages. The failure page is about showing the user a failure message, and then returning to the products page. The success page is about showing the user a success message, and then our process is finished.)

Let's create a concrete example that displays the unique benefits of first-class continuations clearly. (Read the definition of `spawn` first after you grasped the other parts of the example.)

```
> (define (spawn prog)
      (call/cc
        (lambda (start)
          (let* ((resume #f)
                 (suspend (lambda ()
                            (call/cc
                              (lambda (k)
                                (set! resume (lambda () (k #f)))
                                (start resume))))))
            (prog suspend))))))
> (define resume
    (spawn (lambda (suspend)
             (print "Entered, so this is the first
iteration!\n")
```

```
                (suspend)
                (display "Second iteration.\n")
                (suspend)
                (let loop ()
                  (display "Subsequent iteration.\n")
                  (suspend)
                  (loop)))))
Entered, so this is the first iteration!
> (resume)
Second iteration.
> (resume)
Subsequent iteration.
> (resume)
Subsequent iteration.
> (resume)
Subsequent iteration.
```

Of course the `(spawn)` procedure could be extended to pass values between `(suspend)` and `(resume)`.

Rob Warnock summarized the logics of continuations in a humorous comp.lang.lisp Usenet posting, groups.google.com/group/comp.lang.lisp/msg/4e1f782be5ba2841 .

Hibernate Blocks of Code, with or without a State, to Disk or distribute it across Machines live (that is serialize and deserialize Procedures, Closures, Continuations)

Here's the next step: You can serialize them into binary format, send them over the network to other nodes or store them in a database, and then resume their execution through deserializing and invoking them whenever.

This feature is the cornerstone of Termite Scheme, a DSL for distributed computing.

Here's a brief intro to how it works:

Define a node identifier for the local node. Initialize it. (The number 3000 is for what TCP port to listen to on the respective node for incoming communication from other Termite nodes. Set it to whatever you want, an integer generally between 1000 and 32767.)

```
(define local-node (make-node "localhost" 3000))
(node-init local-node)
```

Define a node identifier for the other node. Replace othernode with the hostname or IP of the other node on the respective machine.

```
(define other-node (make-node "othernode" 3000)))
```

spawn is about spawning a process locally.

```
> (spawn (lambda () (print "Process running!\n")))
Process running!
```

remote-spawn is about spawning a process on a remote host.

```
(define process (remote-spawn other-node (lambda () (print "I'm
in the computer!\n"))))
```

Processes can communicate with each other, also across nodes.

`!` sends a message to another node. `?` waits for a message until it is received. `!?` sends a message, and waits for a response.

Just to make the point on closure and continuation serialization clear:

If you remember the counter closure example from the section on closures, we got a counter that worked like this

```
> (first-counter)
3
> (first-counter)
4
```

etc. . Closure serialization implies that you can send it whenever to other machines. Do

```
> (remote-spawn other-node (lambda () (print "The next two
iterations of the counter are " (first-counter) " and " (first-
counter) ".\n")))
```

And “The next two iterations of counter are 5 and 6.” would pop up on its screen. Let's do the same thing about the suspend/resume continuations example: Do

```
Entered, so this is the first iteration!
> (remote-spawn other-node resume)
```

and “Second iteration.” would be printed on its screen.

Part of Termite's philosophy is that any of the actions of communication you do with other processes may fail. `!` does not guarantee that the message was received. `remote-spawn` does not guarantee that a process was spawned. It is up to the program using Termite to implement

More on Termite in the Termite paper, see termite.googlecode.com/files/termite.pdf .

Guess a Code Blob!

It will happen someday that you end up just in the middle of someone else's code, and you need to navigate. How do you?

```
                          (parser-error
                           port
                           "[uniqattspec] after NS expansion
broken for "
                           name-value)))
                    (make-empty-attlist)
                    proper-attrs)
                   namespaces
                   elem-content))))))))))
(define (ssax:read-external-id port)
  (let ((discriminator (ssax:read-NCName port)))
    (assert-curr-char ssax:S-chars "space after SYSTEM or
PUBLIC" port)
    (ssax:skip-S port)
    (let ((delimiter (assert-curr-char '(#\' #\")
                                       "XML [11], XML [12]"
                                       port)))
      (cond
       ((eq? discriminator (string->symbol "SYSTEM"))
        (begin0
         (next-token '() (list delimiter) "XML [11]" port)
         (read-char port)))
       ((eq? discriminator (string->symbol "PUBLIC"))
```

(Quote from ssax/SSAX-code.scm of SSAX-SXML)

In the small context:
Try to get to the beginning of the expression or declaration that you are originating from.

Generally pay attention to the presence of use of macros, as these may have major impact on what code is valid and how the code is evaluated.

Carefully look after notation related to lists, that is, backticks (`), single quotes (`'`), and commas (`,`), as their presence would define when and how the code is executed. Probably it would reflect the programmer's intent as well.

Carefully watch parentheses.

Pay attention to the distinction `let`/`let*`/`letrec`, as these determine the scope of the local variables defined.

In the bigger context:

It should be said that pieces of code less than 2500 lines generally should be manageable

for the unintroduced reader to navigate pretty quickly.

Generally, well-written Scheme code is manageable or easy for the unintroduced reader to read, because Scheme code tends to be close to how human language would describe what the code does. It may be required to take time to get an understanding of how the particular DSL of the particular code works, and by what reasons it looks exactly like it does.

So what Drawbacks are there?

There is a disposition to simplicity in Gambit R5RS Scheme, and Scheme in general, that while carrying the features out of which all other thinkable features can be derived, it occasions it not to have pre-built procedures for solving certain everyday problems.

For example, there is no one pre-built procedure for reading the contents of an entire file into a string, but it is easily constructed by facilitating two or three related pre-built procedures. The same applies to writing one string into one file.

This even more strengthens Scheme's role as a language in which to write languages. Also, it may require you to put a limited amount of time into structuring your Scheme development environment in this respect.

You may need to develop or customize libraries for functionality you require. It is more so in Scheme than in many other languages, because Scheme gives the programmer a much wider interval of possible ways to solve problems.

Like we have already said, Scheme is a language to write languages. Or, to put it similarly, Scheme is a language in which to implement DSLs. It may be that you have a particular problem for which there is already a very well made DSL. For instance, the Unix Shell and related utilities provide a very mature DSL for handling files and program execution at the operating system level.

Some Conclusions

Did you notice you can do everything using this?

I don't know if you noticed that you can do everything you can do in the primary other programming languages in Scheme.

If you really didn't feel this until now, I encourage you to delve through your other programming languages of preference, and look in depth on how you do it in Scheme. And, spend time in Scheme, and do the reverse.

Scheme allows you to do software development ranging from very very high-level to low-level.

Your program is really one s-expression, and your Scheme environment is an s-expression virtual machine!

I don't know if you noticed, but did you think about that the global namespace is pretty much a list, and that every variable, procedure and macro declaration are lists too?

So at the level of concept, what your Scheme environment does for you is is to put this s-expression at your fingertips, and let you traverse it any way you want to. It optimizes parts of it into binary format at your command.

It is appropriate to look at a Scheme environment as an XML virtual machine.

Let's step back a bit.

The major responsibilities of contemporary operating systems are to do disk, network and peripherals I/O, memory management, multitasking, sanctioning of clock-cycles and working memory, access privileges management.

The major responsibilities of a Scheme environment is to do file and network I/O, memory management, generally multitasking with sanctioning of timeslices, code handling (macro expansion and code compilation), debugging, inter-code access privileges (that is namespaces), object serialization and -deserialization.

Do you see the conceptual correlations?

Your code just became very compact, and it's quite similar to human language (that is Domain Specific Language, or DSL for short)

Did you notice that now

- that you have access to all procedures, macros and variables just by referencing them by name,
- that you have macros that abbreviate all repetitive code away from your core logics code,
- that you have closures and first-class procedures that makes code mobilization across your application highly flexible, and
- that you have first-class continuations that spare you from event driven programming,

Your code is really Very Short and Very Concise!

And the interval of what's realistic to implement increased a lot,

And the relative amount of time spent on thinking about what the computer should really do, as compared to spending all days getting it doing something.

How advantageous, isn't it?

Did you notice how holistic an approach this is to software development?

My feel about Scheme is that it integrates the concepts of our current paradigm of software development into the most well balanced mix that we have within contemporary high-level programming.