The Adventures of a Pythonista in Schemeland

Release 0.1

Michele Simionato

June 02, 2010

CONTENTS

1 A bit of history **3**
- 1.1 My target . . . 3
- 1.2 A bit of history: Fortran and Lisp . . . 4
- 1.3 The algorithmic language Scheme . . . 5

2 About Scheme implementations **7**
- 2.1 About Scheme implementations . . . 7
- 2.2 About the library problem . . . 8
- 2.3 Additional difficulties . . . 10

3 Of parentheses and indentation **15**
- 3.1 Of parens and indentation . . . 15
- 3.2 About the prefix syntax . . . 17

4 Scheme bibliography (and a first program) **19**
- 4.1 Scheme resources for beginners . . . 19
- 4.2 A simple Scheme program . . . 20

5 About tail call optimization (and the module system) **23**
- 5.1 There are no `for` loops in Scheme . . . 23
- 5.2 There is no portable module system . . . 25
- 5.3 A simple benchmark . . . 27

6 The danger of benchmarks **31**
- 6.1 Beware of wasted cycles . . . 31
- 6.2 Beware of cheats . . . 33
- 6.3 Beware of naive optimization . . . 34
- 6.4 Recursion vs iteration . . . 35

7 Symbols and lists **39**
- 7.1 Symbols . . . 39
- 7.2 Lists . . . 41
- 7.3 Some example . . . 42

8 Quoting and quasi-quoting **45**
- 8.1 Quoting . . . 45

8.2 Quasi-quoting . . . 46
8.3 Programs writing programs . . . 48
8.4 Appendix: solution of the exercises . . . 50

9 Introduction to (sweet-)macros **51**
9.1 A minimal introduction to Scheme macros . . . 51
9.2 Which macrology should I teach? . . . 52
9.3 Enter sweet-macros . . . 53
9.4 An example: `multi-define` . . . 55

10 Features of (sweet-)macros **57**
10.1 `syntax-match` and introspection features of sweet-macros . . . 57
10.2 A couple of common mistakes . . . 58
10.3 Guarded patterns . . . 59
10.4 Literal identifiers . . . 61

11 The multiple evaluation problem (and easy-test) **63**
11.1 The problem of multiple evaluation . . . 63
11.2 Taking advantage of multiple evaluation . . . 65
11.3 A micro-framework for unit tests . . . 66

12 Are macros really useful? **71**
12.1 Are macros "just syntactic sugar"? . . . 71
12.2 About the usefulness of macros for application programmers . . . 73
12.3 Appendix: a Pythonic `for` loop . . . 74

13 Micro-introduction to functional programming **77**
13.1 A minimal introduction to functional programming . . . 77
13.2 Functional data structures: pairs and lists . . . 78
13.3 Functional update . . . 79

14 Currying, partial application, and fold **83**
14.1 Higher order functions and curried functions . . . 83
14.2 Partial application: cut and cute . . . 85
14.3 fold-left and fold-right . . . 86

15 List destructuring **89**
15.1 About pattern matching . . . 89
15.2 A list destructuring binding form (let+) . . . 91

16 Multiple values (and opt-lambda) **95**
16.1 list destructuring versus let-values . . . 95
16.2 Variadic functions from unary functions . . . 97
16.3 Further examples of destructuring: opt-lambda . . . 98

17 List comprehension **101**
17.1 The APS library . . . 101
17.2 Implementing list comprehension . . . 103
17.3 A tricky point . . . 105

18 Streams **107**
18.1 The eight queens puzzle . . . 107
18.2 Iterators and streams . . . 108
18.3 Lazyness is a virtue . . . 110

19 The R6RS module system **113**
19.1 Modules are not first class objects . . . 114
19.2 Compiling Scheme modules vs compiling Python modules . . . 115
19.3 Compiling is not the same than executing . . . 116

20 The compilation and evaluation strategy of Scheme programs **119**
20.1 Interpreter semantics vs compiler semantics . . . 119
20.2 Macros and helper functions . . . 121
20.3 A note about incremental compilers and interpreters . . . 122
20.4 Discussion . . . 123

21 The different meanings of phase separation **125**
21.1 Compile-time, run-time and optimization-time . . . 125
21.2 Strong vs weak phase separation . . . 127
21.3 A note about politics . . . 129

22 The Dark Tower of Meta-levels **131**
22.1 An easy-looking macro with a deep portability issue . . . 133
22.2 Negative meta-levels . . . 134
22.3 Meta-levels greater than one . . . 135
22.4 Discussion . . . 136

23 Separate compilation and import semantics **139**
23.1 The mysterious import semantics . . . 140
23.2 More implementation-dependent details . . . 142

24 Mutating variables across modules **145**
24.1 Mutating internal variables . . . 145
24.2 Mutating variables across phases . . . 146
24.3 Cross-phase side effects and separate compilation . . . 147
24.4 Conclusion . . . 150

25 Back to macros **151**
25.1 Writing your own programming language . . . 152
25.2 Recursive macros with accumulators . . . 153
25.3 A trick to avoid auxiliary macros . . . 154

26 Macros taking macros as arguments **157**
26.1 Scheme as an unfinished language . . . 157
26.2 Two second order macros to reduce parentheses . . . 159
26.3 The case for parentheses . . . 160

27 Syntax objects **163**
27.1 What `syntax-match` really is . . . 165
27.2 What macros really are . . . 166

27.3 A nicer syntax for association lists . . . 167

28 Hygienic macros **169**
28.1 `syntax-match` vs `syntax-rules` . . . 169
28.2 `syntax-match` vs `syntax-case` . . . 170
28.3 `syntax-match` versus `define-macro` . . . 170
28.4 The hygiene problem . . . 171

29 Breaking hygiene **175**
29.1 `datum->syntax` revisited . . . 175
29.2 Playing with the lexical context . . . 177
29.3 Hygienic vs non-hygienic macro systems . . . 178

30 Comparing identifiers **181**
30.1 `symbol-identifier=?` . . . 181
30.2 `bound-identifier=?` . . . 182
30.3 `free-identifier=?` . . . 184
30.4 Literal identifiers and auxiliary syntax . . . 184

31 Indices and tables **187**

Contents:

CHAPTER

ONE

A BIT OF HISTORY

This is the first episode of a long running series of articles about Scheme. Currently I have published the first 11 episodes of it on Stacktrace. This episode is a revised translation of http://stacktrace.it/2008/02/le-avventure-di-un-pythonista-schemeland-1/

1.1 My target

As you can imagine from the title, this series has been written from the point of view of a Python programmer. Nevertheless, it should be easy to follow for any programmer familiar with any dynamic language such as Perl, Ruby, PHP, Tcl, etc. In general all those languages (let me call them mainstream dynamic languages) are similar: interpreted, very dynamic, with a strong support for scripting (ok, maybe PHP does not fit the last point, but you get the idea ;)

Scheme is different. Even if it is very dynamic and well suited for scripting, very often it is also compiled (both to native code or to a target language such as C, Java or the CLR) and can work at C speed. Moreover Scheme is a functional language and it makes use of a set of functional idioms which are unknown in the mainstream languages. Finally, Scheme offers to its user an extremely advanced macrology (actually it has the most advanced macro system I know) and extremely powerful features (such as continuations) without equivalent in other languages.

That means that learning Scheme is not trivial: actually it takes of lot of effort and motivation to master it. If you see a programming language just as a tool to perform a given job in the smallest possible time, and your job is not programming language research, then you should not learn Scheme. Scheme is for people who want to know the many possible ways of performing the same job, who want to understand the advantages and the disadvantages of the different approaches, who want to explore programming paradigms.

I do not think that the first kind of programmers (let say the engineers) is inferior to the second kind (let say the explorers) or viceversa. It all depends on where your interest lies. If you are doing bioinformatics and you researching a cure for a genetic sickness I expect you to solve your problem in the smallest possible amount of time with a specialized library without dispersing your efforts reinventing the wheel. On the other hand, if you are a Computer Science professor I would expect to you to know many different languages and programming languages paradigms, having reinvented many wheels.

Pythonistas, generically speaking, are in between: they are pragmatic programmers who want to do a real job, but they are also persons which are not content with the first language they find,

otherwise they would stay for their entire life programming Visual Basic, Java or C++. They are both engineers and explorers at the same time (you could say that a good engineer should be a bit of an explorer, too, especially in a fast changing field such a programming).

This series is meant for programmers that fit the description I have just given. Its main goal is to discuss a few features of the Scheme programming language, with the aim to solicit your curiousity and make you think if you can learn something useful from this language which is dismissed by most as being just an academic language.

1.2 A bit of history: Fortran and Lisp

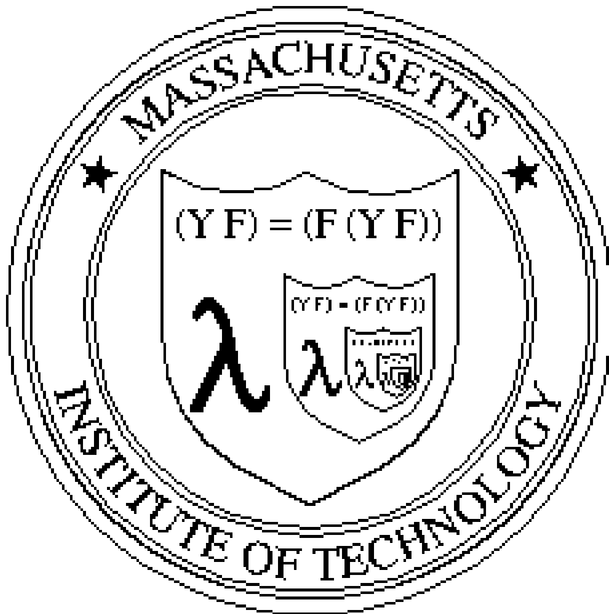
The history of programming begins with two languages with two completely different philosophies and goals: Fortran and Lisp. Both languages come from Academia, but from two opposite fields: on one side we find physicists and engineers interested in numeric computations to be run in the most efficient way to solve concrete problems of physics/engineering; on the other side we find matematicians interested in algorithmic research trying to solve abstract problems like symbolic computation, theorem proving, artificial intelligence and related topics.

Both fields had first class brains and the result of their effort were Fortran, which is still - after fifty years - the reference point for numeric computation, and Lisp, which is still the reference point for metaprogramming techniques. Both languages had and still have and enormous success in they market niche and will be probably still be us one hundred years from now. Nevertheless, both Fortran and Lisp are nowadays languages of small visibility, since their niches has become very small and far away from what we mean as mainstream programming today.

The reason for the little popularity of Fortran is clear: the language has been designed with one and only one goal in mind, efficency in numeric computation (*number crunching*). For everything else, Fortran, is not an appealing choice. Nowadays, most programmers have no reason to write libraries for floating point computations (they are already written, or they are only written by specialized people) so they have no need for Fortran. Also, C and C++ are nearly as efficient as Fortran and they have substantial advantages from the point of view of the interface with the operating system; moreover, most scientific tasks nowadays involves using a variety of technology and glue languages shine in this context: for instance you could use Python for writing the user interface and the visualization software, by calling underlying scientific libraries written in C or in Fortran.

The reason for the little popularity of Lisp is less clear: Lisp (I mean here Lisp in a large sense, intending the whole family of Lisp-derived languages including Scheme) is a general purpose language, it could do everything, it is nearly as fast as C, but nobody is using it. Newsgroups are full of flame wars between people claiming that Lisp is dead versus people claiming that Lisp is not dead at all and that it will be the language of the future. I will prudentelly avoid all these hot debates, I would not formulate any theory about the popularity of Lisp, and I will just discuss the Scheme language, leaving the reader to formulate his own opinion ;)

1.3 The algorithmic language Scheme



Scheme was born in 1975 (it is nearly twice as old as Python) as a dialect of the Lisp family. Nowadays by "Lisp" we refer usually to the language Common Lisp as standardized in 1989, well after Scheme. To discuss the differences and the advantages/disadvantages of Scheme with respect to Common Lisp would be long and I would expose myself to flame wars: usenet is full of furious discussions between Scheme and Lispers saying that their languages are completely different and that the opposite language is complete crap; nevertheless, anybody not knowning Scheme nor Lisp would have difficulty to distinguish one from the other (!)

Basically, both languages share a lot of features and a lot of what I will say about Scheme will apply to Common Lisp too. The biggest differences are sociological: the Scheme community is more academic and interested in research, experimentation and didactic; the Common Lisp community is closer to the IT business world and interested in solving real word problems. Of course this is a simplification but there is some truth in it. In the past, Scheme was meant to be a small language and it was particularly easy to implement; nowadays, this is not true anymore, since compliance with the latest Scheme specification requires a lot of work from the implementor side. Many people on the Scheme community are not happy with that, but a larger specification should in principle improve portability between implementations. Historically, Common Lisp was born as *union* of may features presents in Lisp dialects before standardization, whereas Scheme was born as *intersection* of the same features. The Revised Report 5 on the Algorithmic language Scheme (aka R5RS) says:

> *Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.* – William Clinger

As a consequence of this principle, all Scheme standards up to R5R6 are much smaller than the Common Lisp standard: actually too small, so that it is practically impossible to write "real" applications following the standard only. Recently people have tried to solve this issue by introducing a new standard, much bigger than the previous ones, the hotly debated R6RS (

Revised Report 6 on the Algorithmic language Scheme). The preparation of this standard has generated endless suffering in the Scheme community, since a significant minority has seen it as a betrayal of the spirit of Scheme. Nowadays Scheme is no more a little language: the R6RS requires a module system, a condition system, advanced macrology, a standard library, unicode support and many other features not requested before. Not only it is difficult to write a new implementation, it is also difficult to take an old R5RS implementation and to make it compatible with the new standard. Since R6RS is relatively recent (it was published in September 2007) there are few implementations of it. The first were Larceny and Ikarus; now there is also ypsilon. Moreover, PLT Scheme has grown an R6RS-compatibility mode. I will use Ikarus for my examples.

The installation procedure is trivial, it is enough to download the tarball and to compile with the usual `configure` and `make` dance. You can test that your installation works by invoking the interactive prompt:

```
$ ikarus
Ikarus Scheme version 0.0.3
Copyright (c) 2006-2008 Abdulaziz Ghuloum

> (display "hello world\n")
hello world
```

If you are running Windows, you may want to install Common Larceny, that runs on .NET.

This is the end: in the next episode I will discuss the problem of the implementations of Scheme and the issue of the portability of libraries. See you soon!

CHAPTER

TWO

ABOUT SCHEME IMPLEMENTATIONS

Scheme is a language with many implementations and with few libraries. In this episode I will discuss the current situation and I will give some useful indication to the Scheme beginner.

2.1 About Scheme implementations

One of the biggest problems for the Scheme beginner is the choice of the implementation. I did spend months on this issue and I have been on the verge of quitting many times. Since implementations are fairly different and incompatible if you make the "wrong" choice then you need to spend some effort to reconvert your code. Nowadays in theory this is less of an issue, since the R6RS report mandates an unique module system, but many implentations are still not supporting it. Moreover, in practice, in order to perform enterprise programming tasks you will always be forced to rely on implementation specific libraries such as database drivers and frameworks.

I am sure you will ask me what is the right implementation. The answer is that there is no right implementation: it depends on your needs. Every implementation has different advantages: there are implementations with a very good interoperability with C, others well integrated in Java or in .NET, others with a particularly good documentation, others with especially useful libraries, but there is no single implementation with all the features which is definitively superior to the others. You may use more than an implementation at the time, but you need to be careful in the choice of the libraries you are going to use, if you are interested in portability.

I cannot say I have tried all Scheme implementations (there are dozens and dozens of them) so take my obvervations *cum grano salis*. I did try PLT Scheme, Bigloo, Chicken, Guile, Ikarus and Larceny which are Open Source, multiplatform and free. Other major implementations are Chez Scheme (the interpreter, called Petit Scheme is free, the compiler is not) e and MIT Scheme (available with GPL licence) but I have not tried them and I cannot say anything. All the implementations I tried (except Guile) can be compiled and/or generate C code, and are usually faster than Python. Bigloo in particular is a "high performance compiler" optimized for floating point computations. PLT Scheme provides an interpreter, a compiler and an IDE called DrScheme: it is probably the biggest Scheme implementation out there and it is also probably the most used implementation and the one with most libraries.

Chicken is another big implementation: its major advantage is its author Felix Wilkelmann who literally perform miracles to support his users. I personally felt much more confortable

in the Chicken mailing list than in the PLT one, but it was a few years ago and of course your mileage may vary. Anyway, Chicken is the R5RS-compatible implementation I like most since it has a very practical attitude: it is written by people working in the industry and not in the academy. Chicken is a compiler from Scheme to C and it is extremely easy to write *wrappers* for C/C++ libraries. Moreover, there are already hundreds of interesting libraries available. They are called *eggs*, just as in Python, and they work more a less in the same way. However, it must be noticed that Chicken had eggs years before Python and more rights to use the name ;-)

Guile is the Scheme version sponsored by the Free Software Foundation and it is used as scripting language for GIMP; it has been dreamed that Guile would become the main scripting language for the FSF applications and that it would have replaced Emacs Lisp in Emacs, but that never happened.

There are many other implementations I have not cited here, but my advice is to stick to one of the major implementations, unless you have some very special need. Notice, however, that in my experience, even the so-called major implementations cannot compete with Python for what concerns reliability and professionality. It is not just that there are much less libraries of use for the enterprise programmer (database, GUI, Web, etc), they are also more immature and with more bugs. This clearly has to do with the fact that the total number of users of Scheme (including all implementations) is order of magnitudes smaller than the number of users of Python. I must say however that the situation has improved a lot in recent years.

2.2 About the library problem

Libraries are the weak point of Scheme; there is simply no competition between the number of available libraries for Python and for Scheme. Let me consider just GUI libraries: in Python it is possible to use practically any existing GUI toolkit. The most used are Tk, GTK, Qt, WxPython, etc. If you are lucky, you can find a Scheme implementation supporting one of those toolkits, but certainly not all of them. There are Scheme implementations where it is easy to write wrappers to C/C++ libraries, easier than in Python: however, it is you who must write the wrapper, whereas in Python there is always somebody who did the dirty work for you, and the wrapper is kept up-to-date without any cost for you.

Clearly having a community split in at least a dozen of major implementation does not help. You see the same issue, to a minor degree, in Common Lisp too. Languages with a reference implementation like Perl (which actually has a single implementation) or Python and Ruby (with many implementations, but only one reference implementation) have a substantial advantage for the point of view of the enterprise programmer, since the community attention is focalized on a single spot and everybody benefits from the work of everybody.

The Scheme community tried to improve the situation in various ways. One problem is that the R5RS report is underspecified, so a mechanism for proposing extensions to the standard

was invented, under the name of SRFI (Scheme Request For Implementation). To a Pythonista SRFIs will look a lot like PEPs (Python Enhancement Proposals). Everybody can submit a SRFI, i.e. a paper describing a library or a set of improvements to the language with the ambition of getting them into the standard. In principle the standard committee in charge of the next *Revised Report* would pick up from the best SRFIs for inclusion in the standard, but there is no obligation in this sense.

As a matter of fact, all existing implementations are making efforts to include the most important SRFIs, so that code using the SRFI libraries has better chances of being portable. Unfortunately, the R6RS editors have ignored many existing SRFIs, reinventing them in imcompatible ways and sometimes in inferior ways. The R6RS got a lots of critics and some Scheme implementations claimed that they will never be R6RS-compliant.

Every SRFI *must* be complemented by a working implementation, and this is the reason from the *I* at the end. The implementation must be as much portable as possible, therefore even if you are using a Scheme implementation which does not support the SRFI you are interested in natively, it is usually possible to port the SRFI with little effort. It is very important to study the most relevant SRFIs as soon as you learn Scheme, since if you want to write any practical application with it, you are going to need them.

2.3 Additional difficulties

I did start playing with Scheme in 2003: at the time, I had installation problems with all the implementations I tried (except Guile which is typically pre-installed in Linux and Cygwin). Nowadays things are simpler: basically all implementations provide packages that can be installed on Linux systems via apt-get/yum or other package managers, together with Windows and OS X installers. One thing which is still giving problem is GNU readline: whereas usually the Python version you find pre-installed on your system is readline-enabled, most Scheme implementations do not enable readline by default for reason of license, so you have to download the readline-dev headers, edit the Makefile and recompile everything by hand. This may be annoying, so I suggest you to use rlwrap instead, which can be installed with `apt-get` in Debian/Ubuntu or with``fink`` on the Mac.

rlwrap is a beatiful utility which can add readline support to each command line program (such as an interactive Scheme interpreter) that does not support it. It is enough to type `rlwrap <scheme-executable>` and your REPL magically gains readline line editing, persistent history and completion; moreover, you get *parens matching* for free, which is invaluable in Scheme programming. I make heavy usage of all these features.

By the way, I see now that I have used the term REPL (Read-Eval-Print-Loop) which may be unknown to a few readers; REPL is just the Lisp name for what is called the interactive interpreter or console in the Python world, the one with the >>> prompt. In Scheme the REPL is very well integrated with Emacs, so that you can position the cursor right after a closing parenthesis and send the corresponding expression to the REPL with CTRL-x-e (in Python you are forced to select the expression esplicitely instead, so that the user experience is not great as with Scheme). Of course you need a good Scheme mode: the default one is not so great and I use quack.el by Neil Van Dyke. The Emacs support for Lispish languages is excellent (which is not surprising at all, being Emacs written in a Lisp dialect) and I definitely suggest you to use Emacs as your IDE for Scheme. Of course, lots of people do not like Emacs, so you could use VI instead, or even a specialized Scheme IDE such as DrScheme provided by PLT Scheme. The important thing is to have support for parens matching.

Unfortunately, there is no equivalent to IPython and there will never be, since the language does not have support for docstrings, nor the introspection facilities of Python: you would need to switch to Common Lisp with SLIME to find something comparable or even better.

Generally speaking (with some exception) the support you can get for what concerns specific issues of a library is inferior to the support you can get with Python. The *comp.lang.scheme* newsgroup is friendly and can help you a lot if you ask how to implement a given algorithm or how a subtle Scheme construct works, but you should take in account that the number of posters in comp.lang.scheme is perhaps the 5% of the number of posters in comp.lang.python. On the other hand, the Schemers are highly esperienced and competent people, so you can get sound advice there.

All the Scheme implementations I tried are inferior to Python for what concerns introspection and debugging capabilities. Tracebacks and error messages are not very informative. Sometimes, you cannot even get the number of the line where the error occurred; the reason is that Scheme code can be macro-generated and the notion of line number may become foggy. On the other hand, I must say that in the five years I have being using Scheme (admittedly for toying and not for large projects) I have seen steady improvement in this area.

To show you the difference between a Scheme traceback and a Python traceback, here is an example with PLT Scheme, the most complete Scheme implementation and perhaps the one with the best error management:

```
$rlwrap mzscheme
Welcome to MzScheme v4.1 [3m], Copyright (c) 2004-2008 PLT Scheme Inc.
> (define (inv x) (/ 1 x))
> (inv 0)
/: division by zero

 === context ===
/usr/local/collects/scheme/private/misc.ss:68:7
```

As you see, there is not much information: in particular the information about the name of the function where the error occurred (`inv`) is lost and the line number/char number refers to the `read-eval-print-loop` code. You may contrast that with the Python traceback:

```
$ python
Python 2.5.1c1 (r251c1:54694M, Apr  5 2007, 12:45:14)
[GCC 4.0.1 (Apple Computer, Inc. build 5367)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def inv(x): return 1/x
...
>>> inv(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in inv
ZeroDivisionError: integer division or modulo by zero
```

I should mention however that PLT is meant to be run inside its own IDE, DrScheme. DrScheme highlights the line with the error and includes a debugger. However such functionalities are not that common in the Scheme world and in my own experience it is much more difficult to debug a Scheme program than a Python program.

The documentation system is also very limited as compared to Python: there is no equivalent to pydoc, no help functionality from the REPL, the concept of docstring is missing from the language. The road to Scheme is long and uphill; from the point of view of the tools and reliability of the implementations you will be probably better off with Common Lisp. However, in my personal opinion, even Common Lisp is by far less productive than Python for the typical usage of an enterprise programmer.

My interest here is different: I am not looking for a silver bullet, a language more productive than Python. My aim is to find a language from which a Pythonista can learn something. And certainly from Scheme we can learn *a lot*. But you will see what in the next episodes. See you

soon!

CHAPTER

THREE

OF PARENTHESES AND INDENTATION

In the previous two episodes I have discussed a few important subjects such as availability of libraries, reliability of implementations, support in case of bugs, etc. However, I have not said anything about the language *per se*. In this episode I will talk more about the language, by starting from the syntax, with a discussion of the infamous parentheses. Lisp parens have been the source of infinite debates from the very beginning and always will be. As you probably know, LISP means *Lots of Irritating Superfluous Parentheses*, and Scheme has even more parentheses than other Lisps!

3.1 Of parens and indentation

I did the mistake of writing Scheme code with an editor different from Emacs (the default *scheme mode* is terrible in my opinion). It has been like shooting in my foot. After a few weeks of suffering I came back to Emacs, I asked on comp.lang.scheme and comp.emacs and I was pointed out to excellent *scheme mode*, called quack.el and written by Neil Van Dike. Moreover, I have discovered how to augment the contrast of the parens (parens-edit mode) and I feel completely comfortable. But let me repeat that it is suicidal to try to edit Scheme/Lisp code without a good support from the editor. In my opinion this is the first reason why legions of beginners escape from Scheme/Lisp: who wants to be forced to learn Emacs only to write a few lines of code?

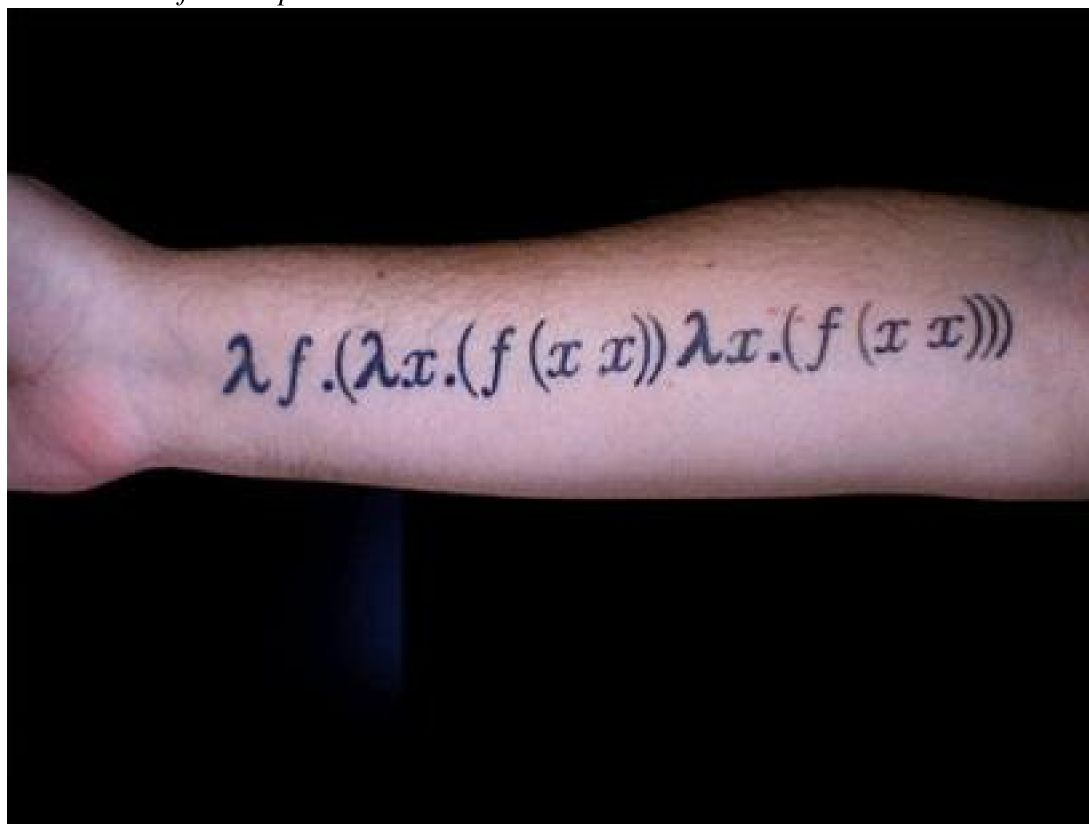
Of course, here I am exaggerating a bit, since there programmers that are able to write Lisp code even with vi and other editors, and there are even Scheme IDEs around (i.e. DrScheme, or a Scheme plugin for Eclipse): nevertheless I would still recommend Emacs to write Scheme/Lisp code, since Emacs itself is written in (Emacs) Lisp and that should tell something about its abilities to manage Lisp code. If you are coding in Common Lisp you should not miss SLIME, the Superior Lisp Interaction Mode for Emacs, which is a really really powerful IDE for Common Lisp.

However, even if I recommend Emacs and even if I think that the time spent to master it is time well spent, I do not think that *forcing* people to use a highly specialized tool to use a general purpose programming language is a good think. In theory, everybody should have the freedom to choose her editor, and it should be possible to program even in Notepad (even if

it is a thing I do not wish to anybody!). This in theory: in practice, every professional programmer use some dedicated tool to write code, so if you don't want to use Emacs please make sure your editor/IDE has a good Scheme mode, otherwise consider changing your developing environment.

It is interesting to notice what Paul Graham - a big name in the Lisp community and the main author of Arc, a new language of the Lisp family recently released and implemented in PLT Scheme - says about the parentheses:

> *We also plan to let programmers omit parentheses where no ambiguity would result, and show structure by indentation instead of parentheses. I find that I spontaneously do both these things when writing Lisp by hand on whiteboards or the backs of envelopes.*



Arc for the moment seems to require the parens, but it has a bit less parentheses than Scheme, as you can infer from the tutorial. It is clear that the parens are NOT necessary and one could imagine various tricks to reduce their number (I personally tried various approaches when I began programming in Scheme).

There is also an SRFI (SRFI-49: *Indentation-sensitive syntax*) that proposes to use indentation instead of parentheses taking inspiration from Python (!) The proposal should be considered as curiosity; discussing about indentation could have had some sense thirty years ago, at the time Scheme was designed. Nowadays, when 100% of Scheme code is written with parentheses, there is no point in not using them. Beginners would be penalized if they started using a style nobody uses.

In my (semi-serious) opinion, parens are a real *initiation test*: if a programmer cannot stand them than he is not worth of using Scheme/Lisp and he should address himself on inferior languages (i.e. *all* languages, according to what the majority of Schemers/Lispers think). In my

experience the snobish attitude is more common in the Common Lisp community whereas in the Scheme community there is more respect for the newbie. Anyway, the initiation test works and the average Scheme/Lisp programmer is usually smarter than the average programmer in other languages, since only the most persistent survive.

As a Pythonista I do not believe in those tricks: I think every language should be made accessible to the largest possible public. That means many second rate programmers will be able to learn it, but this is an opportunity, not an issue: the existence of poor programmers increases the number of available positions, since you need people to fix their mistakes! Otherwise how would you justify the number of job offers for Java and C++? (I said I was only semi-serious, don't take this personally, eh? ;)

Anyway, when after long suffering one has learned to manage with parens, there is no way back: once you have mastered a good editor the parens give you strong advantages when writing code. One of the main ones is automatic indentation with a single keypress. No more snippets of code send via e-mail and made unredable by the newlines added by the mail program; no more time wasted on reindenting code when refactoring.

Of course, nothing is perfect, and you may forget a paren here and there, but overall I will definitively admit that in the long run the parentheses pay off. On the other hand, in the short run, they make life much harder for the newbies; I still think that an optional syntax with less parentheses to be used by beginners or when using a poor editor would make sense, in a new Scheme-like langauge. But it is too late for Scheme itself: for the best or for the worst Scheme is a language full of parentheses and it is better to take full advantage of them.

Nota Bene: new languages based on *s*-expressions are born every day (the newcomers are Arc, which I have already cited, and Clojure, which runs on the Java platform and is very interesting). For those new languages it may have sense to investigate the available options. The best reference discussing alternative to parentheses that I know of is a paper by David Wheeler. It is an interesting reading, you may want to have a look at it, if you are interested in the topic.

3.2 About the prefix syntax

It is time to say something about another peculiarity of lispish language, the prefix syntax. In Scheme you do not write `1+1` as you have learned to write from elementary school. Instead, you write `(+ 1 1)`. The sum operator + is written at the beginning, as a prefix, and not in the middle, as an infix. I never had any trouble with infix syntax (I had trouble with parens instead) since it is something perfectly consistent: in Python the function name is written before the arguments too.

Actually, when you write `1+1` in Python, you should think of it like a shortcut syntax for the full prefix syntax `int.__add__(1,1)`, therefore the prefix syntax should not come as a surprise to a Pythonista. I was disturbed by the fact that there is no standard library functionality in Scheme to simplify the writing of mathematical formulas. I would have welcomed a standard macro able to convert infix syntax to prefix syntax in mathematical formulas, something like

```
(with-infix a+b*c) => (+ a (* b c))
```

Such a macro is standard in Common Lisp, but not in Scheme. Apart from forcing the students to write a parser to convert infix syntax to prefix syntax, I do not see the advantages of such a choice. This is however indicative of the difference between Python and Scheme: Python tries hard to make common tasks easy by providing a large library (the famous *batteries included* concept), whereas Scheme does not care.

Probably the real issue is that it is impossible to get consensus in the committee about the size of the standard library and about what to include in it, but the final result is that user is stuck with a very small standard library. Anyway, I should say that the standard library was much smaller before the R6RS, so the situation is improving. Moreover, concrete implementations often have a lot of useful (but non-portable) libraries.

But let me go back to syntax. It must be noted that the prefix syntax has enourmous advantages when macros enter the game. The absolute regularity of Scheme/Lisp programs, which are sequences of *s*-expressions of the form

```
(arguments ...)
```

where the arguments in turn can be nested *s*-expressions makes the automatic generation of code extremely effective. I will discuss this point in detail in future episodes; here I can anticipate that *Scheme code is not meant to be written by humans, it is intended to be written automatically by macros*. Only after having understood this point you will realize that the parentheses are a Good Thing (TM). I needed a few months to understand it, others never understand it and they quit Scheme with disgust.

If you will be able to pass the initiation test you will see that *s*-expressions (which are usually but not *necessarily* associated to parentheses) make sense. Once understood, the traditional (infix) notation becomes an obstacle more than a help. Moreover the total uniformity of Scheme programs has a kind of beauty and elegance in itself. No useless syntax, no boilerplate code, you breath an air of Zen minimalism.

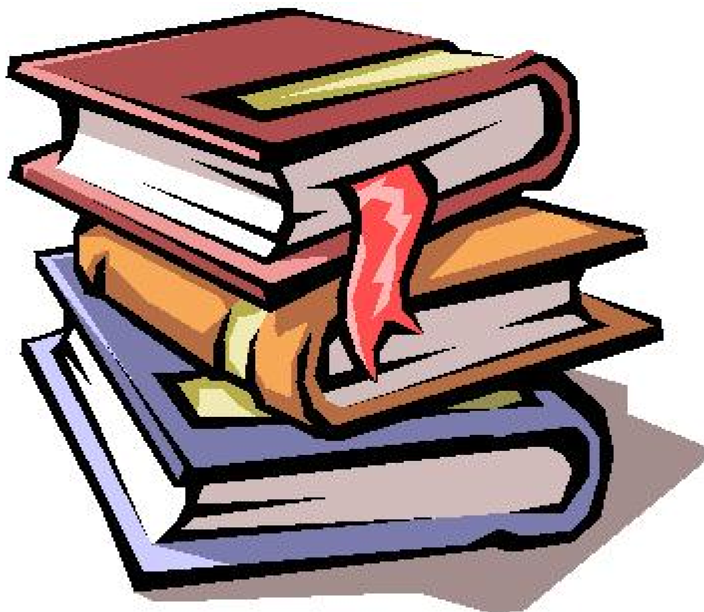
CHAPTER

FOUR

SCHEME BIBLIOGRAPHY (AND A FIRST PROGRAM)

4.1 Scheme resources for beginners

The present series has the ambition to be a nearly self-consistent complement to the R6RS document. In theory you should be able to learn Scheme by reading my "Adventures" and the R6RS only. In practice, however, having a look at other sources cannot hurt, so I will discuss here a few tutorials/textbooks you can find on the net and which are useful for Scheme beginners. There are many good texts, but none focused on my target of readers, i.e. experienced programmers coming from the scripting language world. I myself am the kind of persons that prefer learning from tutorials, articles and newsgroups more than from books, therefore I am not an expert on the existing bibiliografy. Here I will cite only a few resources, readers more knowledgeable are invited to post their recommendations as comments.



The main reference I used to learn Scheme when I started is Teach Yourself Scheme in Fixnum Days by Dorai Sitaram, which has many good things going for it. It is a self-consistent tutorial on Scheme which is very well written, especially for the first part. It is very informative, but at the same time concise and readable by hobbyist Scheme programmers like myself, i.e. by people using another language at work and not having too much free time at their disposal (I

suppose that description fits the majority of my readers). On the other hand, Teach Yourself Scheme in Fixnum Days is a bit old as a reference, and it completely ignores the modern Scheme macrology, based on pattern matching. It only describes the traditional macrology based on `define-macro`, which many seasoned Schemers do not love. My aim in this series is to give a description of modern Scheme, updated to the R6RS document; moreover I want to discuss in detail modern macros.

A more modern text (not covering the R6RS specification anyway) is The Scheme Programming Language (Third Edition) by R. Kent Dybvig. This is a very good book on Scheme in general. It is also the best reference I have read on `syntax-case` macros, but it is book with several hundreds of dense pages, perhaps too much for a hobbyist Schemer.

A very recent book is Programming Languages Application and Interpretation, by Shriram Krishnamurthi, which is also excellent, especially the part about continuations, but also very demanding from the reader, since it is a textbook for university students. Notice that even this book does not cover R6RS (to my knowledge there are no text books covering the R6RS standard, since it is too recent).

There is a habit of denoting Scheme books with their initials, so the two books I have just cited are also known as TSPL3 e PLAI; however, the most famous acronymous is certainly SICP, i.e. Structure and Interpretation of Computer Programs, by Harold Abelson e Gerald Jay Sussman. This book has been used to teach Scheme to generations of students and it is considered a *cult*, but I personally do not know it, therefore I cannot comment. Another book I have not read but I have heard good things about is How to Design Programs by Fellesein, Findler, Flatt and Krishnamurthi, which is a textbook for first year college students. It is up to you to check it and to see if you like it.

As you see, there are plenty of Scheme books, being Scheme a language with a great academical tradition. The problem is not the lack of books, is the lack of time to read them! This is one of the reasons why my *Adventures* are appearing as blog posts and not as a book: a short paper of 5-6 pages is much less scary than a big book of 500-600 pages. Morevoer blog posts are allowed to keep a much more informal tone than books, so they are both easier to write and to read.

4.2 A simple Scheme program

After so much talk, let me show you (finally!) a small example of Scheme program. There is a long tradition of giving the factional function as an example and I do not see a reason to break the tradition. Here is the Scheme code:

```
;; fac.scm for Chicken Scheme
(define (fac x)
   (if (= x 0) 1
    (* x
     (fac (- x 1)))))

(define n (string->number (car (reverse (argv)))))
(display (fac n))
```

The equivalent in Python would be:

```
import sys

def fac(x):
  if x == 0:
    return 1
  else:
    return x * fac(x-1)

n = int(sys.argv[-1])
print fac(n),
```

This trivial example already proves what I have been saying all along:

1. There are lots of parenthesis: five parens at the end of the factorial and four at the end of the definition of `n`. A typical program contains 3-4 parens per line. It should be noticed that all those parens are useless. By using the SRFI-49 the code could have been written as

```
define fac
   if (= x 0) 1
    * x
     fac (- x 1)

define n
 string->number
  car (reverse argv)
display (fac n)
```

2. The script is fully non-portable; to my knowledge it only works in Chicken Scheme. The reason is that the R5RS standard DOES NOT SPECIFY any way to read the command line arguments, hence `argv` is not standard.

   To a Pythonista such a lack looks absurd, but it is only after thirty years that the Schemers have decided how to manage `sys.argv` in the R6RS standard, which however is still little diffused and probably will remain a minority standard for years to come.

3. To get the last element of `argv`, Python uses the standard syntax `argv[-1]`; there is no standard function syntax to do it in Scheme, therefore or you use a non-portable function, or you reverse the list and you keep the first element with `car` (if you want to know the origin of the term you may have a look at this Wikipedia article): this is not

really readable, but readability never counted much in the Scheme world. Some Scheme implementations accepts the more readable name `first` as a synonimous of `car`, but this is again not standard.

4. The result of `fac` depends on the implementation: some implementations support infinite precision numbers (this is required by the R6RS) but some implementations do not. In particular in Chicken one gets

```
$ rlwrap csi
CHICKEN
Version 2.732 - macosx-unix-gnu-x86     [ manyargs dload ptables applyhook cross
(c)2000-2007 Felix L. Winkelmann        compiled 2007-11-01 on michele-mac.local
#;1> (define (fac x)  (if (= x 0) 1 (* x (fac (- x 1)))))
#;2> (fac 10)
3628800
#;3> (fac 100)
9.33262154439441e+157
#;4> (fac 1000)
+inf
```

In Ikarus (which is *R6RS-compliant*) one gets instead:

```
$ rlwrap ikarus
Ikarus Scheme version 0.0.2
Copyright (c) 2006-2007 Abdulaziz Ghuloum
> (define (fac (x) (if (= x 0) 1 (* x (fac (- x 1))))))
> (fac 10)
3628800
> (fac 100)
9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828625369792082722375825118521091686400000000000000000000000
> (fac 1000)
4023872600 ... < many many other digits>
```

After reading these first episodes you may be tempted to quit; I am sure the readers who followed me up to this point had this question floating in their minds: *is it really worth it?*. Probably for most readers the answer is *no*. But this series is for the most persistent readers, and I hope to show them something positive in the next episode. Keep reading and see you next time!

CHAPTER

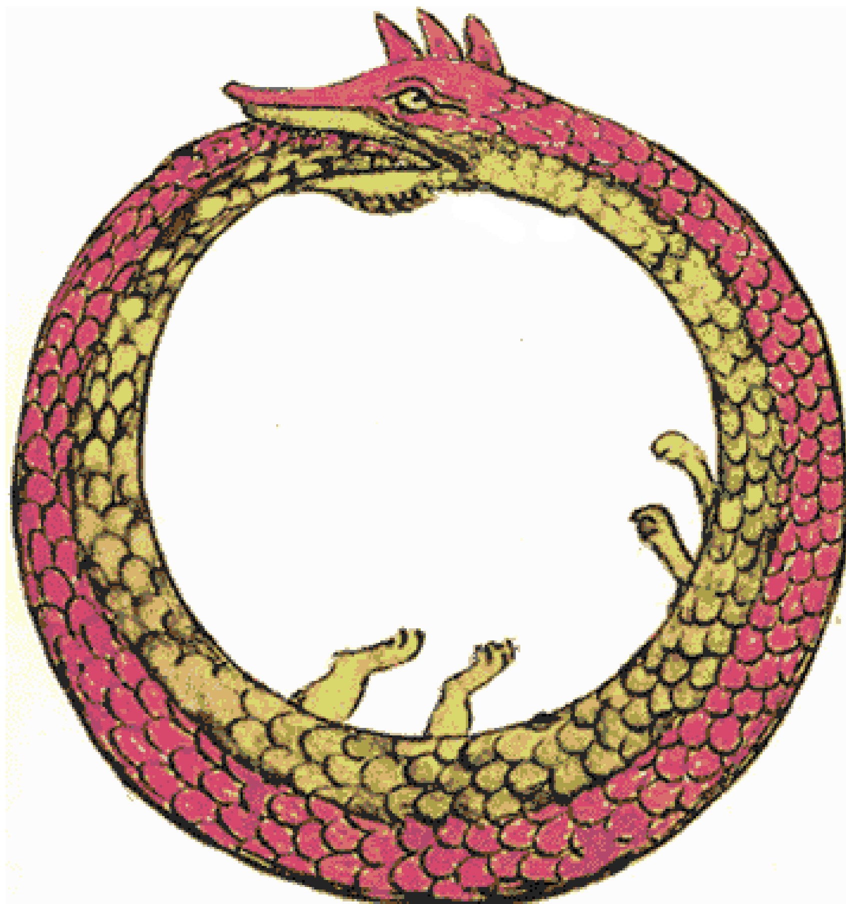
FIVE

ABOUT TAIL CALL OPTIMIZATION (AND THE MODULE SYSTEM)

In order to speak of Scheme performance one needs to write at least a few benchmarks. This is not completely trivial, for at least a couple of reasons. The first reason is shallow, but it may be baffling for beginners: since there is no `for` loop in Scheme, how are we supposed to write a benchmark, which is usually based on running many times the same instructions and measuring the total time spent? We will see in a moment that there is an easy solution to this question (use recursion). On the other hand, there is a second more serious question: since there is no fully portable way to write a library in Scheme, how can we write a benchmark library? There is no real answer, so we will restrict ourselves to R6RS-compliant Scheme implementations where there is a standard concept of library.

5.1 There are no `for` loops in Scheme

The `for` loop is missing in Scheme as a primitive construct since it is useless in a language that guarantees tail call optimization. If you studied the concept of *tail call* at college you know what I am talking about; on the other hand, if you forgot it, or if you did study Physics like me, it is worth spending a couple of words on the subject. The ones who want to know more, may consult this Wikipedia article.



The important point beyond the tail recursion concept is that it is always possibile to convert a `for` into a recursive function in *tail call* form, i.e. a recursive function returning a value or a call to itself. For instance, the Python loop:

```
# a loop printing 1 2 3
for i in range(1,4):
   print i,
```

can be converted to a recursive function `print_up_to_3`:

```
def print_up_to_3(i):
   if i == 4: return
   print i,
   return print_up_to_3(i+1)

print_up_to_3(1)
```

Here the last instruction of the function (the tail) is a call to itself, hence the name *tail call*.

Tail call optimization is guaranteed by the Scheme language. Scheme compilers/interpreters are able to recognize recursive functions in tail call form and to convert them internally in `for` loops. As a consequence, the programmer has no need to write `for` loops directly: she can just use recursive function. Our example would look as follows in Scheme:

```
(define (print-up-to-3 i)
   (unless (= i 4)
     (display i) (display " ")
     (print-up-to-3 (+ i 1))))

(print-up-to-3 1)
```

This works, but it is not really readable; to improve the situation Scheme provides a little syntactic sugar called *named let*:

```
(let loop ((i 1))
  (unless (= i 4)
     (display i) (display " ")
     (loop (+ i 1))))
```

Traditionally the function in the *named let* construct is called `loop` to make clear to the programmer that we are emulating a `for` loop. In this example `loop` is exactly equivalent to `print-up-to-3`.

Let me point out two things before closing this paragraph:

1. there are other `let` forms, used to define local variables. The simplest one is `let`:

   ```
   > (let ((a 1) (b 2)) (+ a b)) ; => 3
   ```

   The scope of `a` and `b` is limited to the current S-expression/form; if `a` and `b` are defined outside the `let` form, internally `a` and `b` *shadow* the outer names.

2. there is actually a `do` loop in the language, but it is cumbersome to use and redundant because the *named let* allows you to perform everything `do` does. I see it as a useless construct in a language that would like to be minimalist but it is not.

5.2 There is no portable module system

As I have anticipated, libraries are the weak point of Scheme. There are few libraries available and it is also difficult to write portable ones. The reason is that historically Scheme never had any standard module system until very recently, with the R6RS document: that means nearly all current implementations provide different and incompatible module systems.

In order to understand the reason for this serious lack, you must understand the philosophy behind Scheme, i.e. the so called MIT approach: things must be done well, or not at all. For thirty years the Scheme community has not been able to converge on a well done single module system. It is only in 2007 that a standard module system has been blessed by the Scheme committee: but even that was done with a lot of opposition and there are implementors who said they will *never* support R6RS.

As a consequence of history and mentality, if you want to write a library for implementation X, you need to do a lot of boring and uninspiring work to port the library to implementations Y, Z, W, ... (there are *dozens* of different implementations). Moreover, a few implementations

do not have a module system at all, so you may be forced to solve name clashes issue *by hand*, changing the names of the functions exported by your library, if they shadow names coming from third party libraries (!)

Personally, I picked up Scheme 5 years ago, but never used it because of the lack of a standardized module system. The main reason why I have decided to go back to Scheme and to write this series is the coming of the R6RS document last year. The R5RS standard has lots of defects, but at least now I can write a library and I can have people using different implementations install it and use it (nearly) seemlessly.

Since there is some hope for a large diffusion of R6RS module system in the future, I have decided to use it and to ignore implementations not supporting it. I should notice however that there are solutions to run R6RS modules on top of R5RS implementations, like the psyntax package, but I have not tried it, so I cannot comment on its reliability.

As first example of usage of the R6RS module system, I will define a `repeat` library exporting a `call` function which is able to call a procedure `n` times. Here is the code, that should be self-explanatory:

```
(library (repeat)
  (export call)
  (import (rnrs))

  (define (call n proc . args)
    (let loop ((i 0))
      (when (< i n) (apply proc args) (loop (+ 1 i))))))
```

The `export` declaration corresponds to Python's `__all__`: only the names listed in `export` are exported. In this case we will export only the function `(call n proc . args)`. Notice the dot in the argument list: that means that the functions accept a variable number of arguments, which are collected in the list `args`. In other words, `. args` is the moral equivalent of `*args` in Python, with some difference that we will ignore for the moment. The `apply` function applies the argument list to the input procedure `proc`, which is called many times until the index `i` reaches the value `n`.

`(import (rnrs))` imports all the libraries of the current version of the "Revised Report on Scheme", i.e. the R6RS report. At the REPL this is automatically done by the system, but for batch scripts it is mandatory (as Pythonistas say *explicit is better than implicit*). It is also possible to import subsections of the whole library. For instance `(import (rnrs base))` imports only the base library of the R6RS, `(import (rnrs io))` imports only the I/O libraries, et cetera.

The usage of the libray is trivial: it is enough to put the file `repeat.sls` somewhere in the Ikarus search path (specified by the environment variable `IKARUS_LIBRARY_PATH`). Then, you can import the library as follows:

```
$ rlwrap ikarus
Ikarus Scheme version 0.0.2
Copyright (c) 2006-2007 Abdulaziz Ghuloum
> (import (repeat))
> (call 3 display "hello!\n")
```

```
hello!
hello!
hello!
```

By default `(import (repeat))` imports all the names exported by the module `repeat`, something that a Pythonista would never do (it is equivalent to a `import * from repeat`); fortunately it is possible to list the names to be imported, or to add a custom prefix:

```
> (import (only (repeat) call)); import only call from repeat
call
#<procedure call>
> (import (prefix (repeat) repeat:)); import all with prefix repeat:
> repeat:call
#<procedure call>
```

5.3 A simple benchmark

The main advantage of Scheme with respect to Python is the performance. In order to show the differences in performance I will go back to the factorial example of episode 4. I will compare the following Python script:

```
# fact.py
import sys, timeit

def fact(x):
    if x == 0: return 1
    else: return x * fact(x-1)

if __name__ == '__main__':
    n = int(sys.argv[-1])
    t = timeit.Timer('fact(%d)' % n, 'from fact import fact')
    print t.repeat(1, number=10000000)
    print fact(n)
```

with the following R6RS-compliant script (written in Ikarus Scheme):

```
; fact.ss
(import (rnrs) (only (repeat) call) (only (ikarus) time))

(define (fac x)
  (if (= x 0) 1
      (* x (fac (- x 1)))))

(define n
  (string->number (car (reverse (command-line)))))
```

```
(time (call 10000000 (lambda () (fac n))))
(display "result:") (display (fac n)) (newline)
```

I will notice two things:

1. Python manages to compute the factorial of 995, but then it faces the stack wall and it raises a `RuntimeError: maximum recursion depth exceeded` whereas Scheme has no issues whatsoever;
2. In order to compute the factorial of 995 ten thousands times, Python takes 15.2 seconds, whereas Ikarus takes 7.2 seconds.

Notice that since the factorial of 995 is a *large* number, the computation time is spent in multiplication of large numbers, which are implemented in C. Python has its own implementation of long integers, whereas Ikarus uses the GNU Multiprecision library (gmp): the times measured here mean that the gmp implementation of long integers is more efficient than the Python one, but they say nothing on the relative performances of the two languages. It is more interesting to see what happens for small numbers. For instance, in order to compute the factorial of 7 for 10 millions of times, Python takes 30.5 seconds, whereas Ikarus takes 3.08 seconds and thus it is nearly *ten times* faster than Python. This is not surprising at all, since function calls in Python are especially slow whereas they are optimized in Scheme. Moreover Ikarus is a native code compiler.

It means Ikarus' REPL works by compiling expressions to native code, whereas Python's REPL compiles to bytecode. The technology is called incremental compilation and it is commonly

used in Lisp languages from decades, even if it may look futuristic for C/C++ programmers. The factorial example is not very practical (on purpose), but it is significant, in the sense that it is legitimate to expect good performances from Scheme compilers. The fastest Scheme compiler out there is Stalin, but I would not recommend it to beginners.

The next episodes will be devoted to the dangers of benchmarks, do not miss it!

CHAPTER

SIX

THE DANGER OF BENCHMARKS

Benchmarks are useful in papers and blog posts, as a good trick to attract readers, but you should never make the mistake of believing them: as Mark Twain would say, *there are lies, damned lies, and benchmarks*. The problem is not only that reality is different from benchmarks; the problem is that it is extremely easy to write a wrong benchmark or to give a wrong interpretation of it.

In this episode I will show some of the dangers hidden under the factorial benchmark shown in the previous episode, which on the surface looks trivial and unquestionable. If a benchmark so simple is so delicate, I leave to your imagination to figure out what may happen for complex benchmarks.

The major advantage of benchmarks is that they make clear how wrong we are when we think that a solution is faster or slower than another solution.

6.1 Beware of wasted cycles

An obvious danger of benchmarks is the issue of vasted cycles. Since usually benchmarks involve calling a function *N* times, the time spent in the loop must be subtracted from the real computation time. If the the computation is complex enough, usually the time spent in the loop

is negligible with respect to the time spent in the computation. However, there are situations where this assumption is not true.

In the factorial example you can measure the wasted cycles by subtracting from the total time the the time spent in the loop performing no operations (for instance by computing the factorial of zero, which contains no multiplications). On my MacBook the total time spent in order to compute the factorial of 7 for ten millions of times is 3.08 seconds, whereas the time spent to compute the factorial of zero is 0.23 seconds, i.e. fairly small but sensible. In the case of fast operations, the time spent in the loop can change completely the results of the benchmark.

For instance, `add1` it is a function which increments a number by one and it is extremely fast. The time to sum 1+1 ten millions of times is 0.307 seconds:

```
> (time (call 10000000 add1 1))
running stats for (call 10000000 add1 1):
    no collections
    307 ms elapsed cpu time, including 0 ms collecting
    308 ms elapsed real time, including 0 ms collecting
    24 bytes allocated
```

If you measure the time spent in the loop and in calling the auxiliary function `call`, by timing a `do-nothing` function, you will find a value of 0.214 seconds, i.e. 2/3 of the total time is wasted:

```
> (define (do-nothing x)  x)
> (time (call 10000000 do-nothing 1))
running stats for (call 10000000 do-nothing):
    no collections
    214 ms elapsed cpu time, including 0 ms collecting
    216 ms elapsed real time, including 0 ms collecting
    16 bytes allocated
```

Serious benchmarks must be careful in subtracting the wasted time correctly, if it is significant. The best thing is to reduce the wasted time. In a future episode we will consider this example again and we will see how to remove the time wasted in `call` by replacing it with a macro.

6.2 Beware of cheats

The issue of wasted cycles is obvious enough; on the other hand, benchmarks are subject to less obvious effects. Here I will show a trick to improve dramatically the performance by cheating. Let us consider the factorial example, but using the Chicken Scheme compiler. Chicken works by compiling Scheme code into C code which is then compiled to machine code. Therefore, Chicken may leverage on all the dirty tricks on the underlying C compiler. In particular, Chicken exposes a benchmark mode where consistency checks are disabled and the `-O3` optiomization of gcc is enabled. By compiling the factorial benchmark in in this way you can get incredible performances:

```
$  csc -Ob fact.scm # csc = Chicken Scheme Compiler
$ ./fact 7
./fact 7
0.176 seconds elapsed
      0 seconds in (major) GC
      0 mutations
      1 minor GCs
      0 major GCs
```

```
result:5040
```

We are *16* times faster than Ikarus and *173* times faster than Python! The only disadvantage is that the script does not work: when the factorial gets large enough (biggen than 2^31) Chicken (or better gcc) starts yielding meaningless values. Everything is fine until `12!`:

```
$ ./fact 12 # this is smaller than 2^31, perfectly correct
  0.332 seconds elapsed
      0 seconds in (major) GC
      0 mutations
      1 minor GCs
      0 major GCs
result:479001600
```

Starting from `13!` you get a surprise:

```
$ ./fact 13 # the correct value is 6227020800, larger than 2^31
  0.352 seconds elapsed
      0 seconds in (major) GC
      0 mutations
      1 minor GCs
      0 major GCs
result:-215430144
```

You see what happens when you cheat? ;)

6.3 Beware of naive optimization

In this last section I will show a positive aspect of benchmarks: they may be usefully employed to avoid useless optimizations. Generally speaking, one should not try to optimize too much,

since one could waste work and get the opposite effect, especially with modern compilers which are pretty smart.

In order to give an example, suppose we want to optimize by hand the factorial benchmark, by replacing the closure `(call 10000000 (lambda () (fac n)))` with the expression `(call 10000000 fac n)`. In theory we would expect a performance improvement since we can skip an indirection level by calling directly `fac` instead of a closure calling `fac`. Actually, this is what happens with: for `n=7`, the program runs in 3.07 secondi with the closure and in 2.95 seconds without.

In Chicken - I am using Chicken 2.732 here - instead, a disaster happens when the benchmark mode is turned on:

```
$  csc -Ob fact.scm
$ ./fact 7
   1.631 seconds elapsed
   0.011 seconds in (major) GC
       0 mutations
    1881 minor GCs
      23 major GCs
result:5040
```

The program is nearly ten times slower! All the time is spent in the garbage collector. Notice that this behavior is proper of the benchmark mode: by compiling with the default options you will not see significant differences in the execution time, even if they are in any case much larger (7.07 seconds with the closure versus 6.88 seconds without). In other words, with the default option to use the closure has a little penalty, as you would expect, but in benchmark mode the closure improves the performance by ten times! I asked for an explation to Chicken's author, Felix Winkelmann, and here is what he said:

In the first case, the compiler can see that all references to fac are call sites: the value of "fac" is only used in positions where the compiler can be absolutely sure it is a call. In the second case the value of fac is passed to "call" (and the compiler is not clever enough to trace the value through the execution of "call" - this would need flow analysis). So in the first case, a specialized representation of fac can be used ("direct" lambdas, i.e. direct-style calls which are very fast).

Compiling with "-debug o" and/or "-debug 7" can also be very instructive.

That should make clear that benchmarks are extremely delicate beasts, where (apparently) insignificant changes may completely change the numbers you get. So, beware of benchmarks, unless you are a compiler expert (and in that case you must be twice as careful! ;)

6.4 Recursion vs iteration

Usually imperative languages do not support recursion too well, in the sense that they may have a *recursion limit*, as well as inefficiencies in the management of the stack. In such a languages it is often convenient to convert ricorsive problems into iterative problems. To this aim, it is convenient to rewrite first the recursive problem in tail call form, possibly by adding auxiliary

variables working as accumulators. At this point, the rewritin as a `while` loop is trivial. For instance, implementing the factorial iteratively in Python has serious advantages: if you run the script

```
# fact_it.py
import sys, timeit

def fact(x):
    acc = 1
    while x > 0:
        acc *= x
        x -= 1
    return acc

if __name__ == '__main__':
    n = int(sys.argv[-1])
    t = timeit.Timer('fact(%d)' % n, 'from fact_it import fact')
    print t.repeat(1, number=10000000)
    print fact(n)
```

you will see a speed-up of circa 50% with respect to the recursive version for "small" numbers. Alternatively, you can get an iterative version of the factorial as `reduce(operator.mul, range(1, n+1)`. This was suggested by Miki Tebeka in a comment to the previous episode and also gives a sensible speedup. However notice that `reduce` is not considered Pythonic and that Guido removed it from the builtins in Python 3.0 - you can find it in `functools` now.

If you execute the equivalent Scheme code,

```
(import (rnrs) (only (ikarus) time) (only (repeat) call))

(define (fac x acc)
  (if (= x 0) acc
      (fac (- x 1) (* x acc))))

(define n
  (string->number (car (reverse (command-line)))))

(time (call 10000000 (lambda () (fac n 1))))
(display "result:") (display (fac n 1)) (newline)
```

you will discover that it is slightly *slower* than the non tail-call version (the tail-call requires less memory to run, anyway). In any case we are an order of magnituder over Python efficiency. If we consider benchmarks strongly dependent on function call efficiency, like the Fibonacci benchmark of Antonio Cangiano, the difference between Python and Scheme is even greater: on my tests Ikarus is *thirty* times faster than Python. Other implementations of Scheme or other functional languages (ML, Haskell) can be even faster (I tried the SML/NJ implementation, which is *forty* times faster than Python 2.5). Of course those benchmarks have no meaning. With benchmarks one can prove that Python is faster than Python is faster than Fortran and C++ in matrix computations. If you do not believe it, please read this ;)

That's all folks, see you next episode!

CHAPTER

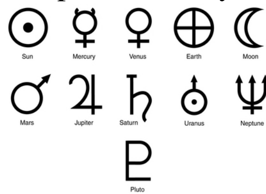
SEVEN

SYMBOLS AND LISTS

In this episode I pave the way to the heart of Lisp, i.e. to the famous *code is data* concept. In order to do that, I will have to introduce two fundamental data types first: symbols and lists.

7.1 Symbols

Scheme and Lisp have a particular data type which is missing in most languages (with the exception of Ruby): the *symbol*.



From the grammar point of view, a symbol is just a quoted identifier, i.e. a sequence of characters corresponding to a valid identifier preceded by a quote. For instance, `'a`, `'b1` e `'c_` are symbols. On the set of symbols there is an equality operator `eq?` which is able to determine if two symbols are the same or not:

```
> (define sym 'a)
> (eq? sym 'b)
#f
> (eq? sym 'a)
#t
```

`#f` e `#t` are the Boolean values False and True respectively, as you may have imagined. The equality operator is extremely efficient on symbols, since the compiler associates to every symbol an integer number (this operation is called *hashing*) and stores it in an interal registry (this operation is called *interning*): when the compiler checks the identity of two symbols it actually checks the equality of two integer numbers, which is an extremely fast operation.

You may get the number associated to a symbol with the function `symbol-hash`:

```
> (symbol-hash sym)
117416170
> (symbol-hash 'b)
```

```
134650981
> (symbol-hash 'a)
117416170
```

It is always possible to convert a string into a symbol and viceversa thanks to the functions `string->symbol` and `symbol->string`, however conceptually - and also practically - symbols in Scheme are completely different from strings.

The situation is not really different in Python. It is true that symbols do not exist as a primitive data type, however strings corresponding to names of Python objects are actually treated as symbols. You can infer this from the documentation about the builtin functions hash e intern, which says: *normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys*. BTW, if you want to know exactly how string comparison works in Python I suggest you to look at this post:

Scheme has much more valid identifiers than Python or C, where the valid characters are restricted to `a-zA-Z-0-9_` (I am ignoring the possibility of having Unicode characters in identifiers, which is possible both in R6RS Scheme and Python 3.0). By convention, symbols ending by `?` are associated to boolean values or to boolean-valued functions, whereas symbols ending by `!` are associated to functions or macros with side effects.

The function `eq?`, is polymorphic and works on any kind of object, but it may surprise you sometimes:

```
> (eq? "pippo" "pippo")
#f
```

The reason is that `eq?` (corrisponding to `is` in Python) checks if two objects are the same object at the pointer level, but it does not check the content. Actually, Python works the same. It is only by accident than `"pippo" is "pippo"` returns True on my machine, since the CPython implementation manages differently "short" strings from "long" strings:

```
>>> "a"*10 is "a"*10 # a short string
True
>>> "a"*100 is "a"*100  # a long string
False
```

If you want to check if two objects have the same content you should use the function `equal?`, corresponding to == in Python:

```
> (equal? "pippo" "pippo")
#t
```

It you know the type of the objects you can use more efficient equality operators; for instance for strings you can use `string=?` and for integer numbers =:

```
> (string=? "pippo" "pippo")
#t
```

```
> (= 42 42)
#t
```

To be entirely accurate, in addition to `eq` and `equal`, Scheme also provides a third equality operator `eqv?`. `eqv?` looks to me like an useless complication of the language, so I will not discuss it. If you are interested, you can read what the R6RS document says about it.

7.2 Lists

The original meaning of LISP was *List Processing*, since lists were the fundamental data type of the language. Nowadays Lisp implements all possible data types, but still lists have a somewhat privileged position, since lists can describe code. A Lisp/Scheme list is a recursive data type such that:

1. the list is empty: `'()`
2. the list is the composition of an element and a list via the `cons` operation (`cons` stands for *constructor*): `(cons elem lst)`.

For instance, one-element lists are obtained as composition of an element with the empty list:

```
> (cons 'x1 '()); one-element list
(x1)
```

Two-elements lists are obtained by composing an element with a one-element list:

```
> (cons 'x1 (cons 'x2 '())); two-element list
(x1 x2)
```

That generalizes to N-element lists:

```
> (cons 'x1 (cons 'x2 (cons 'x3 ..... (cons 'xN '()))) ...)
(x1 x2 x3 ... xN)
```

For simplicity, the language provides an N-ary list constructor `list`

```
> (list x1 x2 x3 ... xN)
(x1 x2 x3 ... xN)
```

but the expression `(list x1 ... xN)` is nothing else than syntactic sugar for the fully explicit expression in terms of constructors.

There are also the so-called *improper lists*, i.e. the ones where the second argument of the `cons` is not a list. In that case the representation of the list displayed at the REPL contains a dot:

```
> (cons 'x1 'x2) ; improper list
(x1 . x2)
```

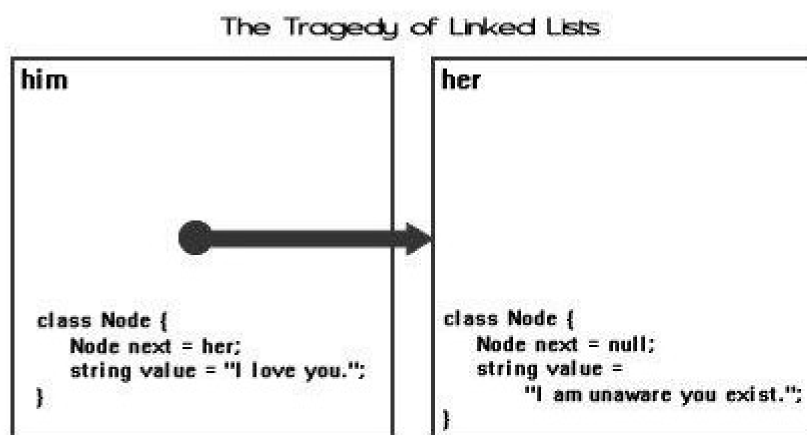
It is important to remember that improper lists *are not lists*, therefore operations like `map`, `filter` and similar do not work on them.

As we anticipated in episode 4, the first element of a list (proper or improper) can be extracted with the function `car`; the tail of the list instead can be extracted with the function `cdr`. If the list is proper, its `cdr` is proper:

```
> (cdr (cons 'x1 'x2))
x2

> (cdr (cons 'x1 (cons 'x2 '())))
(x2)
```

At low level Scheme lists are implemented as *linked list*, i.e. as couples *(pointer-to-sublist, value)* until you arrive at the null pointer.



7.3 Some example

To give an example of how to build Scheme lists, here I show you how you could define a `range` function analogous to Python `range`. Here are the specs:

```
> (range 5); one-argument syntax
(0 1 2 3 4)
> (range 1 5); two-argument syntax
(1 2 3 4)
> (range 1 10 2); three-argument syntax
(1 3 5 7 9)
> (range 10 0 -2); negative step
(10 8 6 4 2)
> (range 5 1); meaningless range
()
```

Here is the implementation:

```
(define range
 (case-lambda
   ((n); one-argument syntax
    (range 0 n 1))
   ((n0 n); two-argument syntax
    (range n0 n 1))
   ((n0 n s); three-argument syntax
    (assert (and (for-all number? (list n0 n s)) (not (zero? s))))
    (let ((cmp (if (positive? s) >= <=)))
      (let loop ((i n0) (acc '()))
        (if (cmp i n) (reverse acc)
            (loop (+ i s) (cons i acc))))))))
```

Here `case-lambda` is a syntactic form that allows to define functions with different behavior according to the number of arguments. `for-all` is an R6RS higher order function: `(for-all pred lst)` applies the predicate `pred` to the elements of list `lst`, until a false value is found - in that case it returns `#f` - otherwise it returns `#t`. Here the assertion checks at runtime that all the passed arguments `n0`, `n` and `s` are numbers, with `s` non-zero.

The first `let` defines a comparison function `cmp` which is >= if the step `s` is positive, or <= if the step `s` is negative. The reason is clear: if `s` is positive at some point the index `i` will get greater than `n`, whereas if `s` is negative at some point `i` will get smaller than `n`.

The trick used in the loop is extremely common: instead of modifying a pre-existing list, at each iteration a new list is created *ex-novo* by adding an element `(cons i acc)`; at the end the accumulator is reversed `(reverse acc)`. The same happens for the counter `i` which is never modified.

This is an example of *functional* loop; imperative loops based on mutation are considered bad style in Scheme and other functional languages. Notice by contrast that in Common Lisp imperative loops are perfectly acceptable and actually there is a very advanced LOOP macro allowing you to do everything with imperative loops.

Actually, the `range` function defined here is more powerful than Python's `range`, since it also works with floating point numbers:

```
> (range 1.3 2.5 .25)
  (1.3 1.55 1.8 2.05 2.3)
```

As always, the only way to really get Scheme lists is to use them. I suggest you try the following exercises:

1. implement an equivalent of Python enumerate for Scheme lists;
2. implement an equivalent of Python zip for Scheme lists.

I will show the solutions in the next episode. If you are lazy and you want an already written list library, I suggest you to give a look at the SRFI-1 library, which is very rich and available practically in all Scheme implementations. Many of the SRFI-1 features are built-in in the R6RS standard, but many other are still available only in the SRFI-1 .

CHAPTER

EIGHT

QUOTING AND QUASI-QUOTING

In this episode I will explain the meaning of the "code is data" concept. To this aim I discuss the quoting operation which allows to convert a code fragment into a list of symbols and primitive values - i.e. it converts code into data. Then, I discuss the issue of evaluating data as code.

8.1 Quoting

A distinguishing feature of the Lisp family of languages is the existence of a quoting operator denoted with a quote `'` or with `(quote )`, the first form being syntactic sugar for the second. The quoting operator works as follows:

1. on primitive values such as numbers, literal strings, symbols, etc, it works as an identity operator:

```
> '1
1
> '"hello"
"hello"
> ''a
'a
```

1. expressions are converted into lists; for instance `'(display "hello")` is the list

```
> (list 'display '"hello")
(display "hello")
```

whereas `'(let ((x 1)) (* 2 x))` is the list

```
> (list 'let (list (list 'x '1)) (list '* '2 'x))
(let ((x 1)) (* 2 x))
```

et cetera.

Hence every Scheme/Lisp program admits a natural representation as a (nested) list of symbols and primitive values: *code is data*. On the other hand, every nested list of symbols and primitive values corresponding to a syntactically valid Scheme/Lisp programs can be executed, both at *runtime* - with `eval` - or at *compilation time* - through macros. The consequences are

fascinating: since every program is a list, it is possible to write programs that, by building lists, build other programs. Of course, you can do the same in other languages: for instance in Python you can generate strings corresponding to valid source code and you can evaluate such strings with various mechanisms (`eval`, `exec`, `__import__`, `compile`, etc). In C/C++ you can generate a string, save it into a file and compile it to a dynamic library, then you can import it at runtime; you also have the mechanism of pre-processor macros at your disposal for working at compile time. The point is that there is no language where code generation is as convenient as in Scheme/Lisp where it is buil-in, thanks to *s*-expressions or, you wish, thanks to parenthesis.

8.2 Quasi-quoting

In all scripting languages there is a form of *string interpolation*; for instance in Python you can write

```
def say_hello(user):
     return "hello, %(user)s" % locals()
```

In Scheme/Lisp, there is also a powerful form of *list interpolation*:

```
> (define (say-hello user)
    `("hello" ,user))

> (say-hello "Michele")
   ("hello" "Michele")
```

The *backquote* or `(quasiquote )` syntax ``` ` ``` introduces a list to be interpolated (*template*); it is possible to replace some variables within the template, by prepending to them the *unquoting* operator `(unquote )` or `,`, denotated by a comma. In our case we are unquoting the user name, `,user`. The function `say-hello` takes the user name as a string and returns a list containing the string `"hello"` together with the username.

There is another operator like `unquote`, called `unquote-splicing` or comma-at and written `,@`, which works as follows:

```
> (let ((ls '(a b c))) `(func ,@ls))
(func a b c)
```

In practice `,@ls` "unpacks" the list `ls` into its components: without the splice operator we would get:

```
> (let ((ls '(a b c))) `(func ,ls))
(func (a b c))
```

The power of quasiquotation stands in the code/data equivalence: since Scheme/Lisp code is nothing else than a list, it is easy to build code by interpolating a list template. For instance, suppose we want to evaluate a Scheme expression in a given context, where the contexts is given as a list of bindings, i.e. a list names/values:

```
(eval-with-context '((a 1)(b 2) (c 3))
  '(* a  (+ b c)))
```

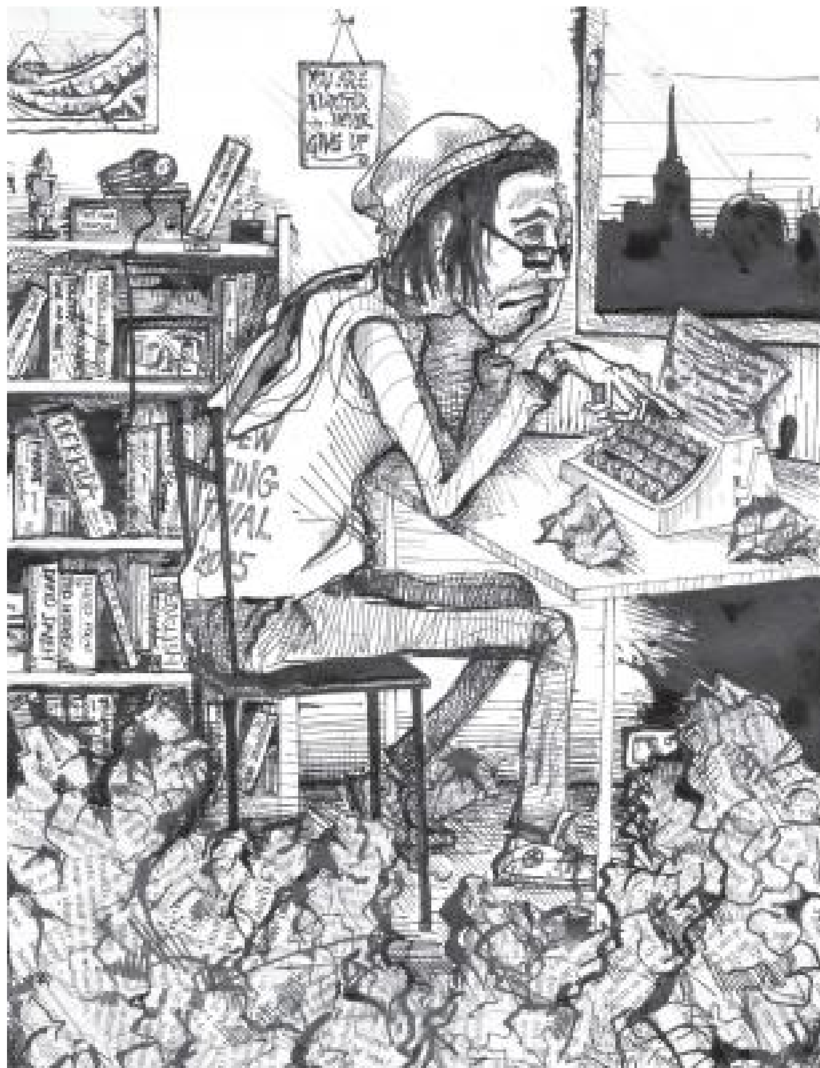
How can we define `eval-with-context`? The answer is by `eval`-uating a template:

```
(define (eval-with-context ctx expr)
 (eval `(let ,ctx ,expr) (environment '(rnrs))))
```

Notice that `eval` requires a second argument that specifies the language known by the interpreter; in our case we declared that `eval` understands all the procedures and macros of the most recent RnRS standard (i.e. the R6RS standard). The environment specification has the same syntax of an `import`, since in practice it is the same concept: it is possible to specify user-defined modules as the `eval` environment. This is especially useful if you have security concerns, since you can run untrusted code in a stricter and safer subset of R6RS Scheme.

`eval` is extremely powerful and sometimes it is the only possible solution, in particular when you want to execute generic code at runtime, i.e. when you are writing an interpreter. However, often you only want to execute code known at compilation time: in this case the job of `eval` can be done more elegantly by a macro. When that happens, in practice you are writing a compiler.

8.3 Programs writing programs



Once you realize that code is nothing else than data, it becomes easy to write programs taking as input source code and generating as output source code, i.e. it is easy to write a compiler. For instance, suppose we want to convert the following code fragment

```
(begin
 (define n 3)
 (display "begin program\n")
 (for i from 1 to n (display i)); for is not defined in R6RS
 (display "\nend program\n"))
```

which is not a valid R6RS program into the following program, which is valid according to the R6RS standard:

```
(begin
 (define n 3)
 (display "begin program\n")
 (let loop (( i 1)) ; for loop expanded into a named let
```

```
    (unless (>= i  n) (display i) (loop (add1 i))))
    (display "\nend program\n")))
```

`begin` is the standard Scheme syntax to group multiple expressions into a single expression *without introducing a new scope* (you may introduce a new scope with `let`) and *preserving the evaluation order* (in most functional languages the evaluation order is unspecified).

More in general, we want to write a script which is able to convert

```
(begin                                          (begin
  (expr1 ...)                                     (expr1' ...)
  (expr2 ...)                -->                  (expr2' ...)
   ...                                             ...
  (exprN ...))                                    (exprN' ...))
```

where the expressions may be of kind `for` or any other kind not containing a subexpression of kind `for`. Such a script can be thought of as a preprocessor expanding source code from an high level language with a primitive `for` syntax into a low level language without a primitive `for`. Preprocessors of this kind are actually very primitive compilers, and Scheme syntax was basically invented to make the writing of compilers easy.

In this case you can write a compiler expanding `for` expressions into named lets as follows:

```
(import (rnrs) (only (ikarus) pretty-print))

;; a very low-level approach
(define (convert-for-into-loop begin-list)
  (assert (eq? 'begin (car begin-list)))
  `(begin
     ,@(map (lambda (expr)
              (if (eq? 'for (car expr)) (apply convert-for (cdr expr)) expr))
            (cdr begin-list))))

; i1 is subject to multiple evaluations, but let's ignore the issue
(define (convert-for i from i0 to i1 . actions)
  ;; from must be 'from and to must be 'to
  (assert (and (eq? 'from from) (eq? 'to to)))
  `(let loop ((i ,i0))
     (unless (>= i ,i1) ,@actions (loop (+ i 1)))))

(pretty-print
 (convert-for-into-loop
  '(begin
     (define n 3)
     (display "begin program\n")
     (for i from 1 to n (display i))
     (display "\nend program\n"))))
```

Running the script you will see that it replaces the `for` expression with a *named let* indeed. It is not difficult to extend the compiler to make it able to manage sub-expressions (the trick is to use recursion) and structures more general than `begin`: but I leave that as an useful exercise. In a

future episode I will talk of *code-walkers* and I will discuss how to convert generic source code. In general, one can convert s-expression based source code by using an external compiler, as we did here, or by using the built-in mechanism provided by Scheme macros. Scheme macros are particularly powerful, since they feature extremely advanced pattern matching capabilities: the example I have just given, based on the primitive list operations `car/cdr/map` is absolutely primitive in comparison.

The next episode will be entirely devoted to macros. Don't miss it!

8.4 Appendix: solution of the exercises

In the latest episode I asked you to write an equivalent (for lists) of Python built-ins `enumerate` and `zip`. Here I give my solutions, so you may check them with yours.

Here the equivalent of Python `enumerate`:

```
(define (py-enumerate lst)
  (let loop ((i 0) (ls lst) (acc '()))
    (if (null? ls) (reverse acc)
        (loop (+ 1 i) (cdr ls) (cons `(,i ,(car ls)) acc)))))
```

and here is an example of usage:

```
> (py-enumerate '(a b c))
  ((0 a) (1 b) (2 c))
```

Here is the equivalent of Python `zip`:

```
(define (zip . lists)
 (apply map list lists))
```

Here is an example of usage:

```
> (zip '(0 a) '(1 b) '(2 c))
((0 1 2) (a b c))
```

Notice that `zip` works like the transposition operation in a matrix: given the rows, it returns the columns of the matrix.

CHAPTER

NINE

INTRODUCTION TO (SWEET-)MACROS

This episode is entirely devoted to Scheme macros from a personal point of view. Pattern matching is introduced as the fundamental mechanism on which macros are built.

9.1 A minimal introduction to Scheme macros

Scheme macros have many faces. You can see them as a general mechanism to extend the syntax of base Scheme, and also as a mechanism to reduce boilerplate. On the other hand, if you focus your attention on the fact that they work at compile time, you can see them as a mechanism to perform arbitrary computations at compile time, including compile time checks.

I think the correct way of looking at macros is to see them as a general facility to write compilers for micro-languages - or Domain Specific Languages, DSL - embedded in the Scheme language. The languages defined through macros can be very different from Scheme; for instance you can define object oriented languages (object systems such as TinyCLOS or Swindle are typical examples) or even languages with static typing (the new language Typed Scheme, built on top of PLT Scheme, is such an example).

In order to address such use cases, Scheme macros have to be extremely advanced, much more than Lisp macros and any other kind of macros I know of; as a consequence, they also have a reputation for complexity. Unfortunately, on top of the intrinsic complexity, Scheme macros also suffers from accidental complexity, due to history (there are too many standard macro systems) and to the tradition of not caring about readability (many Scheme constructs are made to look more complex than they actually are).

Scheme has two macro systems included in the *de jure* standard - `syntax-rules`, which allows to define hygienic macros only, and `syntax-case`, which has the full power of Lisp macros and much more - plus a *de facto* standard based on `define-macro` system, which is available in all implementations and it is well known to everybody because of its strict similarity to Common Lisp `defmacro` system.

9.2 Which macrology should I teach?

Since there are so many macro systems it is difficult to decide from where to start in a pedagogical paper or tutorial. If you look at the original Italian version of this paper, you will see that I did talk about `syntax-rules` macros first. However, after a lot of thinking, I have decided to go my own way in this English series of the *Adventures*. Here I will not discuss `syntax-rules`, nor I will discuss `syntax-case`: instead, I will discuss my own version of Scheme macros, which I called `sweet-macros`.

Why I am doing that? After all, why my readers should study my own version of macros when they surely will be better served off by learning the standard macrology used by everybody? I have spent *years* debating with myself this very question, but at the end I have decided to go this way for a series of reasons:

1. I regard the existence of two separate macro systems in the same standard as a wart of Scheme and as a mistake made by the R6RS editors: they should have included in the language `syntax-case` only, leaving `syntax-rules` as a compatibility library built on top of `syntax-case`;
2. I really don't like the `syntax-case` syntax, it is by far too verbose and unreadable; I find there is a strong need for some sugar on top of it and that is what `sweet-macros` are for;
3. `sweet-macros` are very close to `syntax-case` macros, so once you understand them you will understand `syntax-case` too; from there, understanding `syntax-rules` is a breeze;
4. starting from `sweet-macros` is much better from pedagogical purposes, especially for readers with a Common Lisp background, since it is easy to explain the relation with `defmacro` and the hygiene issue;
5. my target readers are programmers coming from the scripting languages (Perl/Python/Ruby) world. For this kind of public, with no previous exposition to Scheme, bare `syntax-case` is just too hard, so I needed to dress it in nice clothes to make it palatable;
6. `sweet-macros` are intended to easier to use than `syntax-case` macros, but they are also more powerful, since they provide introspection and debugging capabilities as well as guarded patterns, so they should look attractive to experienced users too; however, this is a nice side effect and not the main motivation for the library;
7. `sweet-macros` were written expressely for this series of papers, since I did not want to litter my explanation of Scheme macros with endless rants. So, I took action and I wrote my own library of macros made "right": this is also a tribute to the power of Scheme macros, since you can "fix" them from within the standard macro framework in fifty lines of code.

If you are an advanced reader, i.e. a Schemer knowing `syntax-case/syntax-rules` or a Lisper knowing `defmacro`, I a am sure you will ask yourself what are the differences of `sweet-macros` with respect to the system you know. I will make a comparison of the various systems in the future, in episode #12 and later on. For the moment, you will have to wait. I do not want to confuse my primary target of readers by discussing other macro systems

right now. I also defer to episode #12 the delicate question *are macros a good idea?*. For the moment, focus on what macros are and how you can use them. Then you will decide if they are a good idea or not.

9.3 Enter sweet-macros

My sweet-macros library is a small wrapper around the `syntax-case` macro system. I release it under a liberal BSD licence. You can do whatever you want with it, just keep the attribution right.

The primary goal of sweet-macros is semplicity, so it only exports three macros, `def-syntax`, `syntax-match` and `syntax-expand`:

- `def-syntax` is a macro used to define simple macros, which is similar to `defmacro`, but simpler and strictly more powerful.
- `syntax-match` is a macro used to define complex macro transformers. It is implemented as a thin layer of sugar on top of `syntax-case`.
- `syntax-expand` is a macro which acts as a debugging facility to expand macros defined via `def-syntax` or `syntax-match`.

It should be mentioned that standard Scheme macros do not provide debugging and/or introspection facilities and that every implementation provides different means of debugging

macros. This is unfortunate, since debugging macros is usually difficult and it is done often, since it is uncommon to get a macro right the first time, even if you are an experienced developer.

I wanted to provide my readers with the tools to understand what they are doing, without relying on the details of the implementation they are using. Therefore macros defined via `syntax-match` (and that includes macros defined via `def-syntax`) provide out of the box introspection and debugging features.

Of course, readers who want to rely on the debugging tools of their implementation can do so; for instance I hear that DrScheme has a pretty good macro stepper but I have not tried it since I am an Emacs-addict.

First of all, you should download and install the right sweet-macros library. Unfortunately the R6RS module system does not really solve the portability issue (to my endless frustration) so I had to write different versions of the same library :-(I you are using Ikarus you should download the single file version of the library

```
$ wget http://www.phyast.pitt.edu/~micheles/scheme/sweet-macros.sls
```

and put it everywhere in you IKARUS_LIBRARY_PATH. If you are using PLT Scheme (you need a version of PLT newer than 4.0 for R6RS support) you must download the zip file version

```
$ wget http://www.phyast.pitt.edu/~micheles/scheme/sweet-macros.zip
$ unzip sweet-macros.zip
$ mv sweet-macros <your collects directory>
```

and install it in your `collects` directory, which on my machine is `$HOME/.plt-scheme/4.0/collects`.

Actually, the multifile version of the library works also with Ikarus if you have a recent enough version (right now I am using the trunk, version 0.0.3+, revision 1654). I have not tried the library on Larceny; I have tried it in Ypsilon Scheme which however has a small bug so that it does not run there (the bug is already fixed in the trunk). You should always keep in mind than R6RS implementations are pretty young and that implementors are still working to make them really compatible. I have also prepared an R5RS version which should work in Chicken Scheme, at least in the interpreter:

```
$ wget http://www.phyast.pitt.edu/~micheles/scheme/sweet-macros.scm
```

However I have developed and tested `sweet-macros` in Ikarus only (*caveat emptor*!). Still, since the title of this blog is *The Explorer*, I think it is fine if we deal with exploratory code.

You can check that the installation went well by importing the library:

```
$ ikarus
> (import (sweet-macros))
```

and by trying to define a macro.

9.4 An example: `multi-define`

Here is a `multi-define` binding construct which allows to define many identifiers at once:

```
(def-syntax (multi-define (name ...) (value ...))
  #'(begin (define name value) ...))
```

As you see, Scheme macros are based on pattern matching: we are giving instructions to the compiler, specifying how it must acts when it sees certain patterns. In our example, when the compiler sees a `multi-define` expression followed by two sequences with zero o more arguments, it must replace it with a `begin` expression containing a sequence of zero or more definitions. You can check that this is exactly what happens by means of `syntax-expand`:

```
> (syntax-expand (multi-define (a b) (1 2)))
(begin (define a 1) (define b 2))
```

Notice that `(multi-define () ())` is valid code expanding to a do-nothing `(begin)` expression; if you want to reject this corner case, you should write your macro as

```
(def-syntax (multi-define (name1 name2 ...) (value1 value2 ...))
  #'(begin (define name1 value1) (define name2 value2) ...))
```

so that `multi-define` requires one or more arguments. However, it is often useful to accept degenerate corner cases, because they may simplify automatic code generation (i.e. `multi-define` could appear in the expansion of another macro).

`multi-define` works as you would expect:

```
> (multi-define (a b) (1 2)) ; introduce the bindings a=1 and b=2
> a
1
> b
2
```

I have just scratched the surface of Scheme macros here: I leave the rest for the next episode, don't miss it!

CHAPTER

TEN

FEATURES OF (SWEET-)MACROS

Yet another episode fully devoted to macros. I will discuss introspection, guarded patterns, literal identifiers, and a couple of common beginner's mistakes.

10.1 syntax-match and introspection features of sweet-macros

In the last episode I have defined a very simple multi-define macro by using my own sweet-macros framework. I have also claimed that sweet macros provides introspection facilities, but I have not shown them. Here I will substain my claim.

First of all, let me show how you can get the patterns accepted by multi-define:

```
> (multi-define <patterns>)
((multi-define (name ...) (value ...)))
```

Since multi-define is a simple macro it accepts only a single pattern. However, it is possible to define macros with multiple patterns by relying on the second form of def-syntax, i.e.

```
(def-syntax name transformer)
```

where the transformer is a procedure which is typically built on top of syntax-match. For instance, suppose we wanted to extend multi-define to work also as a replacement of define, i.e. suppose we want to accept the pattern (multi-define name value) where name is an identifier. Here is how to do that by using syntax-match:

```
(def-syntax multi-define2
  (syntax-match ()
   (sub (ctx (name ...) (value ...))
     #'(begin (define name value) ...))
   (sub (ctx name value)
     #'(define name value))
  ))
```

syntax-match recognizes the literal identifier sub as an expected keyword when it appears in the right position, i.e. at the beginning of each clause. sub is there for two reasons:

1. in my opinion it makes the code more readable: you should read a clause `(sub pattern skeleton)` as "substitute a chunk of code matching the pattern with the code obtained by expanding the pattern variables inside the skeleton";
2. it makes `syntax-match` look different from `syntax-case` and `syntax-rules`, which is fine, since `syntax-match` *is* a little different from the Scheme standard macro systems.

The identifier `ctx` that you see as first element of each pattern denotes the context of the macro, a concept that I will explain in a future installment; you can use any valid identitier for the context, including the name of the macro itself - that is a common convention. If you are not interested in the context (which is the usual case) you can discard it and use the special identifier _ to make clear your intent.

I leave as an exercise to check that if you invert the order of the clauses the macro does not work: you must remember to put the most specific clause *first*.

In general you can get the source code for all the macros defined via `def-syntax` and `syntax-match`. For instance, the source code (of the transformer) of our original `multi-define` macro is the following:

```
> (multi-define <source>)
(syntax-match ()
  (sub (multi-define (name ...) (value ...))
    #'(begin (define name value) ...)))
```

As you see, for better readability `def-syntax` use the name of the macro for the context, but any name would do.

I have not explained everything there is to know about `syntax-match`, but we need to leave something out for the next episode, right?

10.2 A couple of common mistakes

If you try to write macros of your own, you will likely incur in mistakes. I think it is worth warning my readers about a couple of such common mistakes.

The first one is forgetting the `begin` for macros expanding to multiple expressions. For instance, you could be tempted to write `multi-define` as follows:

```
> (def-syntax (multi-define-wrong (name ...) (value ...))
   #'((define name value) ...))
```

If you try to use this macro, you will get an exception:

```
> (multi-define-wrong (a) (1))
Unhandled exception
Condition components:
  1. &who: define
  2. &message: "a definition was found where an expression was expected"
```

```
  3. &syntax:
      form: (define a 1)
      subform: #f
```

The problem is that Scheme interprets a pattern of the form `(func arg ...)` as a function application, but in this case `func` is the definition `(define a 1)` which is certainly not an function, it is not even an expression!

Actually, R6RS Scheme distinguishes definitions from expressions, a little bit like in other languages statements are distinguished from expressions, except that in Scheme there are no statements other than definitions. You will get exactly the same error if you try to print a definition `(display (define a 1))`: since a definition does not return anything, you cannot print it.

A second common mistake is to forget the sharp-quote `#'`. If you forget it - for instance if you write `(begin (define name value) ...)` instead of `#'(begin (define name value) ...)` - you will get a strange error message: *reference to pattern variable outside a syntax form*. To understand the message, you must understand what a *syntax form* is. That requires a rather detailed explanation that I will leave for a future episode.

For the moment, be content with a simplified explanation. A syntax form is a special type of quoted form: just as you write `'(some expression)` or `(quote (some expression))` to keep unevaluated a block of (valid or invalid) Scheme code, you can write `#'(some expression)` or `(syntax (some expression))` to denote a block of (valid or invalid) Scheme code which is intended to be used in a macro and contains pattern variables. Pattern variables must always be written inside a `syntax` expression, so that they can be replaced with their right values when the macro is expanded at compile time.

Note: R6RS Scheme requires the syntax `#'x` to be interpreted as a shortcut for `(syntax x)`; however there are R5RS implementation that do not allow the `#'x` syntax or use a different meaning for it. In particular, that was the case for old versions of Chicken Scheme. If you want to be fully portable you should use the extended form `(syntax x)`. However, all the code in this series is intended to work on R6RS Schemes, therefore I will always use the shortcut notation `#'` which in my opinion is *ways* more readable.

10.3 Guarded patterns

There are a few things I did not explain when introducing the `multi-define` macro. For instance, what happens if the number of the identifiers does not match the number of the values? Of course, you get an error:

```
> (multi-define (a b c) (1 2))
Unhandled exception
 Condition components:
   1. &assertion
   2. &who: ...
   3. &message: "length mismatch"
   4. &irritants: ((#<syntax 1> #<syntax 2>) (#<syntax a> #<syntax b> #<syntax c
```

The problem is that the error message is a bit scary, with all those `#<syntax >` things. How do we get an error message which is less scary to the newbie? Answer: by using the guarded patterns feature of `sweet-macros`!

Here is an example:

```
(def-syntax (multi-define (name ...) (value ...))  ; the pattern
  #'(begin (define name value) ...)                ; the skeleton
  (= (length #'(name ...)) (length #'(value ...))) ; the guard
   (syntax-violation 'multi-define
    "Names and values do not match"
    #'((name ...) (value ...))))
```

The line `(= (length #'(name ...)) (length #'(value ...)))` is the guard of the pattern `(multi-define (name ...) (value ...))`. The macro will expand the patterns in the guard into lists *at compile time*, then it will check that the number of names matches the number of values; if the check is satified then the skeleton is expanded, otherwise a `syntax-violation` is raised (i.e. a *compile time* exception) with a nice error message:

```
> (multi-define (a b c) (1 2))
Unhandled exception:
 Condition components:
   1. &who: multi-define
   2. &message: "Names and values do not match"
   3. &syntax:
       form: ((a b c) (1 2))
       subform: #f
```

Because of their working at compile time, guarded patterns are an ideal tool to check the consistency of our macros (remember: it is very important to check for errors as early as possible, and the earliest possible time is compile time).

10.4 Literal identifiers

Guarded patterns can also be (ab)used to recognize keyword-like identifiers in a macro. For instance, here is how you could implement the semantics of the `for` loop discussed in episode #8 with a macro (notice how all the funny characters `' , @ `` disappeared):

```
(def-syntax (for i from i0 to i1 action ...)
  #'(let loop ((i i0))
      (unless (>= i i1) action ... (loop (+ i 1))))
 (and (eq? (syntax->datum #'from) 'from) (eq? (syntax->datum #'to) 'to)))
```

Here the R6RS primitive `syntax->datum` is used to convert the syntax objects `#'from` and `#'to` into regular Scheme objects so that they can be compared for equality with the literal identifiers `'from` and `'to`.

You can check that the macro works by trying to use a wrong syntax. For install if you mispell `from` as `fro` you will get a syntax error *at compilation time*:

```
> (for i fro 1 to 5 (display i))
Unhandled exception:
 Condition components:
   1. &message: "invalid syntax"
   2. &syntax:
       form: (for i fro 1 to 5 (display i))
       subform: #f
```

Notice that this is an abuse of guarded patterns, since `syntax-match` provides a built-in mechanism just for that purpose. Moreover this macro is subject to the multiple evaluation problem which I will discuss in the next episode: thus I do not recommend it as an example of good style when writing macros. Still, I have written it here to compare it with the approach in episode #8: with this macro I have been able to extend the Scheme compiler for within, with just a few lines of code: that is much simpler than writing an external compiler as a preprocessor, as I planned to do before.

As I said, `syntax-match` has the built-in capability of recognizing literal identifiers in the patterns as if they were keywords. This is what the empty parenthesis are for. If you write `(syntax-match (lit ...) clause ...)` the identifiers listed in `(lit ...)` will be treated as literal identifiers in the macro scope. Literal identifiers can be used to enhance readability, or to define complex macros. For instance our `for` macro can be written without need for guarded patterns as:

```
(def-syntax for
  (syntax-match (from to)
    (sub (for i from i0 to i1 action ...)
      #'(let loop ((i i0))
          (unless (>= i i1) action ... (loop (+ i 1)))))))
```

You can even introspect the literal identifiers recognized by `syntax-match`:

```
> (for <literals>)
(from to)
```

Let me close this paragraph by suggesting an exercise in macrology. Try to implement a Python-like `for` loop working as in the following examples:

```
> (for x in '(1 2 3)
    (display x))
123
> (for (x y) in '((a b) (A B) (1 2))
    (display x) (display y))
abAB12
```

Clearly it should work for a generic number of arguments and `in` should be treated as a literal identifier. I will give the solution in episode 12, so you will have some time to play. Have fun!

CHAPTER

ELEVEN

THE MULTIPLE EVALUATION PROBLEM (AND EASY-TEST)

In this episode I will discuss the multiple evaluation issue, then I will show how macros can improve performance. Finally, I will give a practical example of how macros can be used to define a unit test framework.

11.1 The problem of multiple evaluation

In episode #10 I gave an example of a macro implementing a C-like `for` loop and I said that it was suffering from the problem of multiple evaluation. Here I explain what the problem is and how to cure it. In order to understand the issue, you must always remember that macros *expand* code at compile time, but they not *evaluate* it: this means that pattern variables do *not* correspond to evaluated expression, as ordinary variables, but they correspond to expressions to be evaluated later, at runtime.

As a consequence, it is easy to write macros that evaluate expressions more times than needed. For instance, consider the following simplified version of a C-like `for` loop, with a runtime type check:

```
(def-syntax (for i start end body ...)
  #'(begin
     (assert (and (number? start) (number? end))); type-check
     (let loop ((i start))
       (unless (>= i end) body ... (loop (+ i 1))))))
```

Suppose the variable `end` to be determined dynamically with a computation:

```
> (define (get-end)
  (printf "computing the value of end\n")
 3)
```

Then our naive macro suffers from the multiple evaluation problem:

```
> (for i 0 (get-end) 'do-nothing)
computing the value of end
computing the value of end
computing the value of end
computing the value of end
computing the value of end
```

As you see, in this example `end` is recomputed 5 times! The reason is clear if you look at the expansion of the macro:

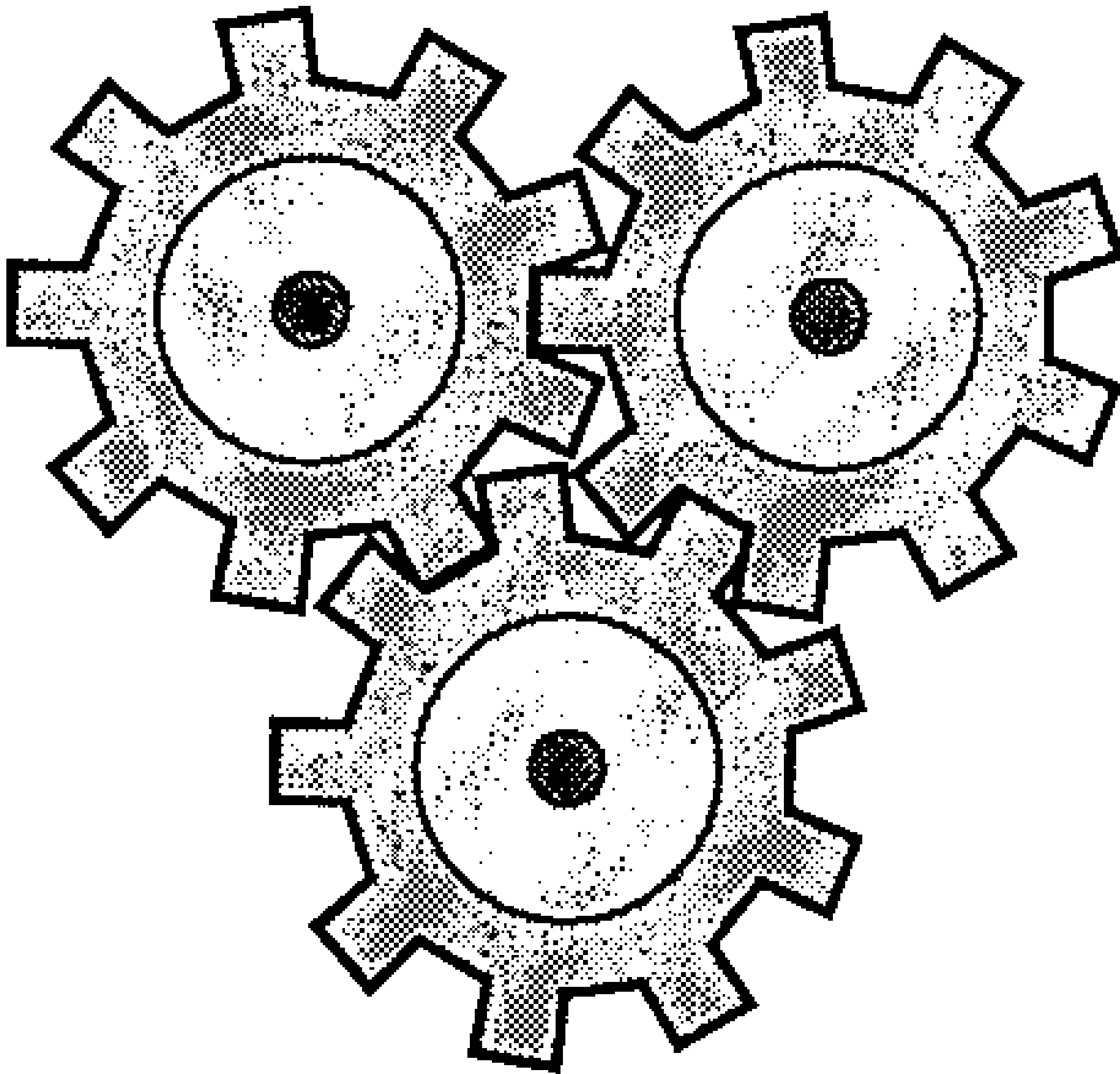
```
> (syntax-expand (for i 0 (get-end) 'do-nothing))
(begin
 (assert (and (number? 0) (number? (get-end))))
 (let loop ((i 0))
   (unless (>= i (get-end)) 'do-nothing (loop (+ i 1)))))
```

The `get-end` function is called once in the assertion and four times in the loop; that is inefficient and can have very dramatic effects if the function has side effects. The solution is to save the value of `end` (and we could do the same for the value of `start`, which is computed twice) in a variable:

```
(def-syntax (for i start end body ...)
   #'(let ((s start) (e end))
      (assert (and (number? s) (number? e)))
      (let loop ((i s))
        (unless (>= i e) body ... (loop (+ i 1))))))
```

Now `get-end` is called only once and we are all happy :-)

As an exercise, you could extend `for` to accept a generic step. You can find the solution in the Italian original version of this article, which is quite different and uses `syntax-rules`.



11.2 Taking advantage of multiple evaluation

Sometimes we can make good use of the multiple evaluation "feature". For instance, let me consided again the higher order function `call` I introduced in episode #5, when discussing benchmark. That function has an issue: it is called at each iteration in the inner loop and therefore it wastes time. However, it is possible to replace the higher order function with a macro, therefore avoiding the cost of a function call. Here is the code for a `repeat` macro doing the job of `call`:

```
(library (repeat-macro)
(export repeat)
(import (rnrs) (sweet-macros))

(def-syntax (repeat n body body* ...)
  #'(let loop ((i 0))
```

```
      (when (< i n) body body* ... (loop (+ 1 i)))))
)
```

`repeat` expands into a loop and therefore the body is evaluated `n` times, which is exactly what we need for a benchmark. To check that the macro is effectively more efficient, I did measure the time spent in summing 1+1 ten million of times:

```
(import (rnrs) (repeat-macro) (repeat) (only (ikarus) time))
(define n (string->number (car (reverse (command-line)))))
(time (call 10000000 + 1 n))
(time (repeat 10000000 (+ 1 n)))
```

I took the number `n` from the command line arguments in order to fool the compiler: if I hard coded `(+ 1 1)`, the compiler would replace it with 2 at compilation time, therefore not performing the computation! (In the original version of this episode I made that mistake, thanks to Aziz Ghuloum for pointing it out). The output of the script is the following:

```
$ scheme-script repeat-benchmark.ss 1
running stats for (call 10000000 + 1 n):
    no collections
    396 ms elapsed cpu time, including 0 ms collecting
    394 ms elapsed real time, including 0 ms collecting
    32 bytes allocated
running stats for (repeat 10000000 (+ 1 n)):
    no collections
    40 ms elapsed cpu time, including 0 ms collecting
    40 ms elapsed real time, including 0 ms collecting
    0 bytes allocated
```

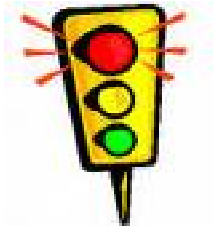
As you see, avoiding the function call makes a lot of difference (the benchmark is 10 times faster!) since the great majority of the time is wasted in calling the benchmarking function and not in the real addition. Here the improvement is spectacular since summing two integers is a very fast operation: replacing `call` with `repeat` in the benchmark factorial does not make a big difference instead.

11.3 A micro-framework for unit tests

It is time to give a more practical example of Scheme macros. In this paragraph, I will define a very simple unit test framework called `easy-test`.

Clearly, there are already unit test frameworks available for Scheme, including two SRFIs (64 and 78); my interests here is not in the testing framework, it is in the implementation, which makes a pedagogical exercise in macrology.

The source code takes just a page:



```
#!r6rs
(library (aps easy-test)
(export catch-error test run-tests runner run)
(import (rnrs) (aps compat) (sweet-macros))

;; helper macro
(def-syntax (catch-error body body* ...)
  #'(let*
        ((error-message #f)
         (result
          (guard (err ;; as a side effect, set! the error message if any
                         ((or (assertion-violation? err) (error? err)
                                (undefined-violation? err))
                          (set! error-message (condition-message err))))
                        body body* ...)))
       (if error-message error-message
           (error 'catch-error "Expected error, got none!"
                        '(body body* ...) '=> result))))

;; test macro
(def-syntax (test description expr expected)
  #'(lambda (cmd)
       (case cmd
         ((descr) description)
         ((values) (list expected expr 'expr))
         ((run) (equal? expr expected))
         (else (error 'test "Invalid command" cmd)))))

;; four helper functions
(define (print-nothing descr expected evalued-expr expr)
  (display ""))

(define (print-dot descr expected evalued-expr expr)
  (display "."))

(define (print-msg descr expected evalued-expr expr)
  (printf "\n~s failed\nExpected ~s, got ~s\nExpression was ~a\n"
          descr expected evalued-expr expr))

(define (print-stats successes failures)
  (define total (+ successes failures))
  (printf "\nRun ~a tests. ~a passed, ~a failed\n" total successes failures))
```

```
;; full runner
(define (run-tests print-success print-failure . tests)
  (let loop ((tests tests) (success 0) (failure 0))
    (if (null? tests)
        (list success failure)
        (let* ((test1 (car tests))
                 (descr (test1 'descr)) (vals (test1 'values)))
           (if (test1 'run)
               (begin; the test succeeded
                 (apply print-success descr vals)
                 (loop (cdr tests) (+ 1 success) failure))
               (begin; the test failed
                 (apply print-failure descr vals)
                 (loop (cdr tests) success (+ 1 failure))))))))

;; runner factory
(define (runner print-success print-failure print-stats)
  (lambda tests
    (define succ-fail (apply run-tests print-success print-failure tests))
    (apply print-stats succ-fail)))

;; default runner
(define run (runner print-dot print-msg print-stats))

)
```

The core of the framework is the `test` macro, which is a bit different from the macros we have defined until now. The reason why the `test` macro is different is that it expands into a `lambda`-expression and therefore the arguments of the macro are evaluated only when the `lambda` function is called and not at definition time. In other words, we are using a pattern of *delayed evaluation* here. This is important, since we want to distinguish the definition of a test from its execution. For instance, let me define a trivial test:

```
> (import (easy-test))
> (define test1 (test "1+1=2" (+ 1 1) 2))
```

The first argument of the macro is a string describing the test, which is nice to have in the error message for failed tests; the second argument of the macro is the expression to check and the third argument is the expected result.

Macro application results in a function which is able to respond to the commands `'descr` (returning the description string), `'values` (returning a list with the quoted input expression and the quoted expected output) and `'run` (returning the result of the test, as a boolean flag). This is implemented via the case expression in the `test` macro:

```
(case cmd
  ((descr) description)
  ((values) '(expr  expected))
  ((run) (equal? expr expected))
  (else (error 'test "Invalid command" cmd)))
```

Here is how it works in our example:

```
> (test1 'descr)
"1+1=2"
> (test1 'values)
((+ 1 1) 2)
> (test1 'run) ; the test passed
#t
```

The framework provides three predefined functions `print-nothing`, `print-msg` and `print-dot` to print feedback about how the tests are going; moreover, it is possible to define custom reporting functions. A reporting function is simply a function with three arguments `(descr expr expected)` where `descr` is a string with the description of the test, `expr` is the expression to be checked and `expected` is the expected result. You can specify the reporting functions to use by defining a test runner as in this example:

```
> (define run-quiet (runner print-nothing print-msg))
> (run-quiet
    (test "1+1=2" (+ 1 1) 2)
    (test "2*1=2" (* 2 1) 2)
    (test "2+2=3" (+ 2 2) 3))
'2+2=3' failed. Expected 3, got 4
(2 1)
```

The runner returns a list with the number of passed tests and failed tests (in our case `'(2 1)`).

It is also possible to use the default runner (`run`): the framework will use the default reporting functions, i.e. `print-dot` for successful tests and `print-msg` for failed tests.

CHAPTER

TWELVE

ARE MACROS REALLY USEFUL?

In this episode I discuss the utility of macros for enterprise programmers.

12.1 Are macros "just syntactic sugar"?

There is a serious problem when teaching macros to beginners: the real power of macros is only seen when solving difficult problems, but you cannot use those problems as teaching examples. As a consequence, virtually all beginner's introductions to macros are dumbed down: usually they just show a few trivial examples about how to modify the Scheme syntax to resemble some

other language. I did the same too. This way of teaching macros has two negative effects:

1. beginners are naturally inclined to pervert the language instead of learning it;
2. beginners can easily dismiss macros as mere syntactic sugar.

The first effect is the most dangerous: the fact that you can implement a C-like `for` loop in Scheme does not mean that you should use it! I strongly believe that learning a language means learning its idioms: learning a new language means that you must change the way you think when writing code. In particular, in Scheme, you must get used to recursion and accumulators, not to imperative loops, there is no other way around.

Actually, there are cases where perverting the language may have business sense. For instance, suppose you are translating a library from another language with a `for` loop to Scheme. If you want to spend a minimal effort in the translation and if for any reason you want to stay close to the original implementation (for instance, for simplifying maintenance), then it makes sense to leverage on the macro facility and to add the `for` loop to the language syntax.

The problem is that it is very easy to abuse the mechanism. Generally speaking, the adaptibility of the Scheme language is a double-edged sword. There is no doubts that it increases the programmer expressivity, but it can also make programs more difficult to read. The language allow you to invent your own idioms that nobody else uses, but perhaps this is not such a good idea if you care about other people reading your code. For this reason macros in the Python community have always been viewed with suspicion: I am also pretty confident that they will never enter in the language.

The second effect (dismissing macros) is less serious: lots of people understimate macros as mere syntactic sugar, by forgetting that all Turing complete languages differ solely on syntactic sugar. Moreover, thinking too much about the syntactic sugar aspect make them blind to others and more important aspects of macros: in particular, the fact that macros are actually *compilers*.

That means that you can implement with macros both *compile time checks* (as I have stressed in episode #10, when talking about guarded patterns) and *compile time computations* (I have not discussed this point yet) with substantial benefits for what concerns both the reliability and the performance of your programs. In episode #11 I have already shown how you can use macros to avoid expensive function calls in benchmarks and the example generalizes to any other situations.

In general, since macros allows you to customize the evaluation mechanism of the language, you can do with macros things which are impossible without them: such an example is the `test` macro discussed in episode #11. I strongly suggest you to read the third comment to that episode, whereas it is argued that it is impossible to implement an equivalent functionality in Python.

So, you should not underestimate the power of macros; on the other hand, you should also not underestimate the complexity of macros. Recently I have started a thread on comp.lang.scheme with 180+ messages about the issues I have encountered when porting my `sweet-macros` library between different Scheme implementations, and the thread ended up discussing a lot of hairy points about macros (expand-time vs run-time, multiple instantiation of modules, separate compilation, and all that).

12.2 About the usefulness of macros for application programmers

I am not an advocate of macros for enterprise programming. Actually, even ignoring the issue with macros, I cannot advocate Scheme for enterprise programming because of the lack of a standard library worth of its name. This was more of an issue with R5RS Scheme, but it is still a problem since Scheme has an extremely small standard library and no concept of *batteries included* à la Python. As a consequence, everybody has to invent its own collections of utilities, each collection a little bit different from the other.

For instance, when I started learning Scheme I wrote a lot of utilities; later on, I discovered that I could find the same utilites, under different names and slightly different signatures, in various Scheme frameworks. This never happened to me in Python to the same extent, since the standard library is already coding in an uniform way most of the useful idioms, so that everybody uses the library and there is less need to reinvent the wheel.

On the other hand, I am not a macro aficionado like Paul Graham, who says:

> *When there are patterns in source code, the response should not be to enshrine them in a list of "best practices," or to find an IDE that can generate them. Patterns in your code mean you are doing something wrong. You should write the macro that will generate them and call that instead.*

I think Graham is right in the first part of its analysis, but not in the conclusion. I agree that patterns are a code smell and I think that they denote a lack in the language or in its standard library. On the other hand, the real solution for the enterprise programmer is not to write her own macro which nobody knows, but to have the feature included in the language by an authoritative source (for instance Guido van Rossum in the case of Python) so that *all* users of the language get the benefit in an uniform way.

This happened recently in Python, with the ternary operator, with the `try .. except .. finally` statement, with the *with statement*, with extended generators and in many other cases. The Scheme way in which everybody writes his own language makes sense for the academic researcher, for the solitary hacker, or for very small team of programmers, but not for the enterprise.

Notice that I am not talking about specialized newly invented constructs: I am talking about *patterns* and by definition, according to the GoF, a pattern cannot be new, it must be a tried and tested solution to a common problem. If something is so common and well known to be a pattern, it also deserves to be in the standard library of the language, or in a standard framework. This works well for scripting languages, which have a fast evolution, and less well in languages designed by committee, where you can wait years and years for any modernization of the language/library (we all know Paul Graham is coming from Common Lisp, so his position is understandable).

In my opinion - and your are free to disagree of course - the enterprise programmer is much better served by a language without macros but with a very complete library where all useful constructs have been codified already. After all, 99.9% of the times the enterprise programmer has to do with already solved problems: it is not by chance that frameworks are so used in the enterprise world. Notice that by "enterprise programmer" I mean the framework *user*, not the

framework *writer*.

Take my case for instance: at work I am doing some Web programming, and I just use one of the major Python web frameworks (there already too many of them!); I do quite of lot of interaction with databases, and I just use the standard or *de facto* standard drivers/libraries provided for the task at hand; I also do some scripting task: then I use the standard library a lot. For all the task I routinely perform at my day job macros would not help me a bit: after all Python has already many solutions to avoid boilerplate (decorators, metaclasses, etc.) and the need for macros is not felt. I admit that some times I wished for new constructs in Python: but usually it was just a matter of time and patience to get them in the language and while waiting I could always solve my problems anyway, maybe in a less elegant way.

There are good use cases for macros, but there also plenty of workarounds for the average application programmer.

For instance, a case where one could argue for macros, is when there are performance issues, since macros are able to lift computations from runtime to compile time, and they can be used for code optimization. However, even without macros, there is plenty of room for optimization in the scripting language world, which typically involve interfacing with C/C++.

There are also various standard techniques for *code generation* in C, whereas C++ has the infamous *templates*: while those are solutions very much inferior to Scheme macros, they also have the enormous advantage of working with an enterprise-familiar technology, and you certainly cannot say that for Scheme.

The other good use for macros is to implement compile time checks: compile time checks are a good thing, but in practice people have learned to live without them by relying on a good unit test coverage, which is needed anyway.

On the other hand, one should not underestimate the downsides of macros. Evaluation of code defined inside of the macro body at compile time or suspension of evaluation therein leads often to bugs that are hard to track. The behaviour of the code is generally not easy to understand and debugging macros is no fun.

That should explain why the current situation about Scheme in the enterprise world is as it is. It is also true that the enterprise programmer's job is sometimes quite boring, and you can risk brain atrophy, whereas for sure you will not incur in this risk if you keep reading my *Adventures* ;)

You may look at this series as a cure against senility!

12.3 Appendix: a Pythonic `for` loop

In this appendix I will give the solution to the exercise suggested at the end of episode #10, i.e. implementing a Python-like `for` loop.

First of all, let me notice that Scheme already has the functionality of Python `for` loop (at least for lists) via the for-each construct:

```
> (for-each (lambda (x y) (display x) (display y)) '(a x 1) '(b y 2))
abxy12
```

The problem is that the syntax looks quite different from Python:

```
>>> for (x, y) in (("a", "b"), ("x", "y"), (1, 2)):
...     sys.stdout.write(x); sys.stdout.write(y)
```

One problem is that the order of the list is completely different, but this is easy to fix with a `transpose` function:

```
(define (transpose llist) ; llist is a list of lists
  (if (and (list? llist) (for-all list? llist))
      (apply map list llist)
      (error 'transpose "Not a list of lists" llist)))
```

(if you have read carefully episode #8 you will notice the similarity between `transpose` and `zip`). `transpose` works as follows:

```
> (transpose '((a b) (x y) (1 2)))
((a x 1) (b y 2))))
```

Then there is the issue of hiding the `lambda` form, but this is an easy job for a macro:

```
(def-syntax for
  (syntax-match (in)
   (sub (for el in lst do something ...)
        #'(for-each (lambda (el) do something ...) lst)
        (identifier? #'el))
   (sub (for (el ...) in lst do something ...)
        #'(apply for-each (lambda (el ...) do something ...) (transpose lst))
        (for-all identifier? #'(el ...))
        (syntax-violation 'for "Non identifier" #'(el ...)
                          (remp identifier? #'(el ...))))
   ))
```

The important thing to notice in this implementation is the usage of a guard with an `else` clause: that allows to introduce two different behaviours for the macro at the same time. If the pattern variable `el` is an identifier, then `for` is converted into a simple `for-each`:

```
> (for x in '(1 2 3) (display x))
123
```

On the other hand, if the pattern variable `el` is a list of identifiers and `lst` is a list of lists, then the macro also reorganizes the arguments of the underlying `for-each` expression, so that `for` works as Python's `for`:

```
> (for (x y) in '((a b) (x y) (1 2)) (display x) (display y))
abxy12
```

CHAPTER

THIRTEEN

MICRO-INTRODUCTION TO FUNCTIONAL PROGRAMMING

The first installment of my long-awaited third cycle of a Pythonista's adventures with Scheme is devoted to Scheme's functional aspects.

13.1 A minimal introduction to functional programming

I assume you already know what *pure functional language* means: a language is purely functional if variables cannot be re-assigned, data structures cannot be modified, and side effects are excluded.

Of course, it is impossible to program with a pure functional language, since input and output are based on side effects and you cannot have a sensible program without input and output. However, a practical functional language can still be as pure as possible if it is able to confine the non-functional aspects to input and output only, in a controlled way.

The purest functional language out there is probably Haskell; on a lower level of purity we find the languages of the ML family (SML, OCAML, F#, ...); on a lower level there is Scheme. Python and Common Lisp are at the same level, both below Scheme.

I would not consider Python and Common Lisp as truly functional languages: they are just imperative languages with some support for functional programming (I mean constructs such as `map`, `filter`, `reduce`, list comprehension, generators, et cetera). However, there is a large gap between an imperative language with some support for functional programming and a true functional language.

True functional languages have strong support for recursion (tail call optimization), for higher order functions and for pattern matching; moreover, true functional languages are based on immutable data structures. Scheme is somewhat less functional than SML and Haskell, since Scheme lists are mutable, currying is not supported by the base syntax of the language, (it can be implemented via macros, of course) and generally speaking one uses higher order functions less.

While not pure, Scheme can be quite functional if you avoid rebinding and you restrict yourself to functional data structures, and it allows many typical idioms of functional programming which have no counterpart in imperative programming. We already saw a common trick in

episode #5, i.e. the accumulator trick, which is a way to avoid mutation in loops by using recursion. In this episode and the next ones we will show many others.

Figure 13.1: The purity of functional languages

13.2 Functional data structures: pairs and lists

Historically, the basic data types of Lisp languages (pairs and lists) have always been mutable. In all Lisp dialects (including Scheme) it has always been possible to modify the `car` and the `cdr` of a pair freely. The situation changed with the R6RS report: nowadays the imperative procedures `set-car!` and `set-cdr!` have been removed from the rnrs environment.

Actually, it is still possible to mutate pairs, but only by extending the rnrs environment, i.e. by importing the `(rnrs mutable-pairs)` extension, which is part of the standard library but not of the core language. This is a clear indication of the fact that the functional paradigm is somewhat a recommended paradigm in Scheme, even if it is not enforced (in strongly functional languages, such as SML and Haskell, lists are immutable and there is no other option).

I should notice that the requirement of importing `(rnrs mutable-pairs)` is only valid for scripts and libraries: the behavior of the REPL is unspecified in the R6RS document (actually the R6RS forbids a REPL but every R6RS implementation provides a REPL with some different semantics) and implementations are free to import or not to import `mutable-pairs` in the REPL. The REPL of Ikarus imports `mutable-pairs` by default, so you have at your

disposal `set-car!` and `set-cdr!`, but this is an implementation specific choice; other implementations can behave differently from Ikarus at the REPL, and in general they do.

Excluding code typed at the REPL, in principle the compiler could use immutability optimizations for library code not importing the `(rnrs mutable-pairs)` extension. In practice, a Scheme compiler cannot perform the optimizations based on the assumption of truly immutable pairs, because the current standard says that `cons` must allocate a new pair and cannot re-use a previously created one (i.e. `(eq? (cons x y) (cons x y))` is always false for any value of `x` and `y`: even if the two conses have the same value, they correspond to different objects).

If pairs were really immutable, a compiler could use the same object for equivalent pairs, i.e. `'(x . y)` could be the same as `(cons x y)` and a compiler could cache that value: with the current standard that cannot never happen. The point is well explained in a recent thread in comp.lang.scheme (Really immutable pairs).

Really immutable pairs (and thus lists) have lots of advantages: I would welcome them in the standard, but I am not sure if that will happen, since there is a potential compatibility breaking problem. I can only hope for the best. Immutability has advantages from the point of view of efficiency and makes the life of language implementors easier, but those are not really important point for an application programmer.

The important point for the mere mortals is that programs based on immutable data structures becomes easier to understand and to debug. For instance, consider a routine taking a list in input and suppose that the content of the list is not what you would expect. If the list is mutable you potentially have to wonder about your whole code base, since everything could have mutated the list before reaching the routine you are interested in. If the list if immutable, you are sure that the bug must be in the procedure which created the list, and in no other place.

Moreover, functional structures avoid whole classes of bugs (I am sure every Pythonista has found some issue with lists being mutable, especially when used as default arguments) especially in the hairy situations of multithreaded code. In my *Adventures* I will never rely on the ability to mutate pairs, and I will use pairs as functional data structures.

13.3 Functional update

There is apparently an issue with immutable data structures: in many imperative programs one needs to modify the data: but how can you update an immutable object?

The answer is actually pretty simple: you don't. Since you cannot mutate an immutable object, the only option is to create a brand new object with a different content from the original one. This mechanism is called *functional update*: for instance, it is easy to define two functional procedures `set-car` and `set-cdr` performing the job of `set-car!` and `set-cdr!` but without mutation:

```
(define (set-car pair value)
  (cons value (cdr pair)))

> (set-car (cons 1 2) 3)
```

```
(3 . 2)

(define (set-cdr pair value)
  (cons (car pair) value))

> (set-cdr (cons 1 2) 3)
(1 . 3)
```

What if you want to (functionally) update the *n*-th value of a list? The trick is to use recursion:

```
(define (list-set n lst value)
  (if (zero? n)
     (set-car lst value)
     (cons (car lst) (list-set (- n 1) (cdr lst) value))))

> (list-set 2 '(a b c d) 'X)
(a b X d)
```

Notice that `list-set` is nicer than its imperative counterpart:

```
(define (list-set! n lst value)
  (set-car! (list-tail lst n) value) lst)

> (define ls '(a b c d))
> (list-set! 2 ls 'X)
> ls
(a b X d)
```

Notice the use of the R6RS procedure `(list-tail lst n)` which returns the tail of `lst` starting from the n-th element. The indexing starts from zero, as usual (for instance `(list-tail '(a b c d) 2)` is the list `(c d)`). The important bit to understand how `list-set!` works is that `list-tail` returns the tail, and *not a copy* of it: by mutating the tail you are actually mutating the original list. This is clearly quite risky.

The R6RS standard does not provide primitives for functional update out of the box (`list-set!` is not in the standard since it is of very little utility: if you want to be able to modify the n-th element of a sequence, you are much better off by using a vector and not a list). However, it does provide primitives to remove elements for a list functionally:

```
> (remp even? '(3 1 4 1 5 9 2 6 5)); complementary of filter
  (3 1 1 5 9 5)

> (remove 1 '(3 1 4 1 5 9 2 6 5))
(3 4 5 9 2 6 5)

> (remv 1 '(3 1 4 1 5 9 2 6 5))
(3 4 5 9 2 6 5)

> (remq 'foo '(bar foo baz))
(bar baz)
```

(you can find other list utilities in R6RS standard library).

One would expect functional update to be much slower than imperative update, because of the need of creating potentially long lists to modify just a single element. However, this is not necessarily true. It all depends on how smart your compiler is. A smart compiler can internally use mutation (at the machine level), so in principle it could be as fast as imperative code. The advantage is that mutation is managed by the compiler, not by the programmer, who can think purely in terms of immutable data structures.

It is true that compilers for functional languages are still slower than C compilers in average, but they are not so bad. In most situations the slowdown is acceptable and in particular situations compilers for functional languages can be faster than a C compiler. Moreover, they allow for memory optimizations. For instance, you can memoize an immutable list, whereas you cannot memoize a mutable one.

Since I come from a Python background I do not care much about performance and optimizations, but I care a lot about maintainability of programs and bug reduction, as well as about conceptual cleanness. Moreover, this series has also some pedagogical intention, therefore I will prefer functional solutions over imperative ones here, in order to show new ways of doing old things.

CHAPTER

FOURTEEN

CURRYING, PARTIAL APPLICATION, AND FOLD

Everything you ever wanted to know about currying, partial application, higher order functions and related topics.

14.1 Higher order functions and curried functions

A language has support for first class functions if it is possible to use a function as a regular value, i.e. if it is possible to pass a function to another function, or return it from a function. In a language with first class functions, it is therefore possible to define the concept of higher order function is: a function which accepts in input or returns in output (or both) another function.

Various imperative languages have support for higher order functions: all the scripting languages, the latest version of C#, `Scala`, and a few others. Still, functional languages have a better support and higher order functions are used in those language much more that in imperative languages. This is especially true for languages such as ML and Haskell, which support curried functions out of the box: in such languages all functions are really unary functions (i.e. they accept a single argument) and functions of *n* arguments are actually unary functions returning closures. In Scheme this behavior can be emulated with macros. Here is an example of how one could define curried functions in Scheme:

```
(def-syntax curried-lambda
  (syntax-match ()
    (sub (curried-lambda () b b* ...)
         #'(begin b b* ...))
    (sub (curried-lambda (x x* ...) b b* ...)
         #'(lambda (x) (curried-lambda (x* ...) b b* ...)))
    ))

(def-syntax (define/curried (f x ...) b b* ...)
  #'(define f (curried-lambda (x ...) b b* ...)))
```

`define/curried` defines a function with (apparently) *n* arguments as an unary function returning a closure, i.e. a function with (apparently) *n-1* arguments which in turns is an unary

function returning a closure with *n-2* arguments and so on, until it returns an unary function. For instance, the following `add` function

```
(define/curried (add x y) (+ x y))
```

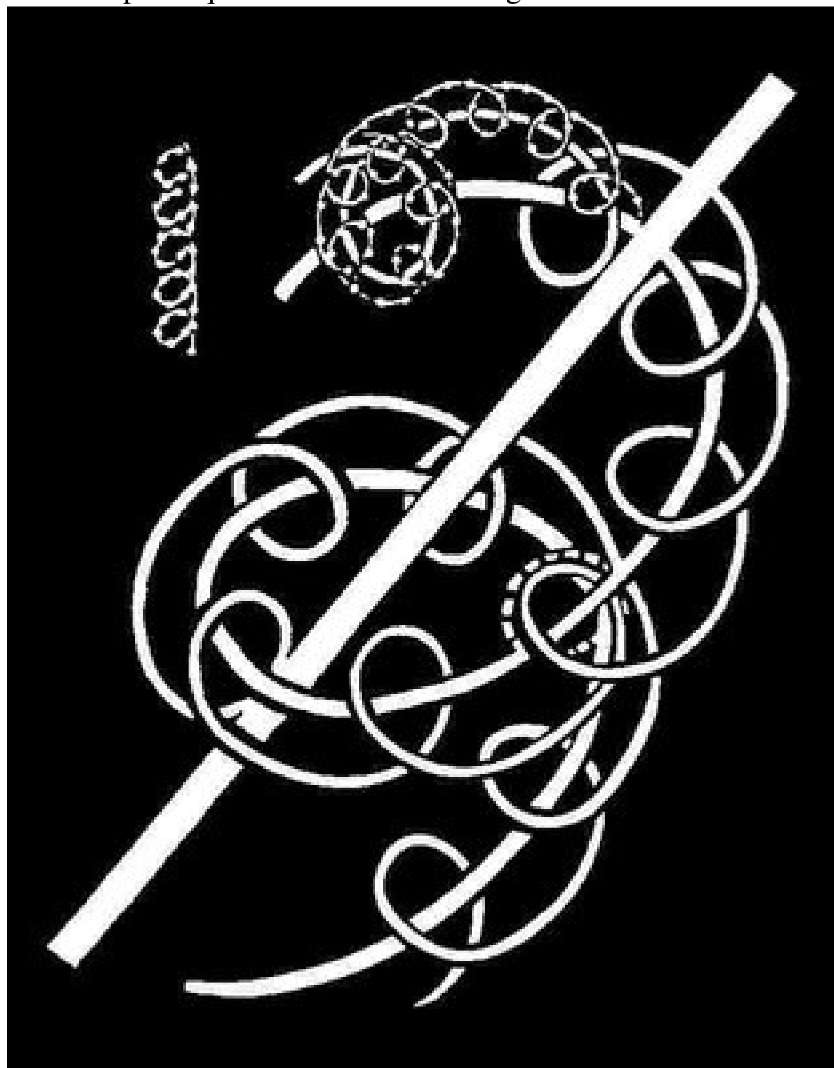
apparently has two arguments, but actually it is an unary function returning an unary closure:

```
> (add 1)
#<procedure>
> ((add 1) 2)
3
```

You can see how the macro works by using `syntax-expand`:

```
> (syntax-expand (curried-lambda (x y) (+ x y)))
(lambda (x) (curried-lambda (y) (+ x y)))
```

The internal `curried-lambda` has a single argument in this case and thus expands to a regular lambda function, but you can see that in general you will have a tower of nested lambdas, which dept is equal to the number of arguments.



Whereas it is possible define curried functions in Scheme, usually this is not very convenient, unless you are trying to emulate ML or Haskell idioms. Out of the box, Scheme supports functions with multiple arguments in a traditional fashion, i.e. the same as in Python: thus, the most convenient construct is not currying, but *partial application*. The Pythonistas here will certainly think of `functools.partial`, an utility which was added to the standard library starting from Python 2.5. Schemers have something similar (but of course better) in the form of SRFI-26, i.e. the `cut` and `cute` macros by Al Petrofsky.

14.2 Partial application: cut and cute

Instead of spending too many words, let me show an example of how partial function application works both in Python and in Scheme.

Here is the Python version:

```
>>> from functools import partial
>>> from operator import add
>>> add1 = partial(add, 1)
>>> add1(2)
3
```

and here is the Scheme version:

```
> (import (srfi-26)); assuming it is available in your implementation
> (define add1 (cut + 1 <>))
> (add1 2)
3
```

In Python, `partial(add, 1)` returns an unary callable object that adds 1 to its argument; in Scheme, `(cut + 1 <>)` returns an unary function that does the same. The Scheme version is better, since the arguments of the resulting functions are immediately vas visible as slots (i.e. the <> symbol). For instance

```
> (define greetings (cut string-append "hello " <> " and " <>))
```

has two slots and therefore is a function of two arguments:

```
> (greetings "Michele" "Mario")
"hello Michele and Mario"
```

It is also possible to define a variable number of arguments by using the rest-slot symbol `<...>`:

```
> (define greetings (cut string-append "hello " <> " and " <...>))
> (display (greetings "Michele" "Mario" "\n"))
hello Michele and Mario
```

We can even use a slot for the function: for instance, the higher order function `apply` could be implemented as `(cut <> <...>)`.

Moreover, there is a `cute` macro which acts exactly as `cut`, with a single difference: the arguments in `cute` are evalued only once (the `e` stands for *evalued*), whereas `cut` is not safe against multiple evaluation. In particular, if you define

```
> (define add-result (cute + (long-computation) <>))
```

then `add-result` performs the long computation only once, at definition time, and not every time it is called. For more details I refer you the SRFI-26 specification.

14.3 fold-left and fold-right

A couple of commonly used higher order functions in Scheme and other functional languages are `fold-left` and `fold-right`. They entered in the R6RS standard, but they are also available from SRFI-1, therefore you can leverage on them even if you are using an R5RS Scheme.

`fold-left` and `fold-right` will remind Pythonistas of `reduce`, which is also a folding function. However, it is well known that Guido dislikes it and nowadays `reduce` is no more a builtin (in Python 3.0); it is still available in the `functools` module, though. For some reason (probabily the order of the arguments which I cannot remember) I cannot use `reduce` in Python, whereas I have less problems with `fold-left` e `fold-right` in Scheme and other functional languages.

`fold-left` and `fold-right` have a nearly identical API: both allow to traverse a list by accumulating values and by returning at the end the final accumulator. For instance, if you want to sum the values of list, here is an idiomatic solution in Scheme (another one is `(apply + numbers-list)`:

```
> (fold-left + 0 '(1 2 3)); sum all elements starting from 0; fold-right works t
6
```

In general, the function in `fold-left` takes *N + 1* arguments, where *N* is the number of lists you are looping over (usually *N = 1*) and the leftmost argument is the accumulator. The same is true for `fold-right`, but then the rightmost argument is the accumulator.

Notice that `fold-left` is quite different from `fold-right`, since they work in opposite order:

```
> (fold-left (lambda (acc el) (cons el acc)) '() '(1 2 3))
(3 2 1)

> (fold-right (lambda (el acc) (cons el acc)) '() '(1 2 3))
(1 2 3)
```

In the first case `fold-left` loops from left to right (the element 1 is the first to be consed, the element 2 is the second to be consed, and the element 3 is the last to be consed, so that

the final result is `(cons 3 (cons 2 (cons 1 '())))` i.e. `(3 2 1))` whereas in the second case `fold-right` loops from right to left.

In order to give an example of use, here is how you could define a flattening procedure by using `fold-right`:

```
(define (flatten lst)
  (fold-right
   (lambda (x a)
     (if (list? x) (append (flatten x) a) (cons x a))) '() lst))
```

You can check that it works with a few tests:

```
(test "flatten null"
      (flatten '())
      '())
(test "flatten plain"
      (flatten '(a b c))
      '(a b c))
(test "flatten nested"
      (flatten '((a b) (c (d e) f)))
      '(a b c d e f))
```

Here is another example, a function to remove duplicates from a list:

```
;; ex: (remove-dupl = '(1 2 3 1 5 2 4)) => (1 2 3 5 4)
(define (remove-dupl eq? lst)
  (reverse
   (fold-left
    (lambda (acc el)
      (if (exists (cut eq? <> el) acc); duplicate
          acc
          (cons el acc)))
    '() lst)))
```

Notice the use of `cut` to define an unary function `(cut eq? <> el)` which checks if its argument is equal - according to the provided equality function - to a given element `el`. `exists` is one of the list processing utilities standardized by the R6RS document. Here is a test:

```
(test "remove-dupl"
      (remove-dupl equal? '(1 #f 2 #f 3))
      '(1 #f 2 3))
```

Having first class functions in a language means much more than having map, filter or fold. Perhaps in the future I will add another episode about advanced usage of functions, such a parsing combinators or formatting combinators; for the moment what I said here should be enough, and the next episode will be devoted to another typical feature of functional languages: *pattern matching*.

CHAPTER

FIFTEEN

LIST DESTRUCTURING

In my Adventures I have referred many times to pattern matching, but only in the context of compile time pattern matching in macros. There is another form of pattern matching, which is quite common in Scheme and in other functional languages: run time pattern matching. This episode will shed some light on the technique.

15.1 About pattern matching

It is impossible to overvalue the importance of pattern matching which is in my opinion one of the most important concepts in programming. Unfortunately, this technique is only available in very high level programming languages and therefore it is usually unknown to the average programmer.

I saw pattern matching for the first time in '95, when using Mathematica for High Energy Physics symbolic computations, which is a very specific usage indeed. Nowadays, however, the trend toward higher and higher abstraction is influencing all programmming languages and I am pretty sure than soon or later pattern matching will enter in mainstream languages.

For the moment, you can find it in functional languages and, in a poor man form, in certain scripting languages. It should be noticed that common functional languages such as SML, OCaml, F#, Haskell (or even Scala) only have run time pattern matching, since they lack macros. In Scheme instead compile time pattern matching is somewhat preferred: you use it to manipulate compile-time lists which are actually blocks of code wrapped in macros.

Runtime pattern matching is used to manipulate lists (or other data structures) which are only know at runtime: in particular, they could be user input. Compile time pattern matching can be used to implement a compiler; run time pattern matching to implement an interpreter.

In this episode I will discuss only a poor man form of runtime pattern matching, list destructuring, i.e. the ability to match (nested) lists with a single predefined pattern. This ability is akin to what `def-syntax` can do at compile time. Full runtime pattern matching is able to manage a whole set of patterns, akin to what `syntax-match` can do at compile time.

The poor form of pattern matching is called tuple unpacking in Python (note for lispers: you would call it destructuring bind). For instance, you can write:

```
>>> (a, (b, [c, d])) = (1, [2, iter((3, 4))])
>>> (a, b, c, d)
(1, 2, 3, 4)
```

Tuple unpacking works at any level of nesting and for any kind of iterable, therefore it is pretty powerful. Moreover, tuple unpacking is even more powerful in Python 3.0, where it is possible to split an iterable into its head (`car`) and tail (`cdr`):

```
>>> head, *tail=(i for i in (1,2,3))
>>> (head, tail)
(1, [2, 3])
```

I have noticed in episode #5 that the star syntax in Python is similar to the dot syntax in Scheme, when used in the signature of functions with a variable number of arguments (variadic functions); the syntactic extension in Python 3.0 makes the similarity stronger.

The main difference between Python and Scheme is that Scheme pattern matching is not polymorphic, i.e. you cannot match with the same pattern a list and a vector or an equivalent iterable. You must use different patterns, or esplicitely convert the types.

There are plenty of libraries for full runtime pattern matching: one of the most common is the match library by Andrew Wright, which is available practically for all Scheme implementations. In Chicken Scheme match is actually built-in in the core language:

```
$ csi
CHICKEN
Version 2.732 - macosx-unix-gnu-x86     [ manyargs dload ptables applyhook cross
(c)2000-2007 Felix L. Winkelmann        compiled 2007-11-01 on michele-mac.local

#;1> (match-define (head . tail) '(1 2 3))
#;2> (list head tail)
(1 (2 3))
```

Recently the implementation of match has been rejuvenated by Alex Shinn, who fixed a few bugs and reimplemented everything in terms of `syntax-rules` macros, whereas the original used `define-macro`: this modern implementation is also available as an R6RS library, thanks to Derick Eddington and you can download it for here, if you want to use this matcher with Ikarus.

Studying the documentation of match is certainly a good use of your time, and a recommended reading; on the other hand, writing your own matcher relying on Scheme macros is even more interesting. In the next paragraph I will implement a `let+` macro with the full power of tuple unpacking, and in future episodes I will implement a fully fledged list matcher.

15.2 A list destructuring binding form (let+)

This paragraph will use a test-first approach, and will begin with a specification of how `let+` is intended to work by means of tests. I am using here the minimal testing framework I have introduced in episode #11. Here are the tests:

```
(test "no args"
 (let+ 1); no bindings; return 1
 1)

(test "name value"
 (let+ (x 1) x); locally bind the name x to the value 1 and return it
 1)

(test "one arg"
 (let+ ((x) '(1)) x); locally bind the name x to the value 1 and return it
 1)

(test "two args"
 (let+ ((x y) (list 1 2)) (list x y)); locally bind the names x and y
 '(1 2))

(test "pair"
  (let+ ((x . y) '(1 2)) y)
  '(2))

(test "nested"
  (let+ ((x (y z)) '(1 (2 3))) (list x y z)); bind x, y and z
  '(1 2 3))
```

Here is an implementation satisfying those tests:

```
(def-syntax let+
  (syntax-match ()
    (sub (let+ expr)
         #'expr)
    (sub (let+ (() lst) expr)
         #'(if (null? lst) expr
                (apply error 'let+ "Too many elements" lst)))
    (sub (let+ ((arg1 arg2 ... . rest) lst) expr)
         #'(let ((ls lst))
              (if (null? ls)
                  (apply error 'let+ "Missing arguments" '(arg1 arg2 ...))
                  (let+ (arg1 (car ls))
                    (let+ ((arg2 ... . rest) (cdr ls)) expr)))))
    (sub (let+ (name value) expr)
         #'(let ((name value)) expr)
         (identifier? #'name)
         (syntax-violation 'let+ "Argument is not an identifier" #'name))
    (sub (let+ (name value) (n v) ... expr)
         #'(let+ (name value) (let+ (n v) ... expr)))
    ))
```

It is not difficult to understand how the macro works by performing a few experiments `syntax-expand`; for instance, `let+` with a single argument expands as follows:

```
> (syntax-expand (let+ ((x) '(1)) x))
(let ((ls '(1)))
  (if (null? ls)
     (apply error 'let+ "Not enough elements" '(x))
     (let+ (x (car ls)) (let+ (() (cdr ls)) x))))
```

whereas `let+` with a required argument and a variadic list of arguments expands as follows:

```
> (syntax-expand (let+ ((x . rest) '(1)) (cons x rest)))
(let ((ls '(1)))
  (if (null? ls)
      (apply error 'let+ "Not enough elements" '(x))
      (let+ (x (car ls)) (let+ (rest (cdr ls)) (cons x rest)))))
```

Notice that in this case the template `(arg2 ... . rest)` has been replaced by `rest`, since there are no arguments. This is the magic of the dots!

Finally, let us see what happens when we try to match a too short list:

```
> (let+ ((x y) '(1)) x)
Unhandled exception
 Condition components:
   1. &error
   2. &who: let+
   3. &message: "Missing arguments"
   4. &irritants: (y)
```

or a too long list:

```
> (let+ ((x y) '(1 2 3)) x)
Unhandled exception
 Condition components:
   1. &error
   2. &who: let+
   3. &message: "Too many elements"
   4. &irritants: (3)
```

In the first case there an argument `(y)` in excess, not matched by any element; in the second case, there is an element `(3)` in excess, not matched by any argument. The implementation also checks (at compile time) that the passed arguments are valid identifiers:

```
> (let+ ((x y 3) '(1 2 3)) x)
Unhandled exception
 Condition components:
   1. &who: let+
   2. &message: "Argument is not an identifier"
   3. &syntax:
       form: 3
       subform: #f
   4. &trace: #<syntax 3>
```

As I said, Scheme pattern matching is not polymorphic: you cannot exchange a vector for a list of viceversa:

```
> (let+ ((x (y z)) (list 1 (vector 2 3))) (list x y z))
Unhandled exception:
 Condition components:
   1. &assertion
   2. &who: car
   3. &message: "argument does not have required pair structure"
   4. &irritants: (#(2 3))
```

The error message is clear, we also know that a `car` was involved, but unfortunately, it does not give much information about *where* exactly the error happened :-(I was serious at the end of episode #12, when I said that debugging macros is no fun: the problem is that the errors happen in expanded code which is invisible to the programmer.

In the next episode I will give a few examples of usage of `let+` which will make clear why it is so useful. See you next time!

CHAPTER

SIXTEEN

MULTIPLE VALUES (AND OPT-LAMBDA)

This episode is a direct continuation of latest issue: it gives example of use of the destructuring bind form let+ introduced there. I also discuss multiple values, unary functions and functions with optional arguments.

16.1 list destructuring versus let-values

There is a feature of Scheme that I never liked, i.e. the fact that functions (more in general continuations) can return multiple values. Multiple values are a relatively late addition to Scheme - they entered in the standard with the R5RS report - and there has always been some opposition against them (see for instance this old post by Jeffrey Susskind). I personally see multiple values as a wart of Scheme, a useless complication motivated by premature optimization concerns.

It is possible to define functions returning multiple values as follows:

```
> (define (return-three-values)
    (values 1 2 3))
> (return-three-values)
1
2
3
```

In order to receive the values a special syntax is needed, and you cannot do things like the following:

```
> (apply + (return-three-values))
Unhandled exception
 Condition components:
   1. &assertion
   2. &who: apply
   3. &message: "incorrect number of values returned to single value context"
   4. &irritants: ((1 2 3))
```

Instead, you are forced to use `let-values` or other constructs which are able to accept values:

```
> (let-values (((x y z) (return-three-values))) (+ x y z))
6
```

In this series I will never use functions returning multiple values, except the ones in the Scheme standard library (this is why I am forced to talk about `let-values`). Instead of using multiple values, I will return a list of values and I will destructure it with `let+`. For instance, I will write

```
> (let+ ((a b) (list 1 2)) (cons a b))
(1 . 2)
```

instead of

```
> (let-values (((a b) (values 1 2))) (cons a b))
(1 . 2)
```

`let+` is more elegant and more general than `let-values`: everything `let-values` can do, `let+` can do too. `let+` can even faster - in some implementations and in some cases. Here is a benchmark in Ikarus Scheme:

```
running stats for (repeat 10000000 (let-values (((x y z) (values 1 2 3))) 'dummy
    no collections
    276 ms elapsed cpu time, including 0 ms collecting
    277 ms elapsed real time, including 0 ms collecting
    0 bytes allocated

running stats for (repeat 10000000 (let+ ((x y z) (list 1 2 3)) 'dummy)):
    58 collections
    211 ms elapsed cpu time, including 42 ms collecting
    213 ms elapsed real time, including 43 ms collecting
    240016384 bytes allocated
```

As you see, `let+` takes only 211 ms to unpack a list of three elements ten million times; `let-values` would take 276 ms instead. On the other hand, `let+` involves garbage collection (in our example 24 bytes are allocate per cycle, and thats means 240 million of bytes) and depending on the situations and the implementation this may cause a serious slowdown. You may find much better benchmarks than mine in this thread on comp.lang.scheme and you will see that you can get any kind of results. `let-values` seems to be slow in Ikarus and in Ypsilon with the default optimization options; it can be faster in other implementations, or in the same implementations with different options or in different releases.

However, those are implementation details. The important thing in my view is the conceptual relevance of a language construct. Conceptually I think the introduction of multiple values in Scheme was a mistake, since it added nothing that could not be done better with containers. I think functions should *always* return a single value, possibly a composite one (a list, a vector, or anything else). Actually, I am even more radical than that and I think that functions should take a *single value*, as in SML and Haskell.

16.2 Variadic functions from unary functions

If you have a minimalistic mindset (as I have) you will recognize that multiple argument functions are useless since they can be implemented as unary functions performing destructuring. Here is a simple implementation of the idea:

```
(def-syntax (fn (arg ... . rest) body body* ...)
  #'(lambda (x)
      (let+ ((arg ... . rest) x)
        (begin body body* ...))))

(def-syntax (define/fn (name arg ... . rest) body body* ...)
  #'(define name (fn (arg ... . rest) body body* ...)))
```

Here are a few examples of usage:

```
> (define/fn (double x) (* 2 x))
> (double '(1))
2

> (define/fn (sum . vals) (apply + vals))
> (sum '(1 2 3))
6

> (define/fn (sum-2D (x1 y1) (x2 y2)) (list (+ x1 x2) (+ y1 y2)))
> (sum-2D '((1 2)(3 4)))
(4 6)
```

All the functions defined via `define/fn` take a single argument, a list, which is then destructured according to the declared structure. `double` expects a list with a single element named `x`; `sum` expects a list with a variable number of elements `val`; `sum-2D` expects a two-element lists made of two-element lists named `(x1 y1)` and `(x2 y2)` respectively. You can easily imagine more complex examples with deeply nested lists.

It is interesting to notice that Python has the list destructuring/tuple unpacking functionality built-in:

```
>>> def sum2D((x1, y1), (x2, y2)):
...     return x1 + x2, y1 + y2
...
>>> sum2D((1,2),(3,4))
(4, 6)
```

This is valid Python code in all versions of Python before Python 3.0. However, in Python 3X this functionality has been removed for lack of use (*sic*).

The advantage of unary functions is that they are easier to compose, and many functional patterns (including currying described in episode #14) becomes possible. However, Scheme is not ML or Haskell, so let us accept functions with multiple arguments and let us take advantage of them to implement optional arguments. This is the subject of the next paragraph.

16.3 Further examples of destructuring: opt-lambda

A weekness of standard Scheme is the lack of functions with default arguments and keyword arguments. In practice, this is a minor weakness since there many libraries implementing the functionality, although in different ways, as usual. I recommend you to look at SRFI-88 and SRFI-89 for more context. Here I will implement equivalent functionality from scratch, as yet another exercise to show the power of `let+`. Let me start from an example, to make clear the intended functionality. Let me define a function `f` with optional arguments as follows:

```
(define/opt (f x y (opt (u 1) (v 2)) . rest)
  (printf "Required: ~a Optional: ~a Other: ~a\n"
    (list x y) (list u v) rest))
```

Here `x` and `y` are required arguments, `u` and `v` are optional arguments and `rest` are variadic arguments. If you do not provide an optional argument, its default value is be used instead, and `f` behaves as follows:

```
> (f 'a 'b 'c 'd 'e 'f)
Required: (a b) Optional: (c d) Other: (e f)
> (f 'a 'b 'c)
Required: (a b) Optional: (c 2) Other: ()
> (f 'a 'b)
Required: (a b) Optional: (1 2) Other: ()
> (f 'a)
Unhandled exception
 Condition components:
   1. &assertion
   2. &who: apply
   3. &message: "incorrect number of arguments"
   4. &irritants: (#<procedure f> 1)
```

It is clear that in order to implement the functionality the trick is to override the defaults of the optional argument with the passed arguments, if any. To this aim we will need the following helper function:

```
;;(override-with '(a b) '(1 2 3)) => '(a b 3)
(define (override-with winner loser)
  (let ((w (length winner)) (l (length loser)))
    (if (>= w l)
        winner ; winner completely overrides loser
        (append winner (list-tail loser w)))))
```

(we introduced the `list-tail` function in episode #13). At this point it is easy to define an `opt-lambda` macro doing the job:

```
(def-syntax opt-lambda
  (syntax-match (opt)
    (sub (opt-lambda (r1 ... (opt (o1 d1) ...) . rest) body1 body2 ...)
    #'(lambda (r1 ... . args)
        (let+ ((o1 ... . rest) (override-with args (list d1 ...)))
              (begin body1 body2 ...))))))
```

`define/opt` is just sugar over `opt-lambda`:

```
(def-syntax (define/opt (name . args) body1 body2 ...)
  #'(define name (opt-lambda args body1 body2 ...)))
```

I should notice that strictly speaking you do not need a full restructuring form to implement `opt-lambda`: since `override-with-args` returns a flat list, a form which is able to destructure flat list is enough. You could implement it easily enough:

```
(def-syntax (let- (name ... . rest) lst expr)
  #'(apply (lambda (name ... . rest) expr) lst))
```

However `let+` subsumes the functionality of `let-` and I do not see the point of introducing yet another binding form, except for sake of the exercise. Strangely enough, `let-` looks even slower than `let+` in Ikarus:

```
running stats for (repeat 10000000 (let- (x y z) (list 1 2 3) 'dummy)):
    58 collections
    324 ms elapsed cpu time, including 16 ms collecting
    324 ms elapsed real time, including 22 ms collecting
    240004096 bytes allocated
```

But please don't trust benchmarks! ;)

CHAPTER

SEVENTEEN

LIST COMPREHENSION

This episodes explains how to implement a functional list comprehension syntax in Scheme. The difference with Python list comprehension is also explained. Moreover, I have decided to distribute the code create for this series as a library: http://www.phyast.pitt.edu/~micheles/scheme/aps.zip

17.1 The APS library

The R6RS standard provides a few list utilities; the SRFI-1 provides a few others. Nevertheless the offering is incomplete: in particular a list comprehension syntax is missing. Therefore I have decided to distribute a library providing list comprehension and more. Such library will be useful for future episodes of my *Adventures*, in particular for part IV, about advanced macro programming. After all, macro programming is nothing else than manipulation of code seen as a nested list of expressions.

With a remarkable lack of fantasy, I have decided to call the library `list-utils` and to put it in a package called `aps` (`aps` of course stands for *Adventures of a Pythonista in Schemeland* and not for *American Physical Society* ;). In this way I will be contributing to the entropy and I will be littering the world with yet another version of utilities that I would rather not write, but

this cannot be helped :-(

For your convenience, I have added in the library the Python-style utilities `range`, `zip`, `transpose`, `enumerate` I did discuss in episodes 7 and 8, as well as the `let+` list destructuring macro I introduced in episode 15. I have also added the reference implementation of SRFI-26 i.e. the `cut` and `cute` macros described in episode _14. Moreover, the `aps` package contains the testing framework discussed in episode 11, renamed as `(aps easy-test)` and slightly improved (the improvement consists in the addition of `catch-error` macro, which captures the error message). Finally, the `aps` library includes a more recent version of `sweet-macros` than the one I discussed in episode 9, so you should replace the old one if you have it.

You can download the package from here: http://www.phyast.pitt.edu/~micheles/scheme/aps.zip

Just unzip the archive and put the files somewhere in your path:

```
$ cd <DIRECTORY-IN-YOUR-SCHEME-PATH>
$ unzip aps.zip
inflating: sweet-macros.sls
inflating: aps/cut.sls
inflating: aps/easy-test.sls
inflating: aps/list-utils.sls
...
```

You can test the library as follows:

```
$ ikarus --r6rs-script aps/test-all.ss
.........................
Run 25 tests. 25 passed, 0 failed
```

Currently all the tests pass with the latest development version of Ikarus. They also pass with the latest development version of Ypsilon and with PLT Scheme version 4, except for the test `zip-with-error`:

```
(test "zip-with-error"
      (catch-error (zip '(a b c) '(1 2)))
      "length mismatch")
```

However, this is an expected failure, since the error messages are different between Ikarus, Ypsilon and PLT Scheme. Clearly, checking for an implementation-dependent error message is a bad idea and I could have thought of a better test, but I am lazy; moreover, I do not want to discuss the error management mechanism in Scheme right now, since it is quite advanced and it is best deferred to future episodes.

Larceny Scheme is not supported since it does not support the `.IMPL.sls` convention. When it does, it could be supported as well, expecially if I get some help from my readers.

If you are wondering about the so-called `.IMPL.sls` convention, let me explain that it is a non-standard convention to enable portability about different R6RS implementations. In particular the `aps` library contains three modules `compat.ikarus.sls`, `compat.mzscheme.sls` and `compat.ypsilon.sls` following the convention. When I write `(import (aps compat))` in Ikarus, the file `compat.ikarus.sls` is read; when

I import `(aps compat)` in PLT, the file `compat.mzscheme.sls` is read; and finally for Ypsilon the file `compat.ypsilon.sls` is read. This mechanism allows for writing compatibility wrappers; for instance, here is the content of `compat.mzscheme.sls`:

```
#!r6rs
(library (aps compat)
(export printf format gensym pretty-print)
(import (rnrs) (only (scheme) printf format gensym pretty-print)))
```

Basically all decent Scheme implementations provide `printf`, `format`, `gensym` and `pretty-print` functionality, usually with these names too, but since they are not standard (which is absurd IMO) one is forced to recur to do-nothing compatibility libraries, which just import the functionality from the implementation-specific module and re-export it :-(

You can try the `(aps list-utils)` library as follows:

```
> (import (aps list-utils))
> (enumerate '(a b c))
((0 a) (1 b) (2 c))
```

Notice that you should consider the `aps` libraries as experimental and subject to changes, at least until I finish the series, in an indetermined and far away future ;)

17.2 Implementing list comprehension

The most important facility in the `(aps list-utils)` library is a syntax for list comprehension. Perhaps list comprehension is not the greatest discovery in computer science since sliced bread, but I find them enormously more readable than `map` and `filter`, which I use only in the simplest case. Nowadays, a lot of languages offer a syntax for list comprehension, starting from Haskell to Python, Javascript and C#.

Scheme does not provide a list comprehension syntax out of the box, but it is a simple exercise in macrology to implement them. Actually there are many available implementations of list comprehension in Scheme. There is even an SRFI (SRFI-42 Eager Comprehensions) which however I do not like at all since it provides too much functionality and an ugly syntax.

Therefore, here I will pursue a different approach to list comprehension, which is shamelessly copied from the work of Phil Bewig, with minor simplifications, and the usage of `let+` instead of regular `let`.

Here is the implementation

```
(def-syntax list-of-aux
  (syntax-match (in is)

    (sub (list-of-aux expr acc)
     #'(cons expr acc))

    (sub (list-of-aux expr acc (var in lst) rest ...)
     #'(let loop ((ls lst))
         (if (null? ls) acc
             (let+ (var (car ls))
               (list-of-aux expr (loop (cdr ls)) rest ...)))))

    (sub (list-of-aux expr acc (var is exp) rest ...)
     #'(let+ (var exp) (list-of-aux expr acc rest ...)))

    (sub (list-of-aux expr acc pred? rest ...)
     #'(if pred? (list-of-aux expr acc rest ...) acc))
  ))
```

```
(def-syntax (list-of expr rest ...)
  #'(list-of-aux expr '() rest ...))
```

We see here the usage of an helper macro `list-of-aux` and the usage of an accumulator `acc` to collect the arguments of the macro. You may understand how it works by judicious use of `syntax-expand`; for instance `(list-of-aux (* 2 x) '() (x in (range 3)))` expands into

```
(let loop ((ls (range 3)))
  (if (null? ls)
      '()
      (let+ (x (car ls))
        (list-of-aux (* 2 x) (loop (cdr ls))))))
```

which builds the list `(0 2 4)`, since `list-of-aux` expands to the list constructor `cons`. Here are a few other test cases you may play with:

```
(test "double comprehension"
      (list-of (list x y) (x in '(a b c)) (y in '(1 2)))
      '((a 1) (a 2) (b 1) (b 2) (c 1) (c 2)))

(test "double comprehension with constraint"
      (list-of (list x y) (x in (range 3)) (y in (range 3)) (= x y))
      '((0 0) (1 1) (2 2)))

(test "comprehension plus destructuring"
      (list-of (+ x y) ((x y) in '((1 2)(3 4))))
      '(3 7))
```

The macro is able to define nested list comprehensions at any level of nesting; the rightmost variables corresponds to the inner loops and its is even possible to implement constraints and destructuring: basically, we have the same power of Python list comprehensions, except that that the objects in the `in` clause must be true lists, whereas in Python they can be generic iterables (including infinite ones).

17.3 A tricky point

On the surface, the `list-of` macro looks the same as Python list comprehension; however, there a few subtle differences under the hood, since the loop variables are treated differently. You can see the different once you consider a list comprehension containing closures. In Scheme a list comprehensions of closures works as you would expect:

```
> (define three-thunks (list-of (lambda () i) (i in '(0 1 2))))
> (list-of (t) (t in three-thunks))
(0 1 2)
```

In Python instead you get a surprising result (unless you really know how the `for` loop work):

```
>>> three_thunks = [(lambda : i) for i in [0, 1, 2]]
>>> [f() for f in three_thunks]
[2, 2, 2]
```

The reason is that Python is not really a functional language, so that the `for` loop works by *mutating* the loop variable `i`: since the thunk is called after the end at the loop, it sees the latest value of `i`, i.e. 2. The same is true in Common Lisp if you use the LOOP macro. In Scheme instead (and in Haskell, the language that invented list comprehension) there is *no mutation* of the loop variable: at each iteration a new fresh `i` is created. You can emulate in Python what Scheme does for free by using two lambdas:

```
>>> three_thunks = [(lambda x : (lambda : x))(i) for i in [0, 1, 2]]
>>> [f() for f in three_thunks]
[0, 1, 2]
```

(another way of course is to use the well know default argument trick, `lambda i=i: i`, but that is not a direct translation of how Scheme of Haskell work by introducing a new scope at each iteration).

On the other hand, Python wins on Scheme for what concern polymorphism: in Python is it possible to iterate on any iterable without any effort, whereas in Scheme you need to specify the data structure you are iterating over. For instance, if you want to iterate on vectors you need to define a `vector-of` macro for vector comprehension; if you want to interate on hash table you need to define an hash-table comprehension macro `hash-table-of`, and so on. Alternatively, you must convert you data structure into a list and use `list-of`. This is annoying. In Python on the contrary there is a common protocol for all iterable objects so that the same `for` syntax can be used everywhere.

The list comprehension defined here only works for finite iterables; Python however has also a generator comprehension that works on potentially infinite iterables. Scheme too allows to define infinite iterables, the so called *streams*, which however are a functional data structure quite different from Python generators, which are imperative. Discussing streams will fill the next episode. For the moment, have patience!

CHAPTER

EIGHTEEN

STREAMS

This episode is about streams, a typical data structure of functional languages. The differences between functional streams and imperative iterators are discussed. En passant, I give a solution of the classic eight queens problem.

18.1 The eight queens puzzle

Before starting the analysis of streams, I want to close the discussion about list comprehension. Last week I had no time to discuss one of the conveniences of the `list-of` macro, i.e. the ability to define internal variables with a `(name is value)` syntax. To give an example of that, I have decided to show you a solution of the infamous eight-queens puzzle that you will find in all the theoretical textbooks about programming.

In my opinion the eight queens puzzle is not so interesting, however, if you want to study Scheme, you will find this kind of academical examples everywhere, so I made my concession to the tradition. In particular, the official document about streams in R6RS Scheme, i.e. SRFI-41, contains a solution of the eight queens puzzle by using the same algorithm I am presenting here, but with streams instead of lists. You may want to compare the list solution to the stream solution.

Figure 18.1: Animation taken from Wikipedia

The algorithm is based on a clever trick which is quite common in the mathematical sciences: to introduce an additional degrees of freedom which apparently makes the problem harder, but actually gives us a fast lane towards the solution. Here the clever idea is to change the question and to considered not a single square chessboard, but a family of rectangular chessboards with *n* rows and *N* columns (with $n<=N$ and $N=8$). Of course we are interested in the $n=8$ solution; however, keeping *n* generic helps, since an easy solution can be found for small *n* (in particular for $n=1$) and we can figure out a recursive algorithm to build the $n+1$ solution starting from the $n=1$ solution, until we reach $n=8$.

Let us express a solution as a list of column positions for the queens, indexed by row. We will enumerate rows and columns starting from zero, as usual. The case $n=1$ (putting a queen on a *1x8* chessboard) has 8 solutions, expressible as the list of lists `'((0)(1)(2)(3)(4)(5)(6)(7))` - the first (and only) queen will be at row 0 and

columns 0, 1, 2, 3, 4, 5, 6 or 7. If there are two queens (*n=2*) one has more solutions; for instance the first queen (i.e. the one at row 0) could be at column 0 and the second queen (i.e. the one at row 1) at column 2, and a solution is `(0 2)`. The solutions for the *n*-queens problem are found by looking at the possible new configurations, starting from the solutions of the *n-1*-queens problem and by discarding the forbidden ones.

A configuration is forbidden if two queens are on the same column (by construction they cannot be in the same row, since they are indexed by row) or on the same diagonal. The condition *being on the same diagonal* translates into *the difference between the row coordinates is the same as the difference between the column coordinates, in absolute value*. Here is the condition expressed in Scheme and making use of `list-of` and of the `is` syntax, where `new-row` and `new-col` is the tentative position of the *n*-th queen and `safe-config` is a solution of the *n-1*-queen problem:

```
(define (all-true lst) ;; a Python-like all, true if all elements are true
  (for-all (lambda (x) x) lst))

(define (safe-queen? new-row new-col safe-config)
  ;; same-col? and same-diag? are boolean inner variables
  (all-true (list-of (not (or same-col? same-diag?))
                    ((row col) in (enumerate safe-config))
                    (same-col? is (= col new-col))
                    (same-diag? is (= (abs (- col new-col)) (abs (- row new-row))))
                    )))
```

`safe-queen?` checks that the new configuration is safe by looking at all the queens already placed. We can find all the solutions with a recursive function:

```
(define (queens n N)
  (if (zero? n) '(())
      (list-of (append safe-config (list col))
                    (n-1 is (- n 1)); inner variable
                    (safe-config in (queens n-1 N))
                    (col in (range N))
                    (safe-queen? n-1 col safe-config)
                    )))
```

In particular we can check that the *n=8* problem has 92 solutions:

```
> (length (queens 8 8))
92
```

I refer you to Wikipedia for nice drawings of the solutions.

18.2 Iterators and streams

Python programmers are well acquainted with generators and iterators, and they know everything about lazyness. In particular they know that the Python iterator

```
>>> it123 = iter(range(1, 4))
```

is left unevaluated until its elements are requested. However, the Python way is only superficially similar to the truly functional way, found in Haskell or in Scheme. Actually, when Python copies from functional languages, it does so in an imperative way. Here the iterator `it123` is an object with an internal state; there is a `next` method which allows to change the state. In particular, if you call `.next()` twice, the same iterator returns different values:

```
>>> it123.next()
1
>>> it123.next()
2
```

Thus, Python iterators *are not functional*. Functional languages such as Scheme, ML and Haskell have no imperative iterators: they have *streams* instead. Ikarus comes with a built-in stream library, so that I can give a concrete example right now (of course you can use streams in other implementations simply by using the reference implementation described in SRFI-41). Here is how to define a stream on the numbers 1,2,3:

```
> (import (streams))
> (define str123 (stream-range 1 4))
```

There is no equivalent of the `next` method for streams, since there is no concept of internal state of a stream. However, there is a `stream-car` procedure which takes the first element of a stream, and a `stream-cdr` procedure returning another stream which lacks the first element. Both procedures are functional, i.e. they act without mutating the original stream object in any way. In particular, if you apply `stream-car` twice, you get always the same result:

```
> (stream-car str123)
1
> (stream-car str123)
1
```

In Python, looping on an iterator exhausts it, and running twice the same loop can have unexpected results:

```
>>> chars = iter('abc')
>>> for c in chars: print c,
...
a b c
>>> for c in chars: print c,
...
```

The first time the loop prints "a b c", but the second time it does not print anything. In a functional language the same code must have the same effect, it cannot depend from the inner state of the stream. Actually, this is what happens:

```
> (define chars (stream #\a #\b #\c))
> (stream-for-each display chars)
abc> (stream-for-each display chars)
abc>
```

`stream-for-each` is an utility from SRFI-41, with obvious meaning. Actually SRFI-41 offers a series of convenient features. The most useful one is stream comprehension, which is very similar to list comprehension. Since I copied the list comprehensions syntax from the work of Phil Bewig, which is the author of the stream library, it is not surprising that the syntax looks the same. The difference between list comprehensions and stream comprehension is that stream comprehension is lazy and can be infinite. This is similar to Python generator expressions (*genexps*). For instance, in Python we can express the infinite set of the even number as a genexp

```
>>> import itertools
>>> even = (i for i in itertools.count(0) if i % 2 == 0)
```

whereas in Scheme we can express it as a stream:

```
> (define even (stream-of i (i in (stream-from 0)) (zero? (modulo i 2))))
```

However the Scheme stream is immutable, whereas the Python genexp is not. It is possible to loop over a stream with `stream-for-each`, `stream-map` and `stream-fold`; such higher order functions work as they counterparts for lists, but they return streams. There is also a `stream-let` syntax, which is stream version of named let, useful when applying the accumulator pattern to streams, and a function `stream->list` with does the obvious.

I am not explaining all the fine details, since the documentations of the SRFI is pretty good and exhaustive. As I anticipated, there is also a solution of the eight queen problem using streams that you may look at. The difference between the stream solution and the list comprehension solution is that the first one is lazy, i.e. you get one solution at the time, on demand, whereas the second one is eager: it computes all the solutions in a single bunch, and returns them together.

18.3 Lazyness is a virtue

The basic feature of streams, apart from immutability, is true lazyness. Streams are truly lazy since they perform work only when forced - i.e. only when an element is explicitly requested - and even there if they had already accomplished a task they do not perform it again - i.e. they *memoize* the elements already computed (and this is *not* the case for Python iterators). An example should make these two points clear. Let us define a `work` procedure which protests when called:

```
> (define (work i)
    (display "Life is hard!\n") i)
```

The protest is expressed an a side effect; other than that, the function, does not perform too much, since it just returns the parameter got in input, but, you know, there is no limit to lazyness!

Now let me define a stream which invokes the function `work` three times:

```
> (define work-work-work (stream-of (work i) (i in (stream-range 1 4))))
```

Since the stream is lazy, it does not perform any work at definition time. It starts working only when elements are expressly required:

```
> (stream->list work-work-work)
Life is hard!
Life is hard!
Life is hard!
(1 2 3)
```

Now the values 1, 2, 3 have been memoized: if we try to loop again on the stream, it will return the precomputed values:

```
> (stream->list work-work-work)
(1 2 3)
```

This shows clearly that the function `work` is not called twice. It is also clear that, had `work` some useful side effect (such as writing a log message) then using a stream would not be a good idea, since you could loose some message. Streams are a functional data structure and it is good practice to use only pure functions with them, i.e. functions without side effects. Moreover, I should also notice that the memoization property implies that a stream can take an unbound amount of memory, whereas an imperative iterator has no such issue.

I could say more. In particular, there are lots of caveats about streams, which are explained in detail in the SRFI-41 documentation (so many caveats that I personally do not feel confident with streams). I am also sure that Pythonistas would be even more interested in true generator-expressions and generators, which can be definite in Scheme by using continuations. However, investigating that direction will astray us away from our path. The intention of this third cycle of *Adventures* was just to give a feeling of what does it mean to be a true functional language, versus being an imperative language with a few functional-looking constructs.

With this episode this cycle of our *Adventures* ends, but a new one will begin shortly. Stay tuned!

CHAPTER NINETEEN

THE R6RS MODULE SYSTEM

For nearly 30 years Scheme lived without a standard module system. The consequences of this omission were the proliferation of dozens of incompatible module systems and neverending debates. The situation changed with the R6RS report: nowadays Scheme *has* an official module system, finally.

Unfortunately the official module system is *not* used by all Scheme implementations, and it is quite possible that some implementation will never support it. For instance Chicken, a major implementation, just released version 4, which includes a brand new module system *not compatible* with the R6RS system. You should be aware that the module system (and actually the whole of the R6RS standard) is controversial, and there are good reasons why it is so.

I cannot do anything about the political issues, but I can do something about the technical issues, by explaining the subtle points and by documenting the most common pitfalls. It will take me six full episodes to explain the module system and its trickiness, especially for macro writers who want to write portable code.

19.1 Modules are not first class objects

Since the title of this series is *The Adventures of a Pythonista in Schemeland* let me begin my excursion about the R6RS module system by contrasting it with the Python module system.

The major difference between Python modules and Scheme modules is that Python modules are first class runtime objects which can be passed and returned from functions, as well as modified and introspected freely; Scheme modules, instead, are compile time entities which cannot be imported at runtime, nor passed to functions or returned from functions; moreover they cannot be modified and cannot be introspected.

Python modules are so flexible because they are basically dictionaries. It would not be difficult to implement a Python-like module system in Scheme, by making use of hash-tables, the equivalent of dictionaries. However, the standard module system does not follow this route, because Scheme modules may contain macros which are not first class objects, therefore modules cannot be first class objects themselves [some may argue that having macros which are not first class objects is the root of all evil, and look for alternative routes with macro-like constructs which are however first class objects; however, I do not want to open this particular can of worms here].

Since Scheme modules are not first class objects it is impossible to add names dynamically to a module, or to replace a binding with another, as in Python. It is also impossible to get the list of names exported by a module: the only way is to look at the export list in the source code. It is also impossible to export all the names from a module automatically: one has to list them all explicitly.

In general Scheme is not too strong at introspection, and that it is really disturbing to me since it is an issue that could be easily solved. For instance, my `sweet-macros` library provides introspection features, so that you can ask at runtime, for instance from the REPL, what are the patterns and the literals accepted by a macro, its source code and its associated transformer, even if the macro is a purely compile time entity. It would be perfectly possible to give an introspection API to every imported module. For instance, every module could automagically define a variable - defined both at runtime and compile time - containing the full list of exported names and there could be some builtin syntax to query the list.

But introspection has been completely neglected by the current standard. One wonders how Schemers cope with large libraries/frameworks like the ones we use every day in the enterprise world, which export thounsands and thousands of names in hundreds and hundreds of modules. Let's hope for something better in the future.

I also want to point out a thing that should be obvious: if you have a Scheme library `lib.sls` which defines a variable `x`, and you import it with a prefix `lib.`, you can access the variable with the Python-like syntax `lib.x`. However, `lib.x` in Scheme means something completely different from `lib.x` in Python: `lib.x` in Scheme is just a name with a prefix, whereas `lib.x` in Python means "take the attribute `x` of the object `lib`" and that involves a function call. In other words, Python must perform an hash table lookup everytime you use the syntax `lib.x`, whereas Scheme does not need to do so.

I should also points out that usually (and unfortunately) in the Scheme world people do not use prefixes; by default all exported names are imported, just as it is the case for Python when the (discouraged) style `from lib import *` is used.

19.2 Compiling Scheme modules vs compiling Python modules

Let me continue my comparison between Python modules and Scheme modules, by comparing the compilation/execution mechanism in the two languages. I will begin from Python, by giving a simplified description which is however not far from the truth.

When you run a script `script.py` depending on some library `lib.py`, the Python interpreter searches fo a bytecode-compiled file `lib.pyc`, updated with respect to the the source file `lib.py`; if it finds it, it imports it, otherwise it compiles the source file *on-the-fly*, generates a `lib.pyc` file and imports it. A bytecompiled file is updated with respect to the source file if it has been generated *after* the source file; if you modify the source file, the `lib.pyc` file becomes outdated: the Python interpreter is smart enough to recognize the issue and to seamlessly recompile `lib.pyc`.

In Scheme the compilation process is very much *implementation-dependent*. Here I will give some example of how things work in three representative R6RS-conforming implementations, Ikarus, Ypsilon and PLT Scheme/mzscheme.

Ikarus has two modes of operation; by default it just compiles everything from scratch, without using any intermediate file. This is possible since the Ikarus compiler is very fast. However, this mechanism does not scale; if you have very large libraries, it does not make sense to recompile everything every time you add a little script. Therefore Ikarus (in the latest development version) added a mechanism similar to the Python one; if you have a file `script.ss` which depends on a library `lib.sls` and run the command

```
$ ikarus --compile-dependencies script.ss
Serializing "./lib.sls.ikarus-fasl" ...
```

the compiler will automatically (re)generate a precompiled file `lib.sls.ikarus-fasl` from the source file `lib.sls` as needed, by looking at the time stamps. Exactly the same as in Python. The only difference is that Python compiles to bytecode, whereas Ikarus compile to native code.

Notice that whereas in theory Ikarus should always be much faster of Python, in practice this is not guaranteed: a lot of Python programs are actually calling underlying C libraries, so that Python can be pretty fast in some cases (for instance in numeric computations using numpy).

All I said for Ikarus, can be said from Ypsilon, with minor differences. Ypsilon compiles to bytecode, like Python. Precompiled files are automatically generated without the need to specify any flag, as in Python; however they are stored in a so called auto-compile-cache directory, which by default is situated in `$HOME/.ypsilon`. The location can be changed by setting the environment variable `YPSILON_ACC` or by passing the `--acc=dir` argument to the Ypsilon interpreter. It is possible to disable the cache and to clear the cache; if you are curious about the details you should look at the Ypsilon manual (`man ypsilon`).

PLT Scheme/mzscheme works in a slightly different way. The command

```
$ plt-r6rs script.ss
```

interprets the script and its dependencies on the fly. The command

```
$ plt-r6rs --compile script.ss
```

compiles the script and its dependencies, and stores the compiled file in the *collects* directory, which on my system is in `$HOME/.plt-scheme/4.1.2/collects`. Each library has its own directory of compiled files.

19.3 Compiling is not the same than executing

There are other similarities between a Python (bytecode) compiler and a Scheme compiler. For instance, they are both very permissive, in the sense that they flag very few errors at compile time. Consider for instance the following Python module:

```
$ cat lib.py
x = 1/0
```

The module contains an obvious error, that in principle should be visible to the (bytecode) compiler. However, the compiler only checks that the module contains syntactically correct Python code, it *does not evaluate it*, and generates a `lib.pyc` file without complaining:

```
$ python -m py_compile lib.py # generates lib.pyc without errors
```

The error will be flagged at runtime, only when you import the module:

```
$ python -c"import lib"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "lib.py", line 1, in <module>
    x = 1/0
ZeroDivisionError: integer division or modulo by zero
```

R6RS Scheme uses a similar model. Consider for instance the library

```
$ echo lib.sls
#!r6rs
(library (lib)
 (export x)
 (import (rnrs))
 (define x (/ 1 0)))
```

and the script

```
$ echo script.ss
(import (rnrs) (lib))
```

You can compile the script and the library without seeing any error:

```
$ plt-r6rs --compile script.ss
 [Compiling ./script.ss]
 [Compiling ./.plt-scheme/4.1.2/collects/lib/main.sls]
```

Running the script however raises an error:

```
$  plt-r6rs  script.ss
/: division by zero
```

Like in Python, the error is raised when the module is imported (the technical name in Scheme is *instantiated*).

However, there is a gray area of the R6RS module system here, and implementations are free to not import unused modules. To my knowledge, Ikarus is the only implementation making using of this freedom. If you run

```
$ ikarus --r6rs-script script.ss
```

no error is raised. Ikarus is just *visiting* the module, i.e. taking notes of the names exported by it and of the dependencies, but the module is not evaluated, because it is not used. However, if you use it, for instance if you try to access the `x` variable, you will get the division error at runtime:

```
$ echo script.ss
(import (rnrs) (prefix (lib) lib:))
(begin
  (display "running ...\n")
  (display lib:x))
$  ikarus --r6rs-script script.ss
Unhandled exception:
 Condition components:
   1. &assertion
   2. &who: /
   3. &message: "division by 0"
   4. &irritants: ()
```

Here I have used an import prefix `lib:`, just to be more explicit. Another difference between Ikarus and PLT is that in PLT both the script and the library are compiled, whereas in Ikarus only the library is compiled. In the next episodes we will see many other examples of differences between R6RS-conforming implementations.

Acknowledgments

All the *Adventures* have my name at the top and I take full responsibility for the opinions and the mistakes. But for the parts which are correct, I deserve little credit, since most of the time I am just reporting advice which I have received from the Scheme community, mostly from comp.lang.scheme and ikarus-users, as well from private emails. This is true for all of my *Adventures*, but especially for the six episodes about the module system you are about to read. I was very ignorant about the module system when I started this project, and this work

would not have been possible without the help of Abdulaziz Ghuloum, Derick Eddington, Will Clinger, Eli Barzilay, Matthew Flatt, André van Tolder and many others. Thank you guys, you rock!

CHAPTER

TWENTY

THE COMPILATION AND EVALUATION STRATEGY OF SCHEME PROGRAMS

One of the trickiest aspects of Scheme, coming from Python, is its distinction between *interpreter semantics* and *compiler semantics*.

The problem is that the same program can be executed both with interpreter semantics (typically when typed at the REPL) and with compiler semantics (typically when run as a script), but the way the program behaves is different. Moreover, there are programs which are valid at the REPL but are rejected by the compiler.

To make things worse, the interpreter semantics is unspecified by the R6RS report, whereas the compiler semantics is loosely specified, so that there are at least three different and incompatible semantics about how programs are compiled and libraries are imported: the Ikarus/Ypsilon/IronScheme/MoshScheme one, the Larceny one and the PLT one.

In other words, there is no hope of making programs with the interpreter semantics portable; moreover, there also plenty of programs with compiler semantics which are not portable.

Fortunately the module system works well enough for most simple cases. The proof is that we introduced the R6RS module system in episode 5, and for 15 episode we could go on safely by just using the basic import/export syntax. However, once nontrivial macros enters in the game, things are not easy anymore.

20.1 Interpreter semantics vs compiler semantics

First of all, let me clarify what I do mean by interpreter semantics and compiler semantics, terms which have nothing to do with being an interpreted or compiled language, since both Scheme interpreters and Scheme compilers exhibit both semantics.

Compiler semantics means that a program has (at least) two phases, the run-time phase and the expand-time phase, and some parts of the programs are executed at expand-time and some other parts of the program are executed at run-time. Scheme has a generic concept of *macro expansion time* which is valid even for interpreted implementation when there is no compilation time.

Interpreter semantics means that a program is fully evaluated at runtime, with no distinction between phases (for pure interpreters) or with interleaved expansion and evaluation phases (for

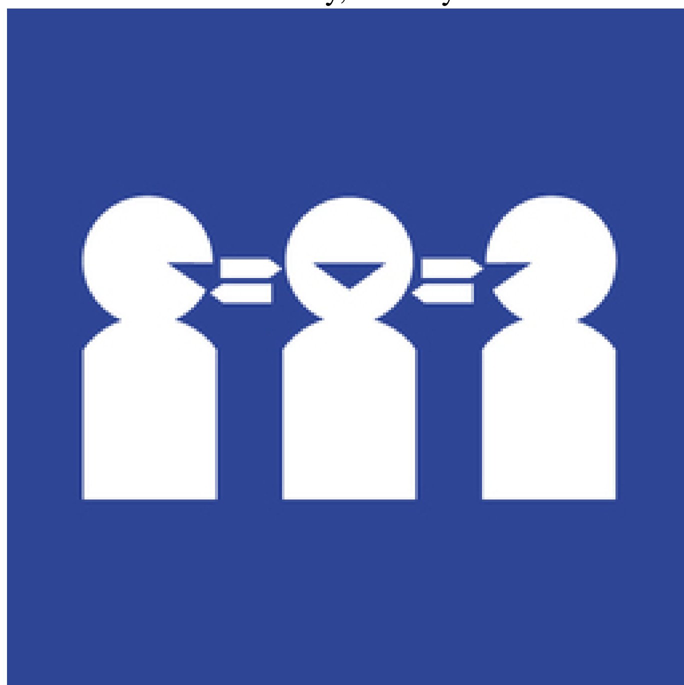
incremental compilers).

For instance Ikarus and Ypsilon work as incremental compilers at the REPL (I consider this as interpreter semantics, by stretching the terminology) and as batch compilers for scripts (for Ypsilon this is true only when the R6RS compatibility flag is set).

Python works as an incremental compiler at the REPL (each time you enter a function in the REPL it is compiled to bytecode, and you can extract the bytecode by looking at `.func_code` attribute) and as batch compiler for scripts.

Conceptually, in Python everything happens at runtime, including bytecode compilation. While technically bytecode compilation is cached, conceptually you may very well think that every module is recompiled at runtime, when you import it - which is actually what happens if the module has changed in the meanwhile.

In short, you can consider Python as an interpreter (as it is usually done) and there is no substantial difference between typing commands at the REPL and writing a script. There are a few minor differences actually, but they are not relevant for what I am discussing now.



Things are quite different in Scheme. The interpreter semantics is *not specified* by the R6RS standard and it is completely implementation-dependent. It is also compatible with the standard to not provide interpreter semantics at all. In particular, PLT Scheme provides a REPL which is actually quite exceptional since it uses compiler semantics and not interpreter semantics.

The compiler semantics i.e. the expansion process of Scheme source code is (loosely) specified by the R6RS standard and is used in libraries. The semantics used in scripts is not clear (in the words of Will Clinger *there is no such thing as an R6RS-conforming Scheme script, because Scheme scripts are described only by a non-binding document that was never ratified*).

The difference between the two semantics is most visible when you have macros depending on helper functions. When a program is read in interpreter semantics, everything happens at runtime: it is possible to define a function and immediately after a macro using that function.

When a program is read in batch compiler semantics instead, *all* the definitions and the expres-

sions are read, the macros are expanded and the program compiled, *before* execution.

Implementations have a considerable freedom in what they allowed to do; for instance Ypsilon scripts use batch compiler semantics when the `--r6rs` flag is set, but by default they use incremental compiler semantics, just as the REPL. On the opposite side of the spectrum, the PLT REPL (in non-R6RS mode) basically uses batch compiler semantics.

In any case the behavior of code typed the REPL is never identical to the behavior of a script: for instance, at the REPL you can import modules at any moment, whereas in a script you must import them at the beginning. There are other subtler differences, for instance in the behavior of continuations. Then bottom line is that you should not believe your REPL blindly.

20.2 Macros and helper functions

As I said, you see the problem of compiler semantics once you start using macros which depend from auxiliary functions. More in general there is the same problem for any identifier which is used in the right hand side of a macro definition and not inside the templates. For instance, consider this simple macro

```
(def-syntax (assert-distinct arg ...)
  #'(#f)
  (distinct? bound-identifier=? #'(arg ...))
  (syntax-violation 'assert-distinct "Duplicated name" #'(arg ...)))
```

which raises a compile-time exception (syntax-violation) if it is invoked with duplicate arguments. Such macro could be used as a helper in macros defining multiple names at the same time, like the `multi-define` macro of episode 9. `assert-distinct` relies on the builtin function `bound-identifier=?` which returns true when two identifiers are equal and false otherwise (this is an extremely simplified explanation, let me refer to the R6RS document for the gory details) and on the helper function `distinct?` defined as follows:

```
;; check if the elements of a list are distinct according to eq?
(define (distinct? eq? items)
  (if (null? items) #t ; no items
      (let+ ((first . rest) items)
        (cond
         ((null? rest) #t); single item
         ((exists (cut eq? first <>) rest) #f); duplicate
         (else (distinct? eq? rest)); look at the sublist
         ))))
```

`distinct?` takes a list of objects and finds out they are all distinct according to some equality operator, of if there are duplicates. Here are a couple of test cases:

```
(test "distinct"
      (distinct? eq? '(a b c))
      #t)
```

```
(test "not-distinct"
      (distinct? eq? '(a b a))
      #f)
```

It is natural, when writing new code, to try things at the REPL and to define first the function and then the macro. The problem is that the code will work in REPL: however, in R6RS-conforming implementations, if you cut and paste from the REPL and convert it into a script, you will run into an error!

The explanation is that in compiler semantics macro definitions and function definitions happens at *different times*. In particular, macro definitions are taken in consideration *before* function definitions, independently from their relative position in the source code. Therefore our example fails to compile since the `assert-distinct` macro makes use of the `distinct?` function which is *not yet defined* at the time the macro is considered, i.e. at expansion time. Actually, not only functions are not evaluated at expansion time and cannot be used inside a macro, but in general the right hand side of any definition is left unevaluated by the compiler. This explains why `(define x (/ 1 0))` is compiled correctly, as we discussed in the previous article .

There are nonportable ways to avoiding writing the helper functions in a separate module. For instance Ypsilon scripts by default (unless the strict R6RS-compatibility flag is set) use interpreter semantics and have no phase separation. On the other end of the spectrum, mzscheme has very strong phase separation, but it is still possible to define helper functions at expand-time without putting them in a separated module, using the nonportable `define-for-syntax` form.

Nevertheless, *the only portable way to make available at expand time a function defined at runtime is to define the function in a different module and to import it at expand time.*

20.3 A note about incremental compilers and interpreters

Ikarus and Ypsilon use the semantics of an *incremental compiler*: each top level block of code is compiled - to native code in Ikarus and to bytecode in Ypsilon - and executed immediately. Each new definition augments the namespace of known names at runtime, both for first class objects and macros. Macros are both defined and expanded at runtime.

It is clear tha the semantics of an incremental compiler is very similar to the semantics of an interpreter; here is an example in Ikarus, where a macro is defined which depends from a helper function:

```
> (define (double x) (* 2 x))
> (def-syntax (m) (double 1))
(m)
2
```

However, an incremental compiler is not identical to an interpreter, since internally it uses phase separation to compile blocks of code; for instance in Ikarus if you put together the previous

definition in a single block you get an error, since the function `double` is known at run-time but not at expand-time:

```
> (let () (define (double x) (* 2 x)) (def-syntax (m) (double 1)) (m))
Unhandled exception
 Condition components:
   1. &who: double
   2. &message: "identifier out of context"
   3. &syntax:
       form: double
       subform: #f
   4. &trace: #<syntax double>
```

There are still Scheme implementations which are pure interpreters and do not distinguish expand time from runtime at all; here is an example in Guile (notice that Guile is *not* an R6RS implementation):

```
guile> (let () (define (double x) (* 2 x)) (define-macro (m) (double 1)) (m))
2
```

I am using `define-macro` here which is the built-in macro mechanism for Guile: as you see the function `double` is immediately available to the macro, even if it is defined inside the same block as the macro, which is not the case for any of the existing R6RS implementations. Notice however that Guile also supports high level macros (via an external library) using compiler semantics.

20.4 Discussion

The interpreter semantics is the most intuitive and easier to understand. In such semantics everything happens at runtime; the code may still be compiled before being executed, as in incremental compiler, but this is an implementation detail: from the point of view of the programmer the feeling is the same as using an interpreter - modulo the tricky point mentioned in the previous paragraph.

The interpreter semantics is also the most powerful semantics of all: for instance, it is possible to redefine identifiers and to import modules at runtime, things which are both impossible in compiler semantics.

If you look at it with honesty, the compiler semantics is basically a performance hack: by separing compilation time from runtime you can perform some computation only once (at compilation time) and gain performance. This is not strange at all: compilers *are* performance hacks. It is just more efficient to convert a a program into machine code with a compiler than to interpret it expression by expression.

The other main reason to favor compilers over interpreters, apart from performance, is compile-time checking. Compilers are able to reject a class of incorrect programs even before executing them. Scheme compilers are traditionally not too strong in this respect, because of dynamic typing and because of the design philosophy of the language (be permissive, we will solve the errors later). Nevertheless, with macros you can in principle add all the compile-time checkings

you want (we just saw the checking for distinct names): it is even possible to turn Scheme into a typed language, like Typed Scheme.

Another (minor) advantage of the compiler semantics is that it makes it easier for static tools to work with a program. For instance in Python an IDE cannot implement autocompletion of names in a reliable way, without having knowledge of the running program. In Scheme an IDE can statically determine all the names imported by the program and thus offer full autocompletion.

CHAPTER

TWENTYONE

THE DIFFERENT MEANINGS OF PHASE SEPARATION

We saw in the latest episode that Scheme programs exhibit phase separation, i.e. some parts of the program are executed at expand time (import declarations, macro definitions and macro expansions) and some other parts are executed at runtime (regular definitions and expressions).

However, things are more complicated than that. There are actually *three* different concepts of phase separation for R6RS-conforming implementations. I will call the three concepts *weak*, *strong* and *extra-strong* phase separation respectively. The difference is in how modules are imported - *instantiated* is the more correct term - and in how variables enter in the namespace.

Ikarus, Ypsilon, IronScheme and MoshScheme have a weak form of phase separation (also called *implicit phasing*): there is a distinction between expand-time and runtime, but it is not possible to import variables in the runtime phase only or in the expand time phase only: variables are imported simultaneously for all phases.

Larceny has a stronger form of phase separation (*explicit phasing*): it can import variables in a specific phase and not in another, depending on the import syntax used. However, if you instantiate a module in more than one phase - for instance both at run-time and at expand-time - only one instance of the module is created and variables are shared.

PLT Scheme has an extra-strong form of phase separation in which phases are completely separated: if you instantiate a module both at run-time and at expand-time, there are two *different and independent instances* of the module.

In this episode I will show the simplest consequences of phase separation. In the next episodes I will show less obvious consequences, such as the tower of metalevels associated to strong phase separation and the multiple instantiation semantics associated to extra-strong phase separation.

21.1 Compile-time, run-time and optimization-time

Before discussing strong phase separation, I want to point out that phase separation, even in its weakest form, has consequences that may be surprising at first. For instance, Scheme compilers (but also the Python compiler) cannot recognize obvious errors like a zero division error in the right hand side of a top level definition, as I have shown in episode 19.

I asked for clarifications on the Ikarus mailing list. It turns out that Scheme compilers are not stupid: they can recognize the zero division error, but they cannot signal it since it is forbidden by the Scheme specification. For instance, Llewellyn Pritchard (Leppie), the implementor of IronScheme wrote:

> *In IronScheme, if I can detect there is an issue at compile time, I simply defer the computation to the runtime, or could even just convert it into a closure that will return an error. This is only one of the things that make Scheme quite hard to implement on a statically typed runtime such as the CLR, as it forces me to box values at method boundries and plenty type checking at runtime.*

whereas Abdul Aziz Ghuloum wrote:

> *Actually, Ikarus does some type checking, and it does detect the division by 0. It however cannot do anything about it in this case since Scheme requires that the exception be raised when the division operation is performed at run time.*

Aziz went further and explained that Ikarus is able to evaluate expressions like

```
(define x 5)
(define y (+ x 1))
(define z (* x y))
```

both in top level definitions in and internal definitions; however, it does so in the optimization phase, *after* the expansion phase, i.e. too late to make the definitions available to macros. It could however at least report a syntax warning (take it as a feature request, Aziz! ;-)

Aziz also brought up an argument in favor of the current specification. First of all, it is pretty clear that we want expressions like

```
(define thunk (lambda () (/ 1 0)))
```

to be compilable, because it is useful to have functions that can raise predictable errors, especially when writing test cases.

Now, a module is not really different from a giant thunk; importing a module calls the thunk (this is essentially what *module instantiation* is) and possibly raises errors at runtime, but the module per se must be compilable even if contains errors which are detectable at compile time.

The two-phases compilation strategy has the advantage of keeping the compiler conceptually simple, working as a traditional preprocessor integrated in the language: we know that the compiler will manage the macros, but will not perform any evaluation.

Actually, there are strong arguments against having the compiler evaluating generic top level or internal definitions; consider for instance the case when you are reading some data from standard input (`(define date (read)`): if the definition were evaluated at compile-time, the compiler would stop during compilation to read the data.

Then, some time later, at execution time, the program would stop again to read potentially different data, so that macros would use the compilation time data and the rest of the program the runtime data!

That would be madness. Clearly it makes no sense to evaluate at compile-time definitions depending on run-time values, except possibly at the REPL, where everything happens at run-time and the phases are intermingled.

Finally, the two-phases enable cross compilation: macros will be expanded independently from the architecture, whereas the runtime structures will be compiled and linked differently depending on the architecture of the target processor.

21.2 Strong vs weak phase separation

To explain the practical difference between strong and weak phase separation let me go back to the example of the `assert-distinct` macro of episode 20. I have put the helper function (`distinct?`) in the `(aps list-utils)` module, so that you can import it. This is enough for Ikarus, but it is not enough for PLT Scheme or Larceny. In other words, in Ikarus (but also IronScheme, MoshScheme and all the systems using the psyntax module system) the following script

```
(import (rnrs) (sweet-macros) (only (aps list-utils) distinct?))

(def-syntax (assert-distinct arg ...)
  #'(#f)
  (distinct? bound-identifier=? #'(arg ...))
  (syntax-violation 'assert-distinct "Duplicate name" #'(arg ...)))
```

is correct, since the import form instantiates the module `(aps list-utils)` both at run-time and expand-time, but in PLT Scheme and Larceny it raises an error:

```
$ plt-r6rs assert-distinct.ss
assert-distinct.ss:5:3: compile: unbound variable in module
(transformer environment) in: distinct?
```

The problem is that PLT Scheme and Larceny have strong phase separation and require *phase specification*: by default names defined in external modules are imported *only* at runtime, *not* at compile time. In a sense this is absurd since names defined in an external pre-compiled modules are of course known at compile time (this is why Ikarus has no trouble importing them); nevertheless PLT Scheme (and Larceny) forces you to specify at which phase the functions must be imported.

In particular, if you want to import `distinct?` at expand time you must use the `(for expand)` form:

`(import (for (only (aps list-utils) distinct?)  expand))`

With this import form, the script is portable in all R6RS implementations, but its meaning is different: in the psyntax based implementations the name `distinct?` is imported both at runtime and at expand-time, whereas in PLT and Larceny it is imported only at expand time.

Notice that there are portability issues associated with phase separation. Not using the phase specification syntax results in non-portable code, therefore if you care about portability you must use phase specification even if your implementation does not use it :-(

For instance in systems based on psyntax and in Ypsilon - which is not based on psyntax - this program

```
(import (rnrs) (for (only (aps list-utils) distinct?) expand))
(display distinct?)
```

will run, but in PLT Scheme and Larceny it will not even compile.

In a sense, implementation with strong phase separation are more powerful than implementations with weak phase separation, since with implicit phasing it is *impossible* to import the name `distinct?` at expand time and not at runtime - notice however that more powerful does not mean necessarily better and the implementations with weak phase separation are easier to use.

The situation for people coming from implementations with strong phase separation is no nice either. For instance the program

```
(import (rnrs) (for (only (aps list-utils) distinct?) run))
(display distinct?)
```

will run on all implementations, but you cannot rely on the fact that the named `distinct?` will be imported only at run-time and not at expand-time.

The point however is moot since the R6RS forbids the same name to be used with different bindings in different phases (see section 7.1, page 23). In particular, if you import the name `distinct?` at run-time the compiler will reserve the name for all phases: it cannot be reused at expand time, unless it has the same binding.

In other words, the namespaces in the different phases are separated but *not completely independend*, which in my opinion undermines the concept of strong phase separation. I believe PLT Scheme in non-R6RS mode has fully independent namespaces for different phases, but this again is not portable.

21.3 A note about politics

The reason for such limitations and inconsistencies can be inferred from this extract from R6RS editors mailing list (from the answer to formal comment 92):

> *A precise specification of the library system remains elusive, partly because different implementors still have different ideas about how the library system should work....*
>
> *The different opinions are supported by two different reference implementations of R6RS libraries: one by Van Tonder and one by Ghuloum and Dybvig. In addition, PLT Scheme implements a library system...*
>
> *Despite the differences in the reference implementations, it appears that many programs will run the same in both variants of the library system. The overlap appears to be large enough to support practical portability between the variants.*
>
> *Under the assumption that the overlap is useful, and given the lack of consensus and relative lack of experience with the two prominent variants of draft R6RS libraries, the R6RS specification of libraries should be designed to admit both of the reference implementations. As a design process, this implementation-driven approach leaves something to be desired, but it seems to be the surest way forward.*

Basically, the R6RS standard is the result of a compromise between the partisans of explicit phasing - people wanting to control in which phases names are imported - and the partisan of implicit phasing - people wanting to import names at all phases, always.

A compromise was reached to make unhappy both parties.

The same kind of compromise was reached on the subject of multiple instantiation: all behaviors are accepted by the R6RS standard, so you cannot rely on the number the times a library is instantiated.

For instance, consider a simple do nothing library like the following:

```
#!r6rs
(library (x)
(export)
(import (rnrs))
(display "instantiated x!\n")
)
```

If you now run the following script

```
$ cat script.ss
(import (for (x) expand run))
```

the message `instantiated x!` will be printed only *once* by Larceny, but *twice* by PLT Scheme. For comparison, Ypsilon prints the message only once (it has single instantiation semantics) and Ikarus does not print any message at all (!), since the module is not used (it would print the message only once if the module were used).

In other words, authors of portable libraries cannot rely on multiple instantiation, nor on single instantiation.

The final outcome for the R6RS module system is certainly unhappy, but I guess it was the best that the R6RS editors could obtain, given the pre-existing situation. Another point in favor of languages designed by (benevolent) dictators!

CHAPTER

TWENTYTWO

THE DARK TOWER OF META-LEVELS

I said in the previous episode that even if your implementation of choice does not use explicit phasing, you must understand it in order to write portable programs. Truly understandanding explicit phasing is nontrivial, since you must reason in terms of a (Dark) Tower of import levels, or *meta-levels*.

Since the publication of the Aristotle's Metaphysics, the word *meta* has been associated to arcane and difficult matters. The concept of meta-level is no exception to the rule. You can find a full description of the tower of meta-levels in the R6RS document, in a rather dense paragraph in section 7 that will make your head hurt.

There is also a celebrated paper by Matthew Flatt, *Composable and Compilable Macros* (a.k.a. You want it when) which predates the R6RS by many years and is more approachable. Its intent is to motivate the module system used by PLT Scheme, which made popular the concept of tower of meta-levels.

Meta-levels are just another name for phases. We have already encountered two meta-levels: the run-time phase (meta-level 0) and expand time phase (meta-level 1). However, the full tower of meta-levels is arbitrarily high and extends in two directions, both for positive and for negative integers (!)

Scheme implementations with explicit phasing allow you to import a module at a generic meta-level `N` with the syntax `(import (for (lib) (meta N)))`, where N is an integer. The forms `(import (for (lib) run))` and `(import (for (lib) expand))` are just shortcuts for `(import (for (lib) (meta 0)))` and `(import (for (lib) (meta 1)))`, respectively.

Instead of discussing much theory, in this episode I will show two concrete examples of macros which require importing variables at a nontrivial meta-level *N*, with *N<0* or *N>1*.

For convenience I am keeping all the code of this episode into a package called `experimental`, which you can download from here: http://www.phyast.pitt.edu/~micheles/scheme/experimental.zip

Figure 22.1: Aziz faces the Dark Tower of Meta-levels

22.1 An easy-looking macro with a deep portability issue

My first example is a compile time `name -> value` mapping, with some introspection:

```
#!r6rs
(library (experimental static-map)
(export static-map)
(import (rnrs) (sweet-macros))

(def-syntax (static-map (name value) ...)
  #'(syntax-match (<names> name ...)
      (sub (ctx <names>) #''(name ...))
      (sub (ctx name) #'value)
      ...))
)
```

This is a kind of second order macro, since it expands to a macro transformer; its usage is obvious in implementations with implicit phasing:

```
$ cat use-static-map.ss


(import (rnrs) (sweet-macros) (for (experimental static-map) expand))
;; the for syntax is ignored in implementations with implicit phasing

(def-syntax color (static-map (red #\R) (green #\G) (yellow #\Y)))

(display "Available colors: ")
(display (color <names>))
(display (list (color red) (color green) (color yellow)))
(newline)
```

`color` is a macro which replaces a symbolic name in the set `red`, `green`, `yellow` with its character representation (`#\R`, `#\G`, `#\Y`) at expand-time (notice that in Scheme characters are different from strings, i.e. the character `\#R` is different from the string of length 1 `"R"`).

If you run this script in Ikarus or Ypsilon or Mosh you will get the following unsurprising result:

```
$ ikarus --r6rs-script use-static-map.ikarus.ss
Available colors: (red green yellow)(R G Y)
```

However, in PLT and Larceny, the above will fail. The PLT error message is particularly cryptic:

```
$ plt-r6rs use-static-map.ss
/home/micheles/.plt-scheme/4.0/collects/experimental/static-map.sls:8:25:
```

```
compile: bad syntax; reference to top-level identifier is not allowed,
because no #%top syntax transformer is bound in: quote
```

I was baffled by this error, so I asked for help in the PLT mailing list, and I discovered that there is nothing wrong with the client script and that there is no way to fix the problem by editing it: the problem is in the library code!

The problem is hidden, since you can compile the library without issues and you see it only when you use it. Also, the fix is pure dark magic: you need to rewrite the import code in `(experimental static-map)` by replacing

```
(import (rnrs))
```

with

```
(import (rnrs) (for (rnrs) (meta -1))
```

i.e. the `static-map` macro must import the `(rnrs)` environment at meta-level -1! Why it is so? and how should I interpret meta-level -1?

22.2 Negative meta-levels

Matthew Flatt explained to me how meta-levels work. The concept of meta-level is only relevant in macro programming. When you define a macro, the right hand side of the definition can only refer to names which are one meta-level up, i.e. typically at meta-level 1 (expand time). On the other hand, inside a template one goes back one meta-level, and therefore usually a template expands at meta-level 0 (run-time).

However, in the case of the `static-map` macro, the template is itself a `syntax-match` form, and since the templates of this inner `syntax-match` expand one level down, we reach meta-level -1. This is why the macro needs to import the `(rnrs)` bindings at meta-level -1 and why the error message says that `quote` is unknown. The comments below should make clear how meta-levels mix:

```
(def-syntax static-map                                ;; meta-level 0
 (begin
   <there could be code here ...>                     ;; meta-level 1
 (syntax-match ()
  (sub (static-map (name value) ...)
       #'(begin
           <there could be code here ...>             ;;  meta-level 0
           (syntax-match (<names> name ...)
             (sub (ctx <names>)
                #''(name ...))                        ;; meta-level -1
             (sub (ctx name)
                #'value)                              ;; meta-level -1
             ...))))
```

Actually `quote` is the only needed binding, so it would be enough to import it with the syntax `(import (for (only (rnrs) quote) (meta -1)))`. If we ignored the introspection feature, i.e. we commented out the line

```
(sub (ctx <names>) #"(name ...))
```

there would be no need to import `quote` at meta-level -1, and the macro would work without us even suspecting the existence of negative meta-levels.

Things are even trickier: if we keep the line `(sub (ctx <names>) #"(name ...))` in the original macro, but we do not use it in client code, the original macro will apparently work, and will break at the first attempt of using the introspection feature, with an error message pointing to the problem in client code, but not in library code :-(

22.3 Meta-levels greater than one

It is clear that the meta-level tower is theoretically unbound in the negative direction, since you can nest macro transformers at any level of depth, and each level decreases the meta-level by one unity; on the other hand, the tower is theoretically unbound even in the positive direction, since a macro can have in its right hand side a macro definition which right hand side will requires bindings defined at an higher level, and so on. In general nested macro definitions *increase* the meta-level; nested macro templates *decrease* the meta-level.

Here is an example of a macro which requires importing names at meta-level 2:

```
$ cat meta2.ss
```

```
#!r6rs
(import (rnrs)
        (for (sweet-macros) (meta 0) (meta 1))
        (for (only (rnrs) begin lambda display) (meta 2)))

(def-syntax m
  (let ()
    (def-syntax m2
      (begin                                   ;; begin, display and
        (display "at metalevel 2\n")           ;; lambda are used here
        (lambda (x) "expanded-m\n")))          ;; at meta-level 2
    (define _ (display "at metalevel 1\n")) ;; meta-level 1
    (lambda (x) (m2))))                        ;; here

(display (m))
```

Notice that right hand side of a `def-syntax` form does not need to be `syntax-match` form; the only requirement for it is to be a transformer, i.e. a one-argument procedure. In this example the inner macro `m2` has a transformer returning the string `"m-expanded"` whereas the outer macro `m` has a transformer returning the expansion of `(m2)` i.e. again the string `"m-expanded"`. Running the script will print the following:

```
$ ikarus --r6rs-script meta2.ss
at meta-level 2
at meta-level 1
expanded-m
```

You will get the same in Larceny and in sufficiently recent versions of PLT Scheme (> 4.1.3). Currently Ypsilon raises an exception but this is just a bug (already fixed in the trunk).

22.4 Discussion

The concept of meta-level is tricky. On one hand, there are only two physical meta-levels, i.e. the run-time (when the code is executed) and the compile time (when the code is compiled). On the other hand, conceptually there is an arbitrary number of positive meta-levels ("before compile time") and negative meta-levels ("after run-time") which have to be taken in account to compile/execute a program correctly: everytime the compiler look at a nested macro, it has to consider the innermost level first, and the outermost level last.

The power (and the complication) of phase specification is that the language used at a given phase can be different from the language used in the other phases. Suppose for instance you are a teacher, and you want to force your students to write their macros using only a functional subset of Scheme. You can do so by importing at compile time all R6RS procedures except the nonfunctional ones (like `set!`) while importing at run-time the whole of R6RS. You could even perform the opposite, and remove `set!` from the run-time, but allowing it at compile time.

However, personally I do not feel a need to distinguish the languages at different phases and I like Scheme to be a Lisp-1 language with a single namespace for all variables. I am also not happy with having to keep manually track of the meta-levels, which is difficult and error prone when writing higher order macros. Moreover, in PLT and Larceny writing a macro which expands to a nested macro with N levels is difficult, since one has to write by hand all the required meta imports.

All this trouble is missing in Ypsilon and in the implementations based on psyntax. In such systems importing a module imports its public variables for *all* meta-levels. In other words all meta-levels share the same language: the tower of meta-levels is effectively destroyed (one could argue that the tower is still there, implicitly, but the point is that the programmer does not need to think about it explicitly). The model of implicit phasing was proposed by Kent Dybvig and Abdul Aziz Ghuloum, who wrote his Ph. D. thesis on the subject.

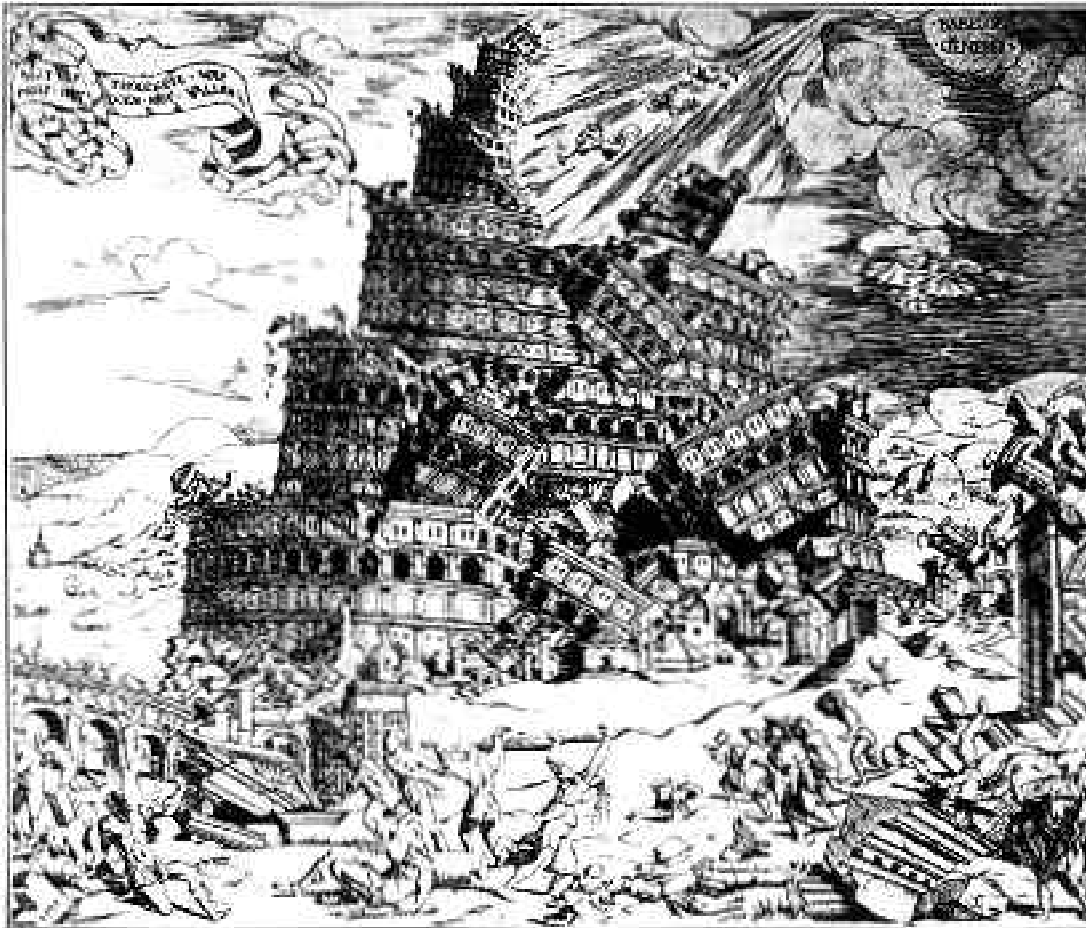


Figure 22.2: Aziz destroys the Tower of Meta-levels

CHAPTER

TWENTYTHREE

SEPARATE COMPILATION AND IMPORT SEMANTICS

Scheme is all about times: there is a run-time, an expand-time, and a discrete set of times associated to the meta-levels. When separate compilation is taken in consideration, there is also another set of times: the times when the libraries are separately compiled. Finally, if the separately compiled libraries define macros which are used in client code, there is yet another set of times, the *visit times*.

To explain what the visit time is, suppose you have a low level library `L`, compiled yesterday, defining a macro you want to use in another middle level library `M`, to be compiled today. The compiler needs to know about the macro defined in `L` at the time of the compilation of `M`, because it has to expand code in `M`. Therefore, the compiler must look at `L` and re-evaluate the macro definition today (the process is called *visiting*). The visit time is different from the time of the compilation of `L` as it happens just before the compilation of `M`.

Here is a concrete example. Consider the following low level library `L`, defining a macro `m` and an integer variable `a`:

```
#!r6rs
(library (experimental L)
(export m a)
(import (rnrs) (sweet-macros))
(def-syntax m
  (begin
    (display "visiting L\n")
    (lambda (x) #f)))
(define a 42)
(display "L instantiated\n")
)
```

You may compile it with PLT Scheme:

```
$ plt-r6rs --compile L.sls
 [Compiling /usr/home/micheles/gcode/scheme/experimental/L.sls]
visiting L
```

Since the right hand side of a macro definition is evaluated at compile time the message `visiting L` is printed during compilation, as expected. Consider now the following middle level library `M` using the macro `m`:

```
#!r6rs
(library (experimental M)
(export a)
(import (rnrs) (experimental L))
(m); this line is expanded at compile-time
(display "M instantiated\n"); at run-time
)
```

In this example the compiler needs to visit `L` in order to compile `M`. This is actually what happens:

```
$ plt-r6rs --compile M.sls
 [Compiling /usr/home/micheles/gcode/scheme/experimental/M.sls]
visiting L
```

If you comment out the line with the macro call, the compiler does not need to visit `L` anymore; some implementations may take advantage of this fact (Ypsilon and Ikarus do). However, PLT Scheme will continue to visit `L` in any case.

23.1 The mysterious import semantics

It is time to ask ourselves the crucial question: what does it mean to *import* a library?

For a Pythonista, things are very simple: importing a library means executing it at run-time. For a Schemer, things are somewhat complicated: importing a library implies that some basic operations are performed at compile time - such as looking at the exported identifiers and at the

dependencies of the library - but there is also a lot of unspecified behavior which may happen both a compile-time and at run-time. In particular at compile-time a library may be only visited, i.e. its macro definitions can be re-evaluated - or can be only instantiated, or both. Different things happens in different situations and in the same situation different implementations can perform different operations.

The example of the previous paragraph is useful in order to get a feeling of what is portable behavior and what is not. Let me first consider what happens in Ikarus. If I want to compile `L` and `M` in Ikarus, I need to introduce a helper script `H.ss`, since Ikarus has no direct way to compile a library from the command line. Here is the script:

```
$ cat H.ss
```

```
#!r6rs
(import (rnrs) (experimental M))
(display a)
```

Here is what I get:

```
$ ikarus --compile-dependencies H.ss
visiting L
Serializing "/home/micheles/gcode/scheme/experimental/M.sls.ikarus-fasl" ...
Serializing "/home/micheles/gcode/scheme/experimental/L.sls.ikarus-fasl" ...
```

Ikarus is lazier than PLT: for instance, if you comment the line invoking the macro in `M.sls` and you recompile the dependencies, then the library `M` is not visited.

Both PLT and Ikarus do not instantiate `L` in order to compile `M` (it is not needed) but Ypsilon does. You may check that if you introduce a dummy macro in `M`, depending on the variable `a` defined in `L` (for instance if you add a line `(def-syntax dummy (lambda (x) a))`) then the library `L` must be instantiated in order to compile `M`, and all implementations do so.

Let us consider the peculiarities of Ypsilon, now. Ypsilon does not have a switch to compile a library without executing it - even if this is possible by invoking the low level compiler API - so we must execute `H.ss` to compile its dependencies:

```
$ ypsilon --r6rs H.ss
L instantiated
visiting L
M instantiated
42
```

There are several things to notice here, since the output of Ypsilon is quite different from the output of Ikarus

```
$ ikarus --r6rs-script H.ss
L instantiated
42
```

and the output of PLT:

```
$ plt-r6rs H.ss
visiting L
visiting L
L instantiated
M instantiated
42
```

The first thing to notice is that both in Ikarus and in PLT we relied on the fact that the libraries were precompiled, so in order to perform a fair comparison we must run Ypsilon again (this second time the libraries L and M will be precompiled):

```
$ ypsilon --r6rs H.ss
L instantiated
M instantiated
42
```

You my notice that this time the library `L` is not visited: it was visited the first time, in order to compile `M`, but there is no need to do so now. During compilation of `M` macros has been expanded and the byte-code of `M` contains the expanded version of the library; moreover the helper script `H` does not use any macro so it does not really need to visit `L` or `M` to be compiled. The same happens for Ikarus. PLT instead visits `L` twice to compile `H.ss`. In PLT all dependencies (both direct and indirect) are always visited when compiling. Only if we compile the script once and for all

```
$ plt-r6rs --compile H.ss
 [Compiling /usr/home/micheles/gcode/scheme/experimental/H.ss]
 [Compiling /home/micheles/.plt-scheme/4.1.5.5/collects/experimental/M.sls]
visiting L
visiting L
```

the `visiting L` message will not be printed:

```
$ plt-r6rs H.ss
L instantiated
M instantiated
42
```

23.2 More implementation-dependent details

Having performed the right number of compilations now the output of PLT and Ypsilon are the same; nevertheless, the output of Ikarus is different, since Ikarus does not instantiate the middle level library `M`. The reason is the *implicit phasing* semantics of Ikarus (other implementations based on psyntax would exhibit the same behavior): the helper script `H.ss` is printing the variable `a` which really comes from the library `L`. Ikarus is clever enough to recognize this fact and lazy enough to avoid instantiating `M` without need.

On the other hand, Ypsilon performs eager instantiation and it instantiates (once) all the libraries it imports (both directly and indirectly), even at compile time and even in situations when the instantiation would not be needed for compilation of the client library. As you see, Scheme implementations have a lot of latitude in such matters.

The implementations based on psyntax are the smartest out there, but begin smart is not always the same thing as being good. It is good to avoid instantiating a library if the instantiation is really unneeded; it is bad if the library has some side effect, since the side effect will mysteriously disappear. In our example the side effect is just printing the message `M instantiated`, in more sophisticated examples the side effect could be writing a log on a database, or initializing some variable, or registering an object, or something else.

For instance, suppose you want to collect a bunch of functions into a global registry acting as a dictionary of functions. You may do so as follows:

```
(library (my-library)
(export)
(import (registry))

(registry-set! 'f1 (lambda (x) 'something1))
(registry-set! 'f2 (lambda (x) 'something2))
...
)
```

The library here does not export anything, since it relies on side effects to populate the global registry of functions; the idea is to access the functions later, with a call of kind `(registry-ref <func-name>)`. This design as it is is not always portable to systems based on psyntax, because such systems will not instantiate the library (the library does not export any variable, nothing of the library can be used in client code!). This can easily be fixed, by introducing an initialization function to be exported and called explicitly from client code, which is a good idea in any case.

Analogously, a library based on side effects at visit time, i.e. in the right hand side of macro definitions, is not portable, since systems based on psyntax will not visit a library with macros which are not used. This is relevant if you want to use the technique described in the You want it when? paper: in order to make sure that the technique work on systems based on psyntax, you must make sure that the library exports at least one macro which is used in client code. Curious readers will find the gory details in this thread on the PLT mailing list.

Generally speaking, you cannot rely on the number of times a library will be instantiated, even within the *same* implementation! Abdulaziz Ghuloum gave a nice example in the Ikarus and PLT lists. You have the following libraries:

```
(library (T0) (export) (import (rnrs)) (display "T0\n"))
(library (T1) (export) (import (for (T0) run expand)))
(library (T2) (export) (import (for (T1) run expand)))
(library (T3) (export) (import (for (T2) run expand)))
```

and the following script:

```
#!r6rs
(import (T3))
```

Running the script (without precompilation) results in printing T0:

```
0 times for Ikarus and Mosh
1 time for Larceny and Ypsilon
10 times for plt-r6rs
13 times for mzscheme
22 times for DrScheme
```

T0 is not printed in psyntax-based implementations, since it does not export any identifier that can be used. T0 is printed once in Larceny and Ypsilon since they are single instantiation implementations with eager import. The situation in PLT Scheme is subtle, and you can find a detailed explanation of what it is happening in this other thread. Otherwise, you will have to wait for the next (and last!) episode of this series, where I will explain the reason why PLT is instantiating (and visiting) modules so many times.

CHAPTER TWENTYFOUR

MUTATING VARIABLES ACROSS MODULES

There are situations where it is handy to mutate a global variable or a data structure across modules, for instance to keep a counter or a registry of objects. However, direct mutation of exported variables is forbidden by the R6RS standard. Consider for instance a module exporting a variable `x` and a function `incr-x` with side effects affecting that variable:

```
#!r6rs
(library (experimental mod1)
  (export x incr-x)
  (import (rnrs))

  (define x 0)
  (define (incr-x)
    (set! x (+ 1 x))
    x)
)
```

This kind of side effect is ruled out by the R6RS specification (section 7.2): *exported variables must be immutable*. This is the reason why Ikarus, Ypsilon and Larceny reject the code with errors like `attempt to export mutated variable` or `attempt to modify immutable variable`. The current SVN version PLT Scheme also raises an error, but the official version (4.1.5 at the time of this writing) is buggy (I submitted the bug report).

24.1 Mutating internal variables

Consider now a module exporting a function with side effects affecting a non-exported variable:

```
#!r6rs
(library (experimental mod2)
  (export get-x incr-x)
  (import (rnrs))
```

```
  (define x 0)

  (define (get-x)
    x)

  (define (incr-x)
    (set! x (+ 1 x))
    x)

  (display "Instantiated mod2\n")
)
```

This is a valid library which compiles correctly. The accessor function `get-x` gives access to the internal variable `x`. We may import it at the REPL and we may experiment with it:

```
$ ikarus
> (import (experimental mod2)); this does not instantiate mod2 immediately
> (get-x); now mod2 must be instantiated
Instantiated mod2
0
> (incr-x)
1
> (incr-x)
2
> (get-x)
2
```

Everything works as one would expect. However, things are trickier when phase separation enters in the game.

24.2 Mutating variables across phases

A Scheme implementation exhibits a *cross-phase side effect* if mutating a variable at expand time affects the value of the same variable at run-time. All R6RS implementations - except PLT Scheme which uses different instances for different phases - may have cross-phase side effects. On the other hand, in all R6RS implementations cross-phase side effects can be avoided by using separated compilation.

In order to give a concrete example, consider the following script:

```
$ cat use-mod2.ss

(import (rnrs) (sweet-macros) (for (experimental mod2) expand run))

(def-syntax m
  (lambda (x)
    (display "At expand-time x=")
    (display (incr-x))
```

```
    (newline)
    "m-expanded"))

(m)

(begin
  (display "At run-time x=")
  (display (incr-x))
  (newline))
```

Here we formally import the module `mod2` twice, both at run-time and at expand time. In PLT Scheme (which is the only implementation with explicit phasing *and* multiple instantiation) there are two fully separated instances of the module, and running the script returns the following:

```
$ plt-r6rs use-mod2.ss
Instantiated mod2
At expand-time x=1
Instantiated mod2
At run-time x=1
```

The fact that `x` was incremented at compile-time has no effect at run-time, since the run-time variable `x` belongs to a completely different instance of the module. In systems with single instantiation instead, there is only a *single instance of the module for all phases*, so that incrementing `x` at expand-time has effect at run-time:

```
$ ikarus --r6rs-script use-mod2.ss
Instantiated mod2
At expand-time x=1
At run-time x=2
```

You would get the same with Ypsilon and Larceny (Larceny has explicit phasing but single instantiation and if you import a module in more than one phase the variables are shared amongst the phases, so that cross-phase side effects may happen).

The phase crossing effect only happens because the script is executed immediately after compilation *in the same process*. Having compile-time effects affecting run-time values is *evil*, since it breaks separate compilation. If we turn the script into a library and we compile it separately, it is clear than the run-time value of `x` cannot be affected by the compile-time value of `x` (maybe the code was compiled 10 years ago!).

24.3 Cross-phase side effects and separate compilation

Let me explain in detail how separate compilation works in Ikarus, Ypsilon and PLT Scheme. Suppose we turn the previous script into a library:

```
$ cat mod3.ss

#!r6rs
(library (experimental mod3)
(export run)
(import (rnrs) (sweet-macros) (for (experimental mod2) expand run))

(def-syntax m
  (lambda (x)
    (display "At expand-time x=")
    (display (incr-x))
    (newline)
    "m-expanded"))

(define (run) ;; this is executed at runtime
  (display "At run-time x=")
  (display (incr-x))
  (newline))

(m) ;; this is executed at expand time
)
```

and let us invoke this library though a script `use-mod3.ss`:

```
#!r6rs
(import (rnrs) (experimental mod3))
(run)
```

If we use PLT Scheme, the value of `x` is the same as before:

```
$ plt-r6rs use-mod3.ss
Instantiated mod2
At expand-time x=1
Instantiated mod2
Instantiated mod2
At run-time x=1
```

This is expected: turning a script into a library did not make anything magic happens (actually mod2 is being instantiated once more during the compilation of mod3, but that should not be surprising). On the other hand, things are very different if we run the same code under different implementations.

For instance in Ypsilon the first time the script is run it prints three lines:

```
$ ypsilon --r6rs use-mod3.ss
Instantiated mod2
At expand-time x=1
At expand-time x=2
At run-time x=3
```

However, if we run the script again it prints just two lines:

```
$ ypsilon --r6rs use-mod3.ss
Instantiated mod2
At run-time x=1
```

The reason is that the first time Ypsilon compiles the libraries, using the same module instance, so that there is a single `x` variable which is incremented twice at expand time and once at run-time. The second time there is nothing to recompile, so only the run-time `x` variable is incremented, and there is no reference to the compile time instance.

The situation for Ikarus is slightly different. If we use the `--r6rs-script` flag we get the same output as before, when `mod3` was just a script:

```
$ ikarus --r6rs-script use-mod3.ss
Instantiated mod2
At expand-time x=1
At run-time x=2
```

However, this only happens because Ikarus is compiling all the libraries at the same time (whole compilation). If we use separate compilation we get:

```
$ ikarus --compile-dependencies use-mod3.ss
Instantiated mod2
At expand-time x=1
Serializing "/home/micheles/gcode/scheme/experimental/mod3.sls.ikarus-fasl" ...
Serializing "/home/micheles/gcode/scheme/experimental/mod2.sls.ikarus-fasl" ...
```

As you see, the message `At expand-time x=1` is printed when `mod2` is compiled. If we run the script `use-mod3.ss` now, we get just the run-time message:

```
$ ikarus --r6rs-script use-mod3.ss
Instantiated mod2
At run-time x=1
```

In Ikarus, Ypsilon and Larceny, the same invocation of this script returns different results for the variable `x`, depending if the libraries have been precompiled or not. This is ugly and error prone. The multiple instantiation mechanism of PLT Scheme has been designed to avoid this problem: in PLT one consistently gets always the same result, which is the result one would get with separation compilation.

I must notice that you could get the same behavior in non-PLT implementations by spawning two separate processes, one after the other: the first to compile the script and its libraries, and the second to execute it. That would make sure that incrementing `x` in the compilation phase would not influence the value of `x` at run-time.

24.4 Conclusion

This is the last episode of part IV. You should have an idea of how the R6RS module system works, and you should be able to grasp the reasons behind the different implementation choices.

In particular, it should be clear that side effects are tricky, that you cannot rely on the compilation/visiting/instantiation procedure being the same in different implementations, that phase separation means different things in different Scheme systems.

Still, I have left out many relevant things. In order to say everything there is to say about the subject, I should have at least doubled the number of episodes.

I did not want to get lost in excessive detail. Instead, I have decided to continue my series with another block of episodes about macros, and to fill the remaining gaps about the module systems in future Adventures.

So, as always, stay tuned and keep reading!

CHAPTER

TWENTYFIVE

BACK TO MACROS

Macros are the reason why I first became interested in Scheme, five or six years ago. At the time - as at any time - there was a bunch of people trolling in comp.lang.python, arguing for the addition of macros to the language. Of course most Pythonistas opposed the proposal.

At the time I had no idea of the advantages/disadvantages of macros and I felt quite ignorant and powerless to argue. I never liked to feel ignorant, so I decided to learn macros, especially Scheme macros, because they are the state of the art for what concerns the topic.

Nowadays I have some arguments to back up the position against macros. I have two main objections, one technical (less important) and one political (more important).

The technical reason is that I do not believe in macros for languages without S-expressions. There are plenty of examples of macro systems without S-expressions - for instance Dylan or PLOT in the Lisp world and Logix and MetaPython in the Python world, but none of them ever convinced me. Scheme macros are much better because of the homoiconicity of the language ("homoiconicity" is just a big word for the *code is data* concept). [Notice that technically Scheme macros work on syntax objects and not directly on S-expressions like traditional Lisp macros, but this is a subtle point I will discuss when talking about hygiene; I can skip it for the moment being.]

I have already stated in episode 12 my political objection, i.e. my belief that macros have a high cost in terms of complication of the language (look how complicated the R6RS module system is!). Moreover, code based on macros tends to be too clever, difficult to debug, and sometime idiosyncratic; I do not want to maintain code such kind of code in a typical enterprise context, with programmers of any kind of competence. Sometimes I wish that even Python was a simpler language! There is a difference between *simpler* and *dumber*, of course. I am *not* implying that every enterprise should adopt only enterprise-oriented languages; as a matter of fact various cutting edge enterprises are taking advantage of non-conventional and/or research-oriented languages, but I see them as exceptions to the general rule.

My opinion is based on the fact that on my daily work (I use Python exclusively there) I have never felt the need for macros. For instance, I had occasion to write both small declarative languages and small command-oriented languages, but they were so simple that I had no need for Scheme macros. Actually, judging from my past experience, I think extremely unlikely that I will ever need something as sophisticated as Scheme macros in my daily work. The one thing that I miss in Python which Scheme has is *pattern matching*, not macros.

Having said that, I do not think that macros are worthless, and actually I think they are ex-

tremely useful and important in another domain, i.e. in the domain of design and research about programming languages. Scheme is certainly not the only language where you can experiment with language design, it is just the best language for this kind of tasks, at least in my humble opinion.

For instance, a few months ago I have described an experiment I did with the Python meta object protocol, in order to change how the object system work, and replace multiple inheritance with traits. Even if in Python it is possible to customize the object system, I do not thing the approach is optimal, because changing the semantics without changing the syntax does not feel right. In Scheme I could have implemented the same with a custom syntax and in a somewhat less magical way. I am interested with this kind of experiments, even if I will never use them in production code, and I use Scheme in preference for such purposes.

25.1 Writing your own programming language

The major interest of Scheme macros for me lies in the fact that they *enable every programmer to write her own programming language*. I think this is a valuable thing. Anybody who has got opinions about language design, or about how an object system should should work, or questions like "what would a language look like if it had feature X?", can solve his doubts by implementing the feature with macros.

Notice that I recognize that perhaps not everybody should design its own programming language, and that certainly not everybody should *distribute* its own personal language. Nevertheless, I think everybody can have opinions about language design. Experimenting with macrology can help to put to test such opinions and to learn something.

The easiest approach is to start from a Domain Specific Language (DSL), which does not need to be a fully grown programming language. For instance, in the Python world everybody is implementing his own templating language to generate web pages. In my opinion, this a good thing *per se*, the problem is that everybody is distributing his own language so that there is a bit of anarchy.

Even for what concerns fully grown programming languages we see nowadays an explosion of new languages, especially for the Java and the .NET platforms, since it is relatively easy to implement a new language there. However, it still takes a substantial amount of work.

On the other hand, writing a custom language embedded in Scheme by means of macros is much easier. I see Scheme as an excellent platform for implementing languages and experimenting with new ideas.

There is a quote of Ian Bicking about Web frameworks which struck me:

Sometimes Python is accused of having too many web frameworks. And it's true, there are a lot. That said, I think writing a framework is a useful exercise. It doesn't let you skip over too much without understanding it. It removes the magic. So even if you go on to use another existing framework (which I'd probably advise you do), you' ll be able to understand it better if you've written something like it on your own.

You can the replace the words "web framework" with "programming language" and the quote still makes sense. You should read my *Adventures* in this spirit: the ambition of the series is

to give to the readers the technical competence to write small Scheme-embedded languages by means of macros. Even if you are not going to design your own language, macros will help you to understand how languages work.

Personally I am interested in the technical competence, *I do not want to write a new language*. There are already lots of languages out there, and writing a real language is a lot of grunt work, because it means writing debugging tools, good error messages, wondering about portability, interacting with an user community, et cetera et cetera.

25.2 Recursive macros with accumulators

The goal of learning macros well enough to implement a programming language is an ambitious one; it is not something I can attain in one episode of the Adventures, nor in six. However, one episode is enough to explain at least one useful technique which is commonly used in Scheme macrology and which is good to know in order to reach our final goal, in time.

The technique I will discuss in this episode is writing recursive macros with accumulators. In Scheme it is common to introduce an auxiliary variable to store a value which is passed in a loop - we discussed it in episode 6 when talking about tail call optimization: the same trick can be used in macros, at expand-time instead that at run-time.

In order to give an example I will define a macro *cond minus* (`cond-`) which works like `cond`, but with less parenthesis. Here is an example:

```
(cond-
   cond-1? return-1
   cond-2? return-2
       ...
   else return-default)
```

should expand to:

```
(cond
  (cond-1? return-1)
```

```
  (cond-2? return-2)
        ...
  (else return-default))
```

Here is a solution, which makes use of an accumulator and of an auxiliary macro `cond-aux`:

```
(def-syntax cond-aux
  (syntax-match ()
   (sub (cond-aux (acc ...))
        #'(cond acc ...))
   (sub (cond-aux (acc ...) x1)
     #'(syntax-violation 'cond- "Mismatched pairs" '(acc ... x1) 'x1))
   (sub (cond-aux (acc ...) x1 x2 x3 ...)
     #'(cond-aux (acc ... (x1 x2)) x3 ...))
   ))

(def-syntax (cond- x1 x2 ...)
  (cond-aux () x1 x2 ...))
```

The code above should be clear. The auxiliary macro `cond-aux` is recursive: it works by collecting the arguments `x1, x2, ..., xn` in the accumulator `(acc ...)`. If the number of arguments is even, at some point we end up having collected all the arguments in the accumulator, which is then expanded into a standard conditional; if the number of arguments is even, at some point we end up having collected all the arguments except one, and a `"Mismatched pairs"` exception is raised. The user-visible macro `cond-` just calls `cond-aux` by setting the initial value of the accumulator to `()`. The entire expansion and error checking is made at compile time. Here is an example of usage:

```
> (let ((n 1))
    (cond- (= n 1) ; missing a clause
      (= n 2) 'two
      (= n 3) 'three
      else 'unknown))
Unhandled exception:
Condition components:
  1. &who: cond-
  2. &message: "Mismatched pairs"
  3. &syntax:
      form: (((= n 1) (= n 2)) ('two (= n 3)) ('three else) 'unknown)
      subform: 'unknown
```

25.3 A trick to avoid auxiliary macros

I have nothing against auxiliary macros, however sometimes you may want to keep all the code in a single macro. This is useful if you are debugging a macro since an auxiliary macro is usually not exported. The trick is to introduce a literal to defined the helper macro inside the main macro. Here is how it would work in this example:

```
(define-syntax cond-
 (syntax-match (aux)
  (sub (cond- aux (acc ...))
    (cond acc ...))
  (sub (cond- aux (acc ...) x1)
    (syntax-violation 'cond- "Mismatched pairs" '(acc ... x1) 'x1))
  (sub (cond- aux (acc ...) x1 x2 x3 ...)
    (cond- aux (acc ... (x1 x2)) x3 ...))
  (sub (cond- x1 x2 ...)
    (cond- aux () x1 x2 ...))))
```

If you do not want to use a literal identifier, you can use a literal string instead:

```
(define-syntax cond-
 (syntax-match ()
  (sub (cond- "aux" (acc ...))
    (cond acc ...))
  (sub (cond- "aux" (acc ...) x)
    (syntax-violation 'cond- "Mismatched pairs" '(acc ... x) 'x))
  (sub (cond- "aux" (acc ...) x1 x2 x3 ...)
    (cond- "aux" (acc ... (x1 x2)) x3 ...))
  (sub (cond- x1 x2 ...)
    (cond- "aux" () x1 x2 ...))))
```

These tricks are quite common in Scheme macros: we may even call them design patterns. In my opinion the best reference detailing these techniques and others is the Syntax-Rules Primer for the Merely Eccentric, by Joe Marshall. The title is a play on the essay An Advanced Syntax-Rules Primer for the Mildly Insane by Al Petrofsky.

Marshall's essay is quite nontrivial, and it is intended for expert Scheme programmers. On the other hand, it is child play compared to Petrofsky's essay, which is intended for foolish Scheme wizards ;)

CHAPTER

TWENTYSIX

MACROS TAKING MACROS AS ARGUMENTS

There is no limit to the sophistication of macros: for instance, it is possible to define higher order macros, i.e. macros taking other macros as arguments or macros expanding into other macros. Higher order macros allow an extremely compact and elegant programming style; on the other hand, they are exposed to the risk of making the code incomprehensible and pretty hard to debug. I have already shown an example of macro expanding into a macro transformer in episode 22, and explained the intricacies of the tower of meta-levels; in this episode instead I will consider a much simpler class of higher order macros, macros taking macros as arguments. Moreover, I will spend some time discussing the philosophy of Scheme and explaining the real reason why there are so many parentheses.

26.1 Scheme as an unfinished language

Most programmers are used to work with a *finished* language. With *finished*, I mean that the language provides not only a basic core of functionalities, but also a toolbox of ready-made solutions making the life of the application programmer easier. Notice that here I am not considering the quality of the library coming with the language (which is extremely important of course) but language-level features, such as providing syntactic sugar for common use cases.

As a matter of fact, developers of the XXIth century take for granted a *lot* of language features that were uncommon just a few years ago. This is particularly true for developers working with dynamic languages, which are used to features like built-in support for regular expressions, a standard object system with a Meta Object Protocol, a Foreign Function Interface, a sockets/networking interface, support for concurrency via microthread *and* native threads *and* multiprocesses and more; nowadays even Javascript has list comprehension and generators!

Modern finished languages spoil the programmer, and this is the reason why they are so much popular. Of course not all finished languages are equivalent, and some are more powerful and/or easier to use than others. Some programmers will prefer Python over Java, others will prefer Ruby, or Scala, or something else, but the concept of finished language should be clear. On the other hand Scheme, at least as specified in the R6RS standard - I am not talking about concrete implementations here - is missing lots of the features that modern languages provide out the box. Compared to the expectations of the modern developer, Scheme feels very much like an unfinished language.

I think the explanation for the current situation is more historical and social than technical. On one hand, a lot of people in the Scheme world want Scheme to stay the way it is, i.e. a language for language experimentations and research more than a language for enterprise work (for instance a standard object system would effectively kill the ability to experiment with other object systems and this is not wanted). On the other hand, the fact that there are so many implementations of Scheme makes difficult/impossible to specify too much: this the reason why there are no standard debugging tools for Scheme, but only implementation-specific ones.

Even if the Scheme language has been left unfinished - it does not matter if by choice or out of necessity - it has been equipped with a built-in mechanism enabling the user to finish the language according to his/her preferences. Such a mechanism is of course the mechanism of macros. Actually, one of the main use of macros is to fill out the deficiencies left out by the standard. Most people nowadays prefer to have ready-made solutions, because they have deadlines, projects to complete and no time nor interest in writing things that should be made by language designers, so they dismiss Scheme immediately after having having read the standard specification.

However, one should make a distinction: while it is true that Scheme - in the sense of the language specified by the R6RS standard - is unfinished, concrete implementations of Scheme tends to be much more complete. If you give up portability and you marry a specific implementations you get all the benefit of a "finished" language. Consider for instance PLT Scheme, or Chicken Scheme, which are two big Scheme implementations: they have support for every language-level feature you get in a mainstream language and decent size libraries so that they are perfectly usable (and used) for practical tasks you could do with Python or Ruby or even a compiled language. Another option if you want to use Scheme in an enterprise context is to use a Scheme implementation running on the Java virtual machine (SISC, Kawa ...) or on the .NET platform (IronScheme). Alternatively, you could use a Scheme-like language such as Clojure, developed by Rich Hickey.

Clojure runs on the Java Virtual Machine, it is half lisp and half Scheme, it has a strong functional flavour in it, and an interesting support to concurrency. It also shares the following caracteristics with Python/Ruby/Perl/...:

1. it is a one-man language (i.e. it is not a comprimise language made by a committee) with a clear philosophy and internal consistency;
2. it is language made from scratch, with no preoccupations of backward compatibility;
3. it provides special syntax/libraries for common operations (syntax conveniences) that would never enter in the Scheme standard.

Such caracteristics make Clojure very appealing. However, personally I have no need to interact with the Java platform professionally (and even there I would probably choose Jython over Clojure for reason of greater familiarity) so I have not checked out Clojure and I have no idea about it except what you can infer after reading its web site. If amongst my readers there is somebody with experience in Clojure, please feel free to add a comment to this article. I personally am using Scheme since I am interested in macrology and no language in existence can beat Scheme in this respect. Also, I am using for Scheme for idle speculation and not to get anything done ;-)

26.2 Two second order macros to reduce parentheses

A typical example of idle speculation is the following question: can we find a way to reduce the number of parentheses required in Scheme? Finding tricks for reducing parentheses is a pointless exercise per se, but it gives a reason to teach a few other macro programming techniques - in particular second order macros taking macros as arguments - and to explain why parentheses are actually good and should not be removed.

In episode 25 I defined a recursive `cond-` macro taking less parentheses than a regular `cond`, using an accumulator. Here I will generalize that approach, by abstracting the accumulator functionality into a second order macro, called `collecting-pairs`, which takes as input another macro and a sequence of arguments, and calls the input macro with the arguments grouped in pairs. That makes it possible to call with less parentheses any macro of the form `(macro expr ... (a b) ...)`, by calling it as `(collecting-pairs (macro expr ...) a b ...)`.

Here is the code implementing `collecting-pairs`:

```
(def-syntax collecting-pairs
  (syntax-match ()
    (sub (collecting-pairs (name arg ...) x1 x2 ...)
     #'(collecting-pairs "helper" (name arg ...) () x1 x2 ...))
    (sub (collecting-pairs "helper" (name arg ...) (acc ...))
     #'(name arg ... acc ...))
    (sub (collecting-pairs "helper" (name arg ...) (acc ...) x)
     #'(syntax-violation 'name "Mismatched pairs" '(name arg ... acc ... x) 'x))
    (sub (collecting-pairs "helper" (name arg ...) (acc ...) x1 x2 x3 ...)
     #'(collecting-pairs "helper" (name arg ...) (acc ... (x1 x2)) x3 ...))
    ))
```

`collecting-pairs` can be used with many syntactic expressions like `cond`, `case`, `syntax-rules`, et cetera. Here is an example with the case expression:

```
> (collecting-pairs (case 1)
      (1) 'one
      (2) 'two
      (3) 'three
      else 'unknown))
one
```

Using a second order macro made us jump up one abstraction level, by encoding the accumulator trick into a general construct that can be used with a whole class of `cond`-style forms. However, `collecting-pairs` cannot do anything to reduce parentheses in `let`-style forms. To this aim we can introduce a different second order macro, such as the following "colon" macro:

```
(def-syntax : ; colon macro
  (syntax-match ()
    (sub (: let-form e); do nothing
         #'e)
    (sub (: let-form e1 e2)
```

```
        (syntax-violation ': "Odd number of arguments" #'let-form))
   (sub (: let-form patt value rest ... expr)
        #'(let-form ((patt value)) (: let-form rest ... expr))
        (identifier? #'let-form)
        (syntax-violation ': "Not an identifier" #'let-form))
   ))
```

The colon macro expects as argument another macro, the `let-form`, which can be any binding macro such that `(let-form ((patt value) ...) expr)` is a valid syntax. For instance `(let ((name value) ...) expr)` can be rewritten as `(: let name value ... expr)`, by removing four parentheses. Here is a test with `let*`:

```
(test "colon-macro" (: let* x 1 y x (+ x y)) 2)
```

The latest version of the `aps` package provides a colon `:` form in the `(aps lang)` module. In the following Adventures I will never use `collecting-pairs` and `:` since I actually like parentheses. The reason is that parens make it easier to write macros with pattern matching techniques, as I argue in the next paragraph.

26.3 The case for parentheses

Paren-haters may want to use `collecting-pairs` and the colon macro to avoid parentheses. They may even go further, and rant that the basic Scheme syntax should require less parentheses. However, that would be against the Scheme philosophy: according to the Scheme philosophy a programmer should not write code, he should write macros writing code for him. In other words, automatic code generation is favored over manual writing.

When writing macros, it is much easier to use a conditional with more parentheses like `cond` than a conditional with less parentheses like `cond-`. The parentheses allows you to group expressions in group that can be repeated via the ellipsis symbol; in practice, you can write things like `(cond (cnd? do-this ...) ...)` which cannot be written with `cond-`. On the other hand, different languages adopt different philosophies; for instance Paul Graham's Arc uses less parentheses. This is possible since it does not provide a macro system based on pattern matching (which is a big *minus* in my opinion).

Is it possible to have both a syntax with few parentheses for writing code manually and a syntax with many parentheses for writing macros? Clearly the answer is yes: the price to pay is to double the constructs of the language. Python is an example of such a language with a two-level syntax: it provides both a simple syntax, limited but able to cover the most common case, and a fully fledged syntax, giving all the power you need, which however is used rarely. For instance, here a table showing some of the most common syntactic sugar used in the Python language:

Simplified syntax	Full syntax
obj.attr	getattr(obj, 'attr')
x + y	x.__add__(y)
c = C()	c = C.__new__(C); c.__init__()

In principle, the Scheme language could follow exactly the same route, by providing syntactic sugar for the common cases and a low level syntax for the general case. For instance, in the case of the conditional syntax, we could have a fully parenthesized `__cond__` syntax for usage in macros and `cond` syntax with less parens for manual usage. That, in theory: in practice Scheme only provides the low level syntax, leaving to the final user the freedom (and the burden) of implementing his preferred high level syntax. Since syntax is such a subjective topic, in practice I think it is impossible for a language designed by a committee to converge on an high level syntax. This is a consequence of the infamous bikeshed effect.

The bikeshed effect is typical of any project designed by a committee: when it comes to proposing advanced functionalities that very few can understand, it is easy to get approval from the larger community. However, when it comes to simple functionality of common usage, everybody has got a different opinion and it is practically impossible to get anything approved at all.

To avoid that, the standard does not provide directly usable instruments: instead, it provides general instruments which are intended as building blocks on that of which everybody can write the usable abstractions he/she prefers. On the other hand Lisp-like languages designed by a BDFL (like Arc and Clojure) provide a high level syntax, which is the one the BDFL like. You may try it and see if you like it. Good luck!

CHAPTER

TWENTYSEVEN

SYNTAX OBJECTS

Scheme macros - as standardized in the R6RS document - are built over the concept of *syntax object*. The concept is peculiar to Scheme and has no counterpart in other languages (including Common Lisp), therefore it is worth to spend some time on it.

A *syntax-object* is a kind of enhanced *s*-expression: it contains the source code as a list of symbols and primitive values, but also additional informations, such as the name of the file containing the source code, the position of the syntax object in the file, a set of marks to distinguish identifiers according to their lexical context, and more.

The easiest way to get a syntax object is to use the syntax quoting operation, i.e. the `syntax` (`#'`) symbol you have seen in all the macros I have defined until now. Consider for instance the following script, which displays the string representation of the syntax object `#'1`:

```
$ cat x.ss
(import (rnrs))
(display #'1)
```

If you run it under PLT Scheme you will get

```
$ plt-r6rs x.ss
#<syntax:/home/micheles/Dropbox/gcode/artima/scheme/x.ss:2:11>
```

In other words, the string representation of the syntax object `#'1` contains the full pathname of the script and the line number/column number where the syntax object appears in the source code. Clearly this information is pretty useful for tools like IDEs and debuggers. The internal implementation of syntax objects is not standardized at all, so that you get different informations in different implementations. For instance Ikarus gives

```
$ ikarus --r6rs-script x.ss
#<syntax 1 [char 28 of x.ss]>
```

i.e. in Ikarus syntax objects do not store line numbers, they just store the character position from the beginning of the file. If you are using the REPL you will have less information, of course, and even more implementation-dependency. Here are a few example of syntax objects obtained from syntax quoting:

```
> #'x ; convert a name into an identifier
#<syntax x>
> #''x ; convert a literal symbol
#<syntax 'x>
> #'1 ; convert a literal number
#<syntax 1>
> #'"s" ; convert a literal string
#<syntax "s">
> #''(1 "a" 'b) ; convert a literal data structure
#<syntax '(1 "a" 'b)>
```

Here I am running all my examples under Ikarus; your Scheme system may have a slightly different output representation for syntax objects.

In general `#'` can be "applied" to any expression:

```
> (define syntax-expr #'(display "hello"))
> syntax-expr
#<syntax (display "hello")>
```

It is possible to extract the *s*-expression underlying the syntax object with the `syntax->datum` primitive:

```
> (equal? (syntax->datum syntax-expr) '(display "hello"))
#t
```

Different syntax-objects can be equivalent: for instance the improper list of syntax objects `(cons #'display (cons #'"hello" #'()))` is equivalent to the syntax object `#'(display "hello")` in the sense that both corresponds to the same datum:

```
> (equal? (syntax->datum (cons #'display (cons #'"hello" #'())))
          (syntax->datum #'(display "hello")))
#t
```

The `(syntax )` macro is analogous to the `(quote )` macro. Mreover, there is a `quasisyntax` macro denoted with `` #` `` which is analogous to the `quasiquote` macro (`` ` ``). In analogy to the operations comma (`,`) and comma-splice (`,@`) on regular lists, there are two operations `unsyntax #,` (*sharp comma*) e `unsyntax-splicing #,@` (*sharp comma splice*) on lists and improper lists of syntax objects.

Here is an example using sharp-comma:

```
> (let ((user "michele")) #`(display #,user))
(#<syntax display> "michele" . #<syntax ()>)
```

Here is an example using sharp-comma-splice:

```
> (define users (list #'"michele" #'"mario"))
> #`(display (list #,@users))
```

```
(#<syntax display>
(#<syntax list> #<syntax "michele"> #<syntax "mario">) . #<syntax ()>)
```

Notice that the output - in Ikarus - is an improper list. This is somewhat consistent with the behavior of usual quoting: for usual quoting `'(a b c)` is a shortcut for `(cons* 'a 'b 'c '())`, which is a proper list, and for syntax-quoting `#'(a b c)` is equivalent to `(cons* #'a #'b #'c #'())`, which is an improper list. The `cons*` operator here is a R6RS shortcut for nested conses: `(cons* w x y z)` is the same as `(cons w (cons x (cons y z)))`.

However, the result of a quasi quote interpolation is very much *implementation-dependent*: Ikarus returns an improper list, but other implementations returns different results; for instance Ypsilon returns a proper list of syntax objects whereas PLT Scheme returns an atomic syntax object. The lesson here is that you cannot rely on properties of the inner representation of syntax objects: what matters is the code they correspond to, i.e. the result of `syntax->datum`.

It is possible to promote a datum to a syntax object with the `datum->syntax` procedure, but in order to do so you need to provide a lexical context, which can be specified by using an identifier:

```
> (datum->syntax #'dummy-context '(display "hello"))
#<syntax (display "hello")
```

(the meaning of the lexical context in `datum->syntax` is tricky and I will go back to that in a future episode).

27.1 What `syntax-match` really is

`syntax-match` is a general utility to perform pattern matching on syntax objects; it takes a syntax object in output and returns a syntax object in output. Here is an example of a simple transformer based on `syntax-match`:

```
> (define transformer
    (syntax-match ()
      (sub (name . args) #'name))); return the name as a syntax object

> (transformer #'(a 1 2 3))
#<syntax a>
```

For convenience, `syntax-match` also accepts a second syntax `(syntax-match x (lit ...) clause ...)` to match syntax expressions directly. This is more convenient than writing `((syntax-match (lit ...) clause ...) x)`. Here is a simple example:

```
> (syntax-match #'(a 1 2 3) ()
   (sub (name . args) #'args)); return the args as a syntax object
#<syntax (1 2 3)>
```

Here is an example using `quasisyntax` and `unsyntax-splicing`:

```
> (syntax-match #'(a 1 2 3) ()
    (sub (name . args) #`(name #,@#'args)))
(#<syntax a> #<syntax 1> #<syntax 2> #<syntax 3>)
```

As you see, it easy to write hieroglyphs if you use `quasisyntax` and `unsyntax-splicing`. You can avoid that by means of the `with-syntax` form:

```
> (syntax-match #'(a 1 2 3) ()
    (sub (name . args) (with-syntax (((a ...) #'args)) #'(name a ...))))
(#<syntax a> #<syntax 1> #<syntax 2> #<syntax 3>)
```

The pattern variables introduced by `with-syntax` are automatically expanded inside the syntax template, without need to resort to the quasisyntax notation (i.e. there is no need for `#`` `#,` `#,@`).

27.2 What macros really are

Macros are in one-to-one correspondence with syntax transformers, i.e. every macro is associated to a transformer which converts a syntax object (the macro and its arguments) into another syntax object (the expansion of the macro). Scheme itself takes care of converting the input code into a syntax object (if you wish, internally there is a `datum->syntax` conversion) and the output syntax object into code (an internal `syntax->datum` conversion).

Consider for instance a macro to apply a function to a (single) argument:

```
(def-syntax (apply1 f a)
  #'(f a))
```

This macro can be equivalently written as

```
(def-syntax apply1 (syntax-match () (sub (apply1 f a) (list #'f #'a))))
```

The sharp-quoted syntax is more readable, but it hides the underlying list representation which in some cases is pretty useful. This second form of the macro is more explicit, but still it relies on `syntax-match`. It is possible to provide the same functionality without using `syntax-match` as follows:

```
(def-syntax apply1
  (lambda (x)
    (let+ ((macro-name func arg) (syntax->datum x))
      (datum->syntax #'apply1 (list func arg)))))
```

Here the macro transformer is explicitly written as a lambda function, and the pattern matching is performed by hand by converting the input syntax object into a list and by using the list destructuring form `let+` introduced in episode 15. At the end, the resulting list is converted back to a syntax object in the context of `apply1`. Here is an example of usage:

```
> (apply1 display "hey")
hey
```

`sweet-macros` provide a convenient feature: it is possible to extract the associated transformer for each macro defined via `def-syntax`. For instance, here is the transformer associated to the `apply1` macro:

```
> (define tr (apply1 <transformer>))
> (tr #'(apply1 display "hey"))
#<syntax (display "hey")>
```

The ability to extract the underlying transformer is useful in certain situations, in particular when debugging. It can also be exploited to define extensible macros, and I will come back to this point in the future.

27.3 A nicer syntax for association lists

The previous paragraphs were a little abstract and probably of unclear utility (but what would you expect from an advanced macro tutorial? ;). Now let me be more concrete. My goal is to provide a nicer syntax for association lists (an association list is just a non-empty list of non-empty lists) by means of an `alist` macro expanding into an association list. The macro accepts a variable number of arguments; every argument is of the form `(name value)` or it is a single identifier: in this case latter case it must be magically converted into the form `(name value)` where `value` is the value of the identifier, assuming it is bound in the current scope, otherwise a run time error is raised `"unbound identifier"`. If you try to pass an argument which is not of the expected form, a compile time syntax error must be raised. In concrete, the macro works as follows:

```
(test "simple"
      (let ((a 0))
        (alist a (b 1) (c (* 2 b))))
        '((a 0) (b 1) (c 2)))

(test "with-error"
     (catch-error (alist a))
     "unbound variable")
```

Here is the implementation:

```
(def-syntax (alist arg ...)
  (with-syntax ((
      ((name value) ...)
      (map (syntax-match ()
             (sub n #'(n n) (identifier? #'n))
             (sub (n v) #'(n v) (identifier? #'n)))
           #'(arg ...)) ))
     #'(let* ((name value) ...)
         (list (list 'name name) ...))))
```

The expression `#'(arg ...)` expands into a list of syntax objects which are then transformed by the `syntax-match` transformer, which converts identifiers of the form `n` into couples of the form `(n n)`, whereas it leaves couples `(n v)` unchanged, just checking that `n` is an identifier. This is a typical use case for `syntax-match` as a list matcher inside a bigger macro. We will see other use cases in the next Adventures.

CHAPTER

TWENTYEIGHT

HYGIENIC MACROS

In episode 9 I noted that Scheme provides three major macro systems (syntax-rules, syntax-case and define-macro), yet I went on to discuss my own personal macro system, sweet-macros. The decision was motivated by various reasons. First of all, I did not want to confuse my readers by describing too many macro systems at the same time. Secondly, I wanted to make macros easier and more debuggable. Finally, sweet-macros are slightly more powerful than the other macro systems, with a better support for guarded patterns and with a few extensibility features which I have not shown yet (but I like to keep some trick under my sleeve ;).

After 19 episodes about macros, I can safely assume that my readers are not beginners anymore. It is time to have a look at the larger Scheme world and to compare/contrast sweet-macros with the other macro systems. I do not want to discuss all the macro systems in existence here, therefore I will skip a few interesting systems such as syntactic closures and explicit renaming macros. However, readers interested in alternative macro systems for Scheme should have a look at this excellent post by Alex Shinn which summarizes the current situation better than I could do. Notice that Alex is strongly biased against syntax-case and very much in favor of explicit renaming macros. The two systems are not incompatible though, and actually Larceny provides a syntax-case implementation built on top of explicit renaming macros (see also SRFI-72).

28.1 syntax-match vs syntax-rules

syntax-rules can be quite trivially defined in terms of syntax-match:

```
(def-syntax (syntax-rules (literal ...) (patt templ) ...)
  #'(syntax-match (literal ...) (sub patt #'templ) ...))
```

As you see, the main difference between syntax-rules (apart for a missing sub) is that syntax-rules automatically adds the syntax-quote #' operator to you templates. That means that you cannot use quasisyntax tricks and that syntax-rules is strictly less powerful than syntax-match. The other difference is that syntax-rules macros do not have guarded patterns; the most direct consequence is that providing good error messages for incorrect syntax is more difficult. You may learn everything you ever wanted to know about syntax-rules in the Syntax-Rules Primer for the Mildly Insane by Al Petrofsky.

28.2 `syntax-match` vs `syntax-case`

In principle, `syntax-case` could be defined in terms of `syntax-match` as follows:

```
(def-syntax syntax-case
  (syntax-match ()
   (sub (syntax-case x (literal ...) (patt guard skel) ...)
    #'(syntax-match x (literal ...) (sub patt skel guard) ...))
   (sub (syntax-case x (literal ...) (patt skel) ...)
    #'(syntax-match x (literal ...) (sub patt skel) ...))
 ))
```

In reality, `syntax-case` is a Scheme primitive and `syntax-match` is defined on top of it. So, `syntax-case` has theoretically the same power as `syntax-match`, but in practice `syntax-match` is more convenient to use.

The major syntactic difference is the position of the guard, which in `syntax-case` is positioned *before* the skeleton, whereas in `syntax-match` is positioned *after* the skeleton I did spent a lot of time thinking about the right position for the guard: I hate gratuitous breaking, but I convinced myself that the position of the guard in `syntax-case` is really broken, so I had to *fix* the issue. The problem with `syntax-case` is that while the pattern is always in the first position, you never know what is in the second position: it could be the guard *or* the template; in order to distinguish the possibilities you have to check if there is a third expression in the clause and that is annoying,

In `syntax-match` the template is *always* in the second position; if there is something in the third position, it is always a guard; moreover, there could be another expression in the clause (in the fourth position) which is used as alternative template if the guard is not satisfied. The advantage of having fixed positions is that it is easier to write higher order macros expanding to `syntax-match` transformers (`syntax-match` itself is implemented in terms of `syntax-case` where the template are not in fixed position and I had to make the implementation more complex just to cope with that).

You may learn (nearly) everything there is to know about `syntax-case` in the the book The Scheme Programming Language by Kent Dybvig.

28.3 `syntax-match` versus `define-macro`

Nowadays macros based on `define-macro` are much less used than in past because macro systems based on pattern matching are much more powerful, easier and safer to use. The R6RS specification made `syntax-case` enter in the standard and it is the preferred macro system for most implementation.

Nowadays, there is a good chance that your Scheme implementation does not provide `define-macro` out of the box, therefore you need to implement it in term of `syntax-case` (or `syntax-match`). Here is an example of such an implementation:

```
(def-syntax define-macro
  (syntax-match ()
    (sub (define-macro (name . params) body1 body2 ...)
      #'(define-macro name (lambda params body1 body2 ...)))
    (sub (define-macro name expander)
      #'(def-syntax (name . args)
          (datum->syntax #'name (apply expander (syntax->datum #'args)))))
    ))
```

The code should be clear: the arguments of the macro are converted into a regular list which is then transformed with the expander, and converted back into a syntax object in the context of the macro. `define-macro` macros are based on simple list manipulations and are very easy to explain and to understand: unfortunately, they are affected by the hygiene problem.

You can find examples of use of `define-macro` in many references; I learned it from Teach Yourself Scheme in Fixnum Days by Dorai Sitaram.

28.4 The hygiene problem

If you have experience in Common Lisp or other Lisp dialects, you will have heard about the problem of hygiene in macros, a.k.a. the problem of *variable capture*. As Paul Graham puts it, *errors caused by variable capture are rare, but what they lack in frequency they make up in viciousness*. The hygiene problem is the main reason why `define-macro` is becoming less and less used in the Scheme world. PLT Scheme has being deprecating it for years and nowadays even Chicken Scheme, which traditionally used `define-macro`, has removed it from the core, by using hygienic macros instead: this is the reason why the current Chicken (Chicken 4) is called "hygienic Chicken".

You can find good discussions of the hygiene problem in Common Lisp in many places; I am familiar with Paul Graham's book On Lisp which I definitively recommend: chapter 9 on variable capture has influenced this section. Another good reference is the chapter about syntax-case - by Kent Dybvig - in the book Beautiful Code. Here I will give just a short example exhibiting the problem, for the sake of the readers unfamiliar with it.

Consider this "dirty" definition of the `for` loop:

```
(define-macro (dirty-for i i1 i2 . body)
  `(let ((start ,i1) (stop ,i2))
```

```
     (let loop ((,i start))
       (unless (>= ,i stop) ,@body (loop (+ 1 ,i))))))
```

Superficially define-macro looks quite similar to def-syntax, except that in the macro body you need to a add a comma in front of each macro argument argument. Internally, however, macros based on define-macro are completely different. In particular, they are not safe under variable capture and that may cause surprises. For instance, code such as

```
> (let ((start 42))
   (dirty-for i 1 3 (display start) (newline)))
1
1
```

prints the number 1 (twice) and not the number 42!

The reason is clear if you expand the macro (notice the if you implement define-macro in terms of syntax-match then syntax-expand still works):

```
> (syntax-expand (dirty-for i 1 3 (display start) (newline)))
(let ((start 1) (stop 3))
  (let loop ((i start))
    (unless (>= i stop) (display start) (newline)
      (loop (+ 1 i)))))
```

Since the inner variable start is shadowing the outer variable start, the number 1 is printed instead of the number 42. The problem can be solved by introducing unique identifiers in the macro by means of gensym (gensym is not in the R6RS standard, but in practice every Scheme implementation has it; for convenience I have included it in my (aps compat) compatibility library).

The dirty-for macro can be improved to use gensym for every variable which is internally defined (and it is not a macro argument):

```
(define-macro (less-dirty-for i i1 i2 . body)
  (let ((start (gensym)) (stop (gensym)) (loop (gensym)))
   `(let ((,start ,i1) (,stop ,i2))
      (let ,loop ((,i ,start))
        (unless (>= ,i ,stop) ,@body (,loop (+ 1 ,i)))))))
```

```
> (let ((start 42))
   (less-dirty-for i 1 3 (display start) (newline)))
42
42
```

less-dirty-for works because all internal variables have now unique names that cannot collide with existing identifiers by construction. You can see the names used internally by invoking syntax-expand (notice that by construction such names change every time you expand the macro):

```
> (syntax-expand (less-dirty-for i 1 3 (display i))); in Ikarus
(let ((#{g0 |K!ZoUGmIIl%SrMfI|} 1) (#{g1 |qecoGeEAOv0R8$%0|} 3))
  (let #{g2 |kD?xfov61j0$G1=M|} ((i #{g0 |K!ZoUGmIIl%SrMfI|}))
    (unless (>= i #{g1 |qecoGeEAOv0R8$%0|}) (display i)
     (#{g2 |kD?xfov61j0$G1=M|} (+ 1 i)))))
```

Unfortunately `less-dirty-for` is not really clean. `gensym` cannot do anything to solve the *free-symbol capture* problem (I am using Paul Graham's terminology here).

The problem is that identifiers used (but not defined) in the macro have the scope of expanded code, not the scope of the original macro: in particular, if the outer scope in expanded code redefines the meaning of an identifier used internally, the macro will not work as you would expect. Consider for instance the following expression:

```
> (let ((unless 'unless))
   (less-dirty-for i 1 3 (display i)))
```

If you try to run this, your system will go into an infinite loop!

The problem here is that shadowing `unless` in the outer scope affects the inner working of the macro (I leave as an exercise to understand what exactly is going on). As you can easily see, this kind of problem is pretty tricky to debug: in practice, it means that the macro user is forced to know all the identifiers that are used internally by the macro.

On the other hand, if you use hygienic macros, all the subtle problems I described before simply disappear and you can write a clean for loop just as it should be written:

```
(def-syntax (clean-for i i1 i2 body1 body2 ...)
  #'(let ((start i1) (stop i2))
     (let loop ((i start))
       (unless (>= i stop) body1 body2 ... (loop (+ 1 i))))))
```

Everything works fine with this definition, and the macro looks even better, with less commas and splices ;-)

That's all for today. The next episode will discuss how to break hygiene on purpose, don't miss it!

CHAPTER

TWENTYNINE

BREAKING HYGIENE

In the previous episode I said that hygienic macros are good, since they solve the variable capture problem. However, purely hygienic macros introduce a problem of their own, since they make it impossible to introduce variables at all. Consider for instance the following trivial macro:

```
(def-syntax (define-a x)
  #'(define a x))
```

`(define-a x)` *apparently* expand to `(define a x)`, so you may find the following surprising:

```
> (define-a 1)
> a
Unhandled exception
 Condition components:
   1. &undefined
   2. &who: eval
   3. &message: "unbound variable"
   4. &irritants: (a)
```

Why is the variable `a` not bound to 1? The problem is that hygienic macros *never introduce identifiers implicitly*. Auxiliary names introduced in a macro *are not visible outside* and the only names which enter in the expansion are the ones we put in. A mechanism to introduce identifiers, i.e. a mechanism to break hygiene, is needed if you want to define binding forms.

29.1 `datum->syntax` revisited

Scheme has a builtin mechanism to break hygiene, and we already saw it: it is the `datum->syntax` utility which converts literal objects (*datums*) into syntax objects. I have shown `datum->syntax` at work in episodes 27 and 28 : it was used there to convert lists describing source code into syntax objects. A more typical use case for `datum->syntax` is to turn symbols into proper identifiers. Such identifiers can then be introduced in macros and made visible to expanded code.

In order to understand the mechanism, you must always remember that identifiers in Scheme - in the technical sense of objects recognized by the `identifier?` predicate - are not just raw symbols, they are syntax objects with lexical information attached to them. If you want to turn a raw symbol into an identifier you must add the lexical information to it, and this is done by copying the lexical information coming from the context object in `datum->syntax`.

For instance, here is how you can "fix" the macro `define-a`:

```
(def-syntax (define-a* x)
  #`(define #,(datum->syntax #'define-a* 'a) x))
```

The symbol `'a` here is being promoted to a *bona fide* identifier, by adding to it the lexical context associated to the macro name. You can check that the identifier `a` is really introduced as follows:

```
> (define-a* 1)
> a
1
```

A more realistic example is to use `syntax->datum` to build new identifiers from strings. For that purpose I have added an `identifier-append` utility in my `(aps lang)` library, defined as follow:

```
;; take an identifier and return a new one with an appended suffix
(define (identifier-append id . strings)
  (define id-str (symbol->string (syntax->datum id)))
  (datum->syntax id (string->symbol (apply string-append id-str strings))))
```

Here is a simple `def-book` macro using `identifier-append`:

```
(def-syntax (def-book name title author)
  (with-syntax (
     (name-title (identifier-append #'name "-title"))
     (name-author (identifier-append #'name "-author")))
     #'(begin
         (define name (vector title author))
         (define name-title (vector-ref name 0))
         (define name-author (vector-ref name 1)))))
```

`def-book` here is just as an example of use of `identifier-append`, it is not as a recommended pattern to define records. There are much better ways to define records in Scheme, as we will see in part VI of these Adventures.

Anyway, `def-book` works as follows. Given a single identifier `name` and two values it introduces three identifiers in the current lexical scope: `name` (bound to a vector containing the two values), `name-title` (bound to the first value) and `name-author` (bound to the second value).

```
> (def-book bible "The Bible" "God")
> bible
```

```
#("The Bible" "God")
> bible-title
"The Bible"
> bible-author
"God"
```

29.2 Playing with the lexical context

The lexical context is just the set of names which are visible to an object in a given lexical position in the source code. Here is an example of a lexical context which is particularly restricted:

```
#!r6rs
(library (experimental dummy-ctxt)
 (export dummy-ctxt)
 (import (only (rnrs) define syntax))
 (define dummy-ctxt #'here)
 )
```

The identifier `#'here` only sees the names `define`, `syntax` and `dummy-ctxt`: this is the lexical context of any object in its position in the source code. Had we not restricted the import, the lexical context of `#'here` would have been the entire `rnrs` set of identifiers. We can use `dummy-ctxt` to expand a macro into a minimal context. Here is an example of a trivial macro expanding into such minimal context:

```
> (import (experimental dummy-ctxt))
> (def-syntax expand-to-car
   (lambda (x) (datum->syntax dummy-ctxt 'car)))
```

The macro `expand-to-car` expands to a syntax object obtained by attaching to the symbol `'car` the lexical context `dummy-ctxt`. Since in such lexical context the built-in `car` is not defined, the expansion fails:

```
> (expand-to-car)
 Unhandled exception
  Condition components:
    1. &undefined
    2. &who: eval
    3. &message: "unbound variable"
    4. &irritants: (car)
```

A similar macro `expand-to-dummy-ctxt` instead would succeed since `dummy-ctxt` is bound in that lexical context:

```
> (def-syntax expand-to-dummy-ctxt
    (lambda (x) (datum->syntax dummy-ctxt 'dummy-ctxt)))
```

```
>  (expand-to-dummy-ctxt)
#<syntax here [char 115 of /home/micheles/gcode/scheme/aps/dummy-ctxt.sls]>
```

In the definition of `define-macro` I gave in episode 28 I used the name of the defined macro as lexical context. The consequence of this choice is that `define-macro` style macros are expanded within the lexical context of the code where the macro is invoked. For instance in this example

```
> (let ((x 42))
   (define-macro (m) 'x) ; (m) should expand to 42
   (let ((x 43))
     (m)))
43 ; surprise!
```

`(m)` expand to 43 since in the lexical context where the macro is invoked `x` is bound to 43. However, this behavior is quite surprising, and most likely not what it is wanted. This is actually another example of the free-symbol capture problem. It should be clear that the capture comes from expanding the macro in the macro-call context, not in the macro-definition context.

29.3 Hygienic vs non-hygienic macro systems

Understanding non-hygienic macros is important if you intend to work in the larger Lisp world. In the scheme community everybody thinks that hygiene is an essential feature and all major Scheme implementations provide hygienic macros; nevertheless, in the rest of the world things are different.

For instance, Common Lisp does not use hygienic macros and it copes with the variable capture problem by using `gensym`; the free symbol capture problem is not solved, but it is extremely rare, because Common Lisp has multiple namespaces and a package system.

The hygiene problem is more serious in Lisp-1 dialects like the newborns Arc and Clojure. Arc macros behave just like `define-macro` and are fully unhygienic, whereas Clojure macros are rather special, being nearly hygienic. In particular Clojure macros are not affected by the free-symbol capture problem:

```
user=> (defmacro m[x] `(list ~x))
#'user/m
user=> (let [list 1] (m 2))
(2)
```

The reason is that Clojure is smart enough to recognize the fully qualified `list` object appearing at macro definition time (`clojure.core/list`) as a distinct object from the local variable `list` bound to the number 1. Moreover, the ordinary capture problem can be solved with `gensym` or even cooler feature, automatic gensyms (look at the documentation of the

syntax-quote reader macro if you want to know more). Speaking as a non-expert, Clojure macros seem to fare pretty well with respect to the hygiene issue.

It is worth mentioning that if you use a package system (like in Common Lisp) or a namespace system (like in Clojure) in practice variable capture becomes pretty rare. In Scheme instead, which uses a module system, hygiene is essential: if you are writing a module containing macros which can be imported and expanded in an unknown lexical scope, in absence of hygiene you could introduce name clashes impossible to foresee in advance, and that could be solved only by the final user, which however will likely be ignorant of how your library works.

This is why in Scheme the macro expansion is not literally inserted in the original code, and a lot of magic takes place to avoid name clashes. In practice, the implementation of Scheme macros takes care of distinguishing the introduced identifiers with some specific mechanism (it could be based on marking the names, or on explicit renaming). As a consequence, the mechanism of macro expansion is less simple to explain: you cannot just cut and paste the result of the expansion in your original code.

Personally I have made my mind up and I am in the pro-hygiene camp now. I should admit that for a long time I have been in the opposite camp, preferring the simplicity of `define-macro` over the complexity of `syntax-case`. It turns out I was wrong. The major problem of `syntax-case` is a cosmetic one: it looks very complex and cumbersome to use, but that can be easily solved by providing a nicer API - which I did with `sweeet-macros`. Actually I have been able to use `sweet-macros` for twenty episodes without explaining the intricacies of the hygienic expansion.

Having attended to a talk on the subject at the EuroLisp Symposium, I will mention here that there are ways to implement hygienic macros on top of `defmacro` in Common Lisp *portably*. Therefore there is no technical reason why hygienic macros are not widespread in the whole Lisp world, just a matter of different opinions on the importance of the problem and the different tradeoffs. I believe that eventually all Lisp dialects will start using hygienic macros, but that could take decades, because of inertia and backward-compatibility concerns.

CHAPTER

THIRTY

COMPARING IDENTIFIERS

This is the last episode of part V of my Adventures. In the latest episodes I have discussed various technicalities of Scheme macros, such as the concepts of syntax object, hygiene and lexical context. There is still an important subject to be discuss in order to become really proficient with Scheme macros: *identifier equality*. Equality of identifiers is one of the trickiest things in Scheme.

First of all, identifier equality is a compile-time concept which has nothing to do with the run-time concept of equality between variables. Identifiers are not variables: they are syntax objects with an underlying symbol and an underlying lexical context, which are known statically at compile time. It is possible to know if an identifier is bound or not at compile-time, but the value the identifier will be bound to at run-time is (in general) unknown.

Secondly, there is not a single concept of identifier equality, but different definitions are possible. In this episode I will discuss three different predicates to compare identifiers: `symbol-identifier=?`, `bound-identifier=?` and `free-identifier=?` (the latter two are part of the R6RS standard).

30.1 `symbol-identifier=?`

The simplest concept of identifier equality is expressed by the following `symbol-identifier=?` comparison function (for convenience, I have added the `symbol-identifier=?` precedure to the `(aps lang)` library):

```
(define (symbol-identifier=? id1 id2)
  (symbol=? (syntax->datum id1) (syntax->datum id2)))
```

Two identifiers are `symbol-identifier=?` if they are equal as symbols, once their lexical information has been stripped out.

For instance, symbol-identifier=? can be used to find duplicated names in macros defining name->value tables, such as the static-map macro I discussed in episode 22. Moreover, symbol-identifier=? can be used to reject reserved identifiers (you may need such functionality if are building a mini-language on top of Scheme and you want to reject a few identifiers as language keywords), as in the following example:

```
(def-syntax (check-reserved id)
  (syntax-violation 'check-reserved "Reserved identifier" #'id)
  (exists (cut symbol-identifier=? #'id <>) (list #'reserved1 #'reserved2))
  'non-reserved)
```

(check-reserved id) will raise a syntax-violation if id is one of the keyword reserved1 or reserved2.

30.2 bound-identifier=?

symbol-identifier=? is simple and easy to understand, but it cannot be used in all situations. Consider for instance the very first macro I wrote, in episode 9:

```
(def-syntax (multi-define (name1 name2 ...) (value1 value2 ...))
  #'(begin (define name1 value1) (define name2 value2) ...))
```

It is quite common to write macros defining multiple bindings, such as multi-define. multi-define as written does not perform any check for duplicated identifiers, so that it relies on the standard behavior of R6RS scheme, raising an error. However, the standard behavior only applies to programs and scripts, whereas the REPL is quite free to behaves differently and indeed it does in most implementations:

```
> (multi-define (a a) (1 2)); in Ikarus, Ypsilon, ...
a
2
```

(in the REPL latter definitions override previous definitions). If you are unhappy with that, you can introduce a bound-identifier=? check and raise a custom exception:

```
(def-syntax (multi-define (name1 name2 ...) (value1 value2 ...))
    #'(begin (define name1 value1) (define name2 value2) ...)
    (distinct? bound-identifier=? #'(name1 name2 ...))
    (syntax-violation 'multi-define "Found duplicated identifier in"
                      #'(name1 name2 ...)))
```

Two identifiers are equal according to bound-identifier=? only if they have the same name *and* the same marks. The name is misleading since the arguments of bound-identifier=? are not required to be bound identifiers; a better name would be strict-identifier=?.

You can check that multi-define correctly reject duplicated identifiers:

```
> (multi-define (a a) (1 2))
Unhandled exception
 Condition components:
   1. &who: multi-define
   2. &message: "Found duplicated identifier in"
   3. &syntax:
       form: (a a)
       subform: #f
```

In this simple example using `symbol-identifier=?` would work too. However this is not the geneal case. Consider for instance the following macro expanding to `multi-define`:

```
(def-syntax (multi-define2 id value)
   #'(multi-define (id id2) (value 'dummy)))
```

`multi-define2` introduces a dummy identifier `id2`. Had we defined `multi-define` in terms of `symbol-identifier=?`, calling `multi-define2` with argument `id` equal to `id2` would have generated a spurious name clash. Fortunately, since we defined `multi-define` in terms of `bound-identifier=?`, nothing bad happens:

```
> (multi-define2 id2 1)
id2
1
```

`bound-identifier=?` works in this case because the identifier `id2` introduced by the macro has different marks from the identifier `id2` coming as an argument.

`bound-identifier=?` is not good for every circumstance. Consider for instance the following variation of `multi-define`, featuring a literal keyword `as`:

```
;; not checking for duplicated identifiers here
(define-syntax multi-def
  (syntax-rules (as)
    ((multi-def (name as value) ...)
     (begin (define name value) ...))))
```

This work, but the error messages could stand some improvement. For instance, if an user misspells the infix identifier `as`, she gets a generic `"invalid syntax"` error, whereas we would like to provide a customized error message showing the misspelled literal identifier. Using `bound-identifier=?` we could try to solve the problem as follows:

```
(def-syntax  (multi-def-bad (name as_ value) ...)
  #'(begin (define name value) ...)
  (for-all (lambda (id)
             (when (not (bound-identifier=? id #'as))
               (syntax-violation
                'multi-def-bad "Offending infix syntax (expected `as')" id)))
           #'(as_ ...)))
```

Unfortunately this solution does not work at all, since it raises an error even when the `as` identifiers are spelled correctly:

```
> (multi-def-bad (x as y) (1 as 2))
Unhandled exception
 Condition components:
   1. &who: multi-def-bad
   2. &message: "Offending infix syntax (expected 'as')"
   3. &syntax:
       form: as
       subform: #f
   4. &trace: #<syntax as>
```

The reason is that `as` is not `bound-identifier=?` to `#'as`. We need a less strict comparison predicate. To this aim the Scheme standard provides another equality procedures for identifiers, `free-identifier=?`, which however is not quite right.

30.3 free-identifier=?

`free-identifier=?` is the most complicated equality predicate. I find its description in the R6RS document particularly confusing and the name is misleading since the arguments of `free-identifier=?` are not required to be free identifiers. A better name would be `lax-identifier=?`. Two identifiers are `free-identifier=?` if

1. they are both bound to the same binding and they share the same name (or they shared the same name before renaming at import time);
2. they are both unbound and they share the same name.

In implementations with full phase separation, the identifiers must also be both bound/unbound in the same phase. In all other cases the two identifiers are not `free-identifier=?`. Here is an example:

```
> (import (only (aps list-utils) range))
> (import (rename (aps list-utils) (range r)))
> (free-identifier=? #'r #'range)
#t
```

Notice that both `symbol-identifier=?` and `bound-identifier=?` would fail to recognize the identity of `range` and `r` in this case.

It is important to know about `free-identifier=?` because in macros with literal identifiers the literal identifiers are compared by using it, internally. That explain a behavior which can be quite surprising.

30.4 Literal identifiers and auxiliary syntax

Consider the macro `multi-def` defined in the previous paragraph. This works:

```
> (let ()
    (multi-def (x as 1) (y as 2))
    (list x y))
(1 2)
```

But this does not work:

```
> (let ((as 2))
     (multi-def (x as 1) (y as 2))
     (list x y))
Unhandled exception
 Condition components:
   1. &message: "invalid syntax"
   2. &syntax:
       form: (multi-def (x as 1) (y as 2))
       subform: #f
   3. &trace: #<syntax (multi-def (x as 1) (y as 2))>
```

That looks surprising, but it is not once you realize that internally literal identifiers are compared via `free-identifier=?`. In the second example `as` is bound, and therefore it is not `free-identifier=?` to the literal identifier `#'as`, which is unbound.

The recommended "solution" is to introduce at top level some dummy definitions for the literal identifiers you are going to use in your macro, and to export them. Following this policy, the R6RS document defines a set of special macros, _, `...`, `else` and =>, which lives in the global namespace and are available to all R6RS programs.

Such macros are used as auxiliary syntax in various special forms, like `cond` and `syntax-case`; for this reason they are usually called auxiliary keywords. The existence of such global variables makes it impossible to redefine them at top-level in scripts (but it can be done at the REPL); however they can be redefined locally, thus breaking the macros using the auxiliary syntax:

```
> (let ((else #f))
    (cond (else 'something)))
> ; does not return something
```

I think this is fundamentally broken: literal identifiers should be a concept internal to the macro and they should not be exported. The mistake is that the R6RS requires the literal identifiers to be matched via `free-identifier=?`, whereas they should be matched with `symbol-identifier=?`. I never understood why the editors decided to use `free-identifier=?`, perhaps because it makes it possible to rename the identifiers used as literal identifiers, a feature that looks of little utility to me. All in all, I think `free-identifier=?` is another dark corner of R6RS Scheme.

CHAPTER

THIRTYONE

INDICES AND TABLES

- *Index*
- *Search Page*