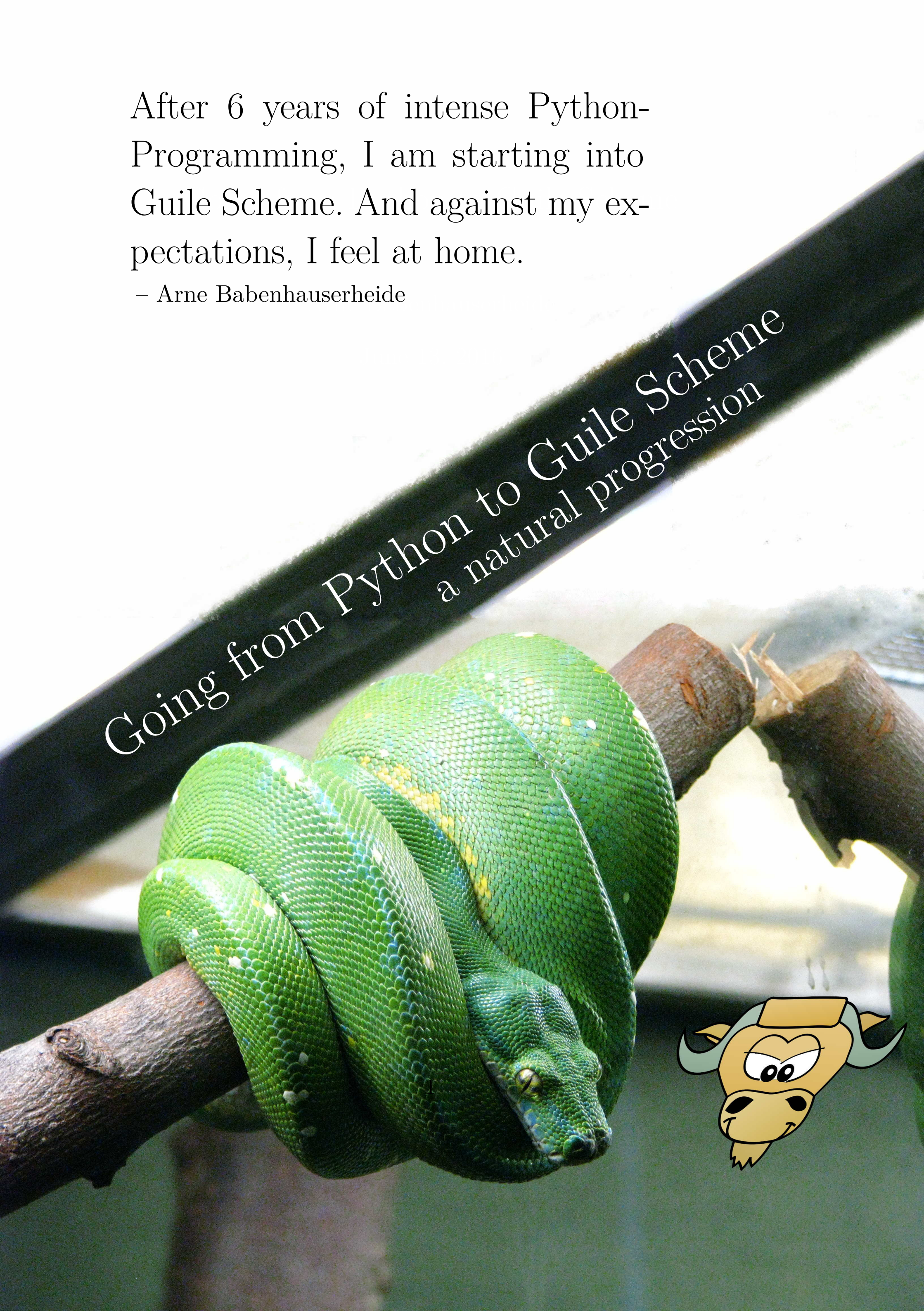
After 6 years of intense Python-Programming, I am starting into Guile Scheme. And against my expectations, I feel at home.
– Arne Babenhauserheide

Going from Python to Guile Scheme

a natural progression



Going from Python to Guile Scheme

a natural progression

For my children
who give me the strength to keep going every day

my love
who helps me keep my feet on the ground

my roleplaying group
which keeps my mind alive

my friendly collegues at the institute
whom I helped with Python

and all the Free Software and Free Culture Hackers
who make our world a better place.

The cover is made by VHR and Arne Babenhauserheide and builds on Green Tree Python from Michael Gil, licensed under the creativecommons attribution license, and Guile GNU Goatee from Martin Grabmüller, licensed under GPLv3 or later. It follows the time-honored tradition of ignoring the real history of the name Python.

© 2014–2015 Arne Babenhauserheide

Cover and Illustration: VHR and Arne Babenhauserheide, images from Michael Gil and Martin Grabmüller.

Released: 2015-09

Editing: Arne Babenhauserheide

Contributions by Ludovic Courtes (Ludo), Clinton (Unknown Lamer), David Thompson (davexunit), Mark Witmer and Mu Lei (NalaGinrut)

Publisher: Arne Babenhauserheide, Spöcker Str. 10, 76676 Graben-Neudorf, Germany

Dedicated to Jürgen Hesser.

I once tried to convince you
to teach Python instead of Scheme.
You were right in the end.
It seems those 30 years of experience
amount for something.

Thank you!

Contents

Part I

Python

Chapter 1

My story

1.1 Into Python

When I was still at school, I learned HTML and CSS. I was delighted: I could tell the computer to follow my orders. I was so happy that I even wrote text directly in HTML. It was a horrible syntax, but it worked. And I did not know better.

Later in my school-life I started with Java to contribute to a program. It felt bloated and wrong - even compared to HTML. I never learned to really program in it. When I went to university, I took a course on C. The hardest challenge in the first lessons was the syntax. I stopped halfway through the course because I learned more by asking my flatmate than from the course (thanks RK!).

A few months later I learned about Python, and it worked at the first glance. I felt at home.

1.2 And beyond

It's now 6-7 years since I wrote my first lines of Python, and I've been hacking on Python-projects ever since (and did my Diploma-Thesis in a mix of Python, template-heavy C++ and R, but that's a story for another day).

In Summer 2013 I then read The White Belt in the book Apprenticeship Patterns:[1]

> *"You are struggling to learn new things and it seems somehow harder than it was before to acquire new skills. The pace of your self-education seems to be slowing down despite your best efforts. You fear that your personal development may have stalled."*
> — David H. Hoover and Adewale Oshineye in Apprenticeship Patterns

I felt a pang of recognition.

I have grown so familiar with Python that the challenges I face at my PhD no longer require me to dig deeper with it. I mostly recreate solutions I already used for something else. So I decided to take a leap and learn something completely different. I chose Guile Scheme, because it provides its own sense of elegance and challenged my existing notions of programming.

[1]The book Apprenticeship Patterns applies the idea of patterns to personal development. It has been licensed under a Creative Commons License, though the O'Reilly page does not state that anymore. You can find the book in the chimera labs: `http://chimera.labs.oreilly.com/books/1234000001813/index.html`

To my surprise it felt strangely natural, so much so, that I wanted to share my experience. Where Python was my first home, nowadays it feels like Guile Scheme could become a second home for me.

If you want to use Guile Scheme, also have a look at Guile Basics[2] *which answers some of the questions I faced when starting my dive into Scheme.*

[2] Guile Basics collects some of the solutions I found when searching my way into GNU Guile. It is no book but rather a loose and unstructured hybrid, somewhere between a FAQ and a series of blog posts: `http://draketo.de/proj/guile-basics/`

Chapter 2

The Strengths of Python

To understand new experience, we need to know where we are. So, before I go into my experience with Scheme, let's start with Python and why it's awesome:

> *"Pseudocode with one right way to do it and hackable, scalable batteries."* — Arne Babenhauserheide

2.1 Pseudocode which runs

Python is rightfully known as pseudocode which actually runs. You can give a book-author a (simple) Python script and he or she will understand what it does without much explanation:

```python
for address in addresses:
  print address
```

But this is only part of what makes it special.

2.2 One way to do it

> *"Python is a language where I can teach a handful of APIs and cause people to learn most of the language as a whole."* — Raymond Hettinger (2011-06-20)

The simplicity of learning Python illustrated by this quote is enhanced by one of the pillars of the philosophy of Python:

> *"There should be one – and preferably only one – obvious way to do it."* — Aphorism 13 in the Zen of Python by Tim Peters

This philosophy, among others, is enshrined in the Zen of Python, also know as pep-20, easily called up in the Python interpreter by invoking:

```
import this
```

Following the Zen of Python creates nicely usable APIs and minimizes guesswork - and when you have to guess, you are right most of the time. Since most Python developers follow this philosophy, Python is a really nice language for facing real-life challenges: It provides only the functions needed to solve problems, with great default options, a strong focus on the actual tasks and polished so deeply that its different aspects merge together into one consistent whole.

It is a minimal ontology which encodes the most common operations in readily understandable wording, designed in a way which provides a clearly distinguishable canonical way to tackle a problem. A wonderful user-interface.

Together with looking like pseudocode, this makes Python a good choice for beginning programmers. In my first years of programming I thought that I'd never need anything else.

2.3 Hackable, but painfully

In all this simplicity, Python provides access to its inner workings. It allows you to do all the crazy things you need to do at times to solve problems.

Almost every aspect of its functionality is explicitly exposed at runtime in dictionaries. This provides for great introspection - and enables features like doctests, the most intuitive way I ever saw to test simple functions:

```python
def hello():
  """Greet the World.

  >>> hello()
  Hello World!
  """
  print "Hello World!"
```

You can create classes whose instances can be executed by adding a `__call__` method, and change the effect of mathematical operators by redefining the `__add__` method, and you can fiddle with the local bindings in your namespace and much more.

And if you need to get really nasty, there are always `eval` and `exec` to run self-generated code. I only had reason to use that one single time, but there it really worked out - and became completely unmaintainable. Luckily it was a one-shot script. I only had to change it once after the initial creation. In hindsight using `exec` was lot's of fun - and I hope I won't have to ever do it again.

While Python offers the possibilities, all this hacking feels hindered, as if the language provided resistance at every step.

For example, if you want to find the file in which a given function is defined, you can do it like this:

```python
# preparation: Get a function without context
from math import log
# get the module name
modulename = log.__module__
# import the module
mod = __import__(modulename)
# get the filename. Use dir(mod) to find out what you can do.
# Or use tab-completion in the python-shell
# (if you enabled GNU readline - if you did not:
# Best enable readline right now!)
filename = mod.__file__
print filename
# here this gives /usr/lib64/python2.7/lib-dynload/math.so
```

This is not what I would call elegant.

And I think that is intentional: Make the canonical way easier than other ways, but allow using other ways if someone really wants to.

Despite the verbosity and despite the double underscores screaming "do not touch this!", the discoverability is very good, because we can get all these options with dir() or tab-completion. Just explore to find out about the hidden tricks you could use.

2.4 Batteries and Bindings

Most of the time, you do not need to go to such extremes, though. Any standard Python installation already includes solutions for most problems you need to tackle in normal programming tasks, and there are bindings to almost every library under the sun.

```python
import antigravity
```

Most of these libraries are really well-done and modern, like matplotlib. And tools like cython make it very easy to write bindings - as well as extensions which compile as C-code and get all the performance you can wish for. The best program to do a job is one which already ships the solution. By that metric Python is a very good fit for most jobs nowadays.

Together with hackability this makes Python a good and "pragmatic" choice for experienced programmers.

Most of this is a product of hackability, searching for the canonical "pythonic" way to solve a problem and popularity, but it is still driven by the choice to provide these batteries and make it easy to use them.

2.5 Scales up

And finally, Python actually scales up from simple scripts to complex frameworks.

- Namespaces cleanly separate imported code *by default*,
- modules make code reusable *by default*,

- on-demand-import hacks minimize the loading cost of huge frameworks,
- properties allow starting with simple attributes which can use getters and setters later on *without breaking the exposed API*, and
- positional arguments which double as keyword-arguments make it easy to keep functions manageable *without breaking the exposed API* when their argument-list starts to grow.

Together with the other strengths, its scalability makes Python a very nice tool which accompanies you from your first tentative steps into programming up to highly productive professional work.

Chapter 3

Limitations of Python

With all its strengths, Python is still a language with a limited syntax. It is very broadly applicable, but it has strict rules how things can be done. These rules create a straightjacket you cannot escape easily. Most of the time, they are convenient, and they can help when you develop code in a community. But regardless of the chains you choose, they can never be convenient for all tasks. And whatever the task, when you go deep enough, even golden chains hurt.

3.1 The warped mind

> *“You must unlearn what you have learned.”* — Yoda in “The Empire Strikes Back“

If a programming language warps your mind, that manifests itself in limited imagination: When you tackle a problem, you think in the syntax of that language, and if that syntax cannot express something in a convenient way, you have a hard time even imagining that the solution could be easy.

Different from C++ and Git, Python only starts warping your mind very late in the game. But when it does so, it still hurts.

And hacking the syntax of Python is a task which is very distinct from general Python programming, so you cannot easily escape its chains.

On another front you could say that Python is the worst of mind warpers: It makes you think that source code can be easy to read and understand and versatile at the same time. And it is right, though it does not itself reach that goal completely. It set an important upper limit for acceptable unintelligibility: If a language is too painful, people will just use Python instead.

3.2 Templates condemn a language

```python
if __name__ == "__main__":
    # run the script
```

I really started feeling the limitations of Python when I had to write certain phrases over and over again. It requires quite a bit of ceremony[1] for regularly needed tasks. When you start thinking about using code-templates in your editor to comply with the requirements of your language, then it is likely that something is wrong with the language.

[1]Ceremony describes actions without information content which are needed only to fulfil the requirements of your tool.

A programming language is an interface between humans and the computer. If you need a tool to use the language, then it does not do its job.[2]

Though Python works pretty long with the basic indentation-support which also helps when writing prose, some of its long-winded phrases begin to really disrupt work. And a Python-Programmer cannot escape them.

3.3 Python syntax reached its limits

> *"Why, I feel all thin, sort of stretched if you know what I mean: like butter that has been scraped over too much bread."* — Bilbo Baggins in the Lord of the Rings from J.R.R. Tolkien

I find myself seeing constructs in Python-code as hacky workarounds which I previously accepted as "how programming works". I now look at this:

[2]This statement is a bit too general: A programming language actually is the interface between the programmer, the computer and other programmers - including the later self of the original programmer. Sometimes a bit of syntax-overhead can improve readability at the expense of convenience for the initial creation of the code. For those cases, templates can actually make sense. But this is not the case for `__name__ == "__main__"` and similar kludges in Python. If you want to dive into these issues, then you should start with prose or math: Text written by humans for humans, without a computer in the mix.

```python
if x:
    a = 1
else:
    a = 2
```

and I hate the fact, that I cannot just say

```
a = if(x 1 2)
```

Though I can actually do something similar since Python 2.5 (PEP-308), but it is written like this:

```python
a = 1 if x else 2
```

And that just looks alien. It does not feel like Python. But with Python-syntax, the only better solution is adding parentheses to make it look like a generator expression and as such very different from other Python-code (this is what Guido van Rossum recommends,[3] but in real-life I see people recommend the previous version):

```python
a = (1 if x else 2)
```

This isn't the simple Python syntax anymore. The uniformity of the syntax is broken. And this is not necessary for a programming language. For example with Scheme, where all code is just a tree of evaluated expressions, I can write the following - and it looks the same as all other code:

[3] "The decision was made to not require parentheses in the Python language's grammar, but as a matter of style I think you should always use them" — Description of PEP 308 in the release notes of Python 2.5. In my opinion this is a case where Python would have benefitted from requiring more parens. But then it would have shown much more clearly that the new syntax is essentially a different language style merged into Python. But still, do listen when your elected BDFL speaks. Use parens around inline if-else.

```scheme
(define a (if x 1 2))
```

This might seem useless, but I am currently hitting code would benefit from it all the time. For example here (real code):

```python
direction = [(math.atan(v[1]/u[1])
              if ((v[1]*u[1] > 0) and not v[1] < 0) else
              -math.atan(v[1]/u[1]))
             for u, v in zip(mfu, mfv)]
```

The new syntax additions to Python feel like Python syntax is already stretched to its limit. It is expanding into generator-expressions and list-comprehensions, because its "natural" syntax is breaking down with the new features people wish for (at least that's how it looks to me).

3.4 Complexity is on the rise

This expansion into non-pythonic territory does not stop at the syntax, though. When I started learning Scheme, I read an article by Michele Simionato on compile time vs. runtime in Scheme.[4] I thought "Luckily I don't have to worry about that in Python" – and just a few weeks later I stumbled over strange breakage of my Python function cache decorator.

It turned out that the simple act of adding function decorator syntax introduced all the complexities of separating compile time

[4]In compile time vs. runtime in Scheme Michele Simionato describes expansion, optimization and run-time, why Scheme tracks these phases and how it escaped from the dark tower of Meta-levels: http://www.phyast.pitt.edu/~micheles/scheme/scheme21.html

vs. runtime functionality into Python. The extent of this change can be shown in a few lines of code:

```python
def deco(fun):
    return mun

@deco
def mun():
    print ("Welcome to compile-time breakage!")

def mun():
    print ("We have to introduce a proxy.")

@deco
def proxy():
    pass
```

When this code tries to define the decorated `mun()` function, it fails with `NameError: global name 'mun' is not defined`. The reason is as simple as horrifying: The decorator `@deco` forces the function `deco` to run while `mun` is being defined, and the `deco` function requires access to the `mun` function during execution. This breaks the assumption that functions can use not yet defined functions as long as the execution of these functions happens later. It brings new import-conflicts and increases the depth of understanding you need to be able to anticipate how a given piece of code will behave.

This change, along with generator expressions and a few other features, strongly increased the complexity of the language without giving its users all the benefits which are available in languages that were designed from first principles to provide these features. Looking at this, it seems like the experiment which Guido van Rossum dared with Python failed to some degree:

> *"Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered."* — Guido van Rossum, 1996-08-13

There clearly is a need for more complex functionality among Python programmers – its limitations are perceptible – and in trying to fulfill this need, Python stretched syntactically and conceptually, and keeps stretching, but its limits already come into view. As it is pushed over the limitations of its design, the complexity of Python explodes and increases the cost of future additions. This makes it unlikely that Python can overcome its limitations without losing the strengths which made it an ideal tool to start programming.

This does not void any of its advantages. Python stays a great language, with an active community which promotes readable code and practically useful tools along with a mentality that stuff should just work. It is the language I would recommend and use any day to solve practical problems. But for diving into the depth of programming I have to broaden my view.

3.5 Time to free myself

And this brings us back to The White Belt from Apprenticeship patterns, this time in longer form:

> *"You have developed a deep understanding of your first language and are walking comfortably on a plateau of competence. [. . . but. . .] You are struggling*

> *to learn new things and it seems somehow harder than it was before to acquire new skills. The pace of your self-education seems to be slowing down despite your best efforts. You fear that your personal development may have stalled."* — David H. Hoover and Adewale Oshineye in Apprenticeship Patterns

I tried every trick with Python - from very clean, automatic documentation up to runtime code-generation. And now I hit a wall: Its limitations do not allow me to move onward.

Python accompanies you on your road from beginner to experienced programmer, but no further. I learned a lot about structuring information by programming in Python. I learned that programs can be easy to read and understand for newcomers. And I learned about the importance of having identifiers whose names in themselves form a well-defined and optimized language. But it is time to free myself from its shackles.

Part II

Guile Scheme

Chapter 4

Starting into Guile Scheme

So I started looking into other programming languages. I had two main contenders:

- Fortran, the tried and true tool for engineers, and
- Scheme, the academic branch of the Lisps.

I started with Fortran as the pragmanic choice for a physicist, but soon I caught myself replicating every Fortran-experiment in Scheme. So I decided to follow my gut and dive into Scheme. From there on, the choice was easy: There are several Scheme implementations - and one of them is from the GNU Project:

Guile.[1], [2] But before really starting with that, I read The Adventures of a Pythonista in Schemeland (PDF) by Michele Simionato.

It is half a year later, and Scheme now feels natural to me. Actually more natural than Python.

The expressiveness of the original syntax of Python was a bit too limited, and this caused the language to hatch a new syntax which makes the whole of Python much more complex. There is no longer a simple uniform syntax, but two complementary styles with different structures. It is still very easy to understand and I think that it set a new standard for readability of code - so in that aspect the Python experiment was a phenomenal success - but it is starting to break down as people expand it into more and more directions.

[1] Aside from Guile, there are lots of other Scheme implementations. Using the posting rates to mailing lists as a rough estimate of activity, Racket (user, devel), PLT (devel, plt) and Guile (user, devel) are roughly on the same level of activity of 5 to 10 messages per day while all the other Schemes are at least factor 2 below that. So from estimating activity, Guile looks like a sane choice. But to set this into perspective: The combined posting rate of these three most active Scheme lists together only approaches the posting rate of clojure, python-devel (without counting all the other python-lists) or javascript v8 devel alone. So if you're looking for pure activity, Scheme might not be the most obvious choice. But then, I did not find my way to Guile Scheme by searching for the most popular language. I found it while searching for ways to go beyond the limits of the languages I knew.

[2] The opinionated guide to scheme implementations from Andy Wingo provides an overview of the wealth of Scheme implementations: `http://wingolog.org/archives/2013/01/07/an-opinionated-guide-to-scheme-implementations`

Guile Scheme on the other hand can accomodate new functionality much more easily. And from the intuitive side, I now see commas between function arguments and they feel like needless bloat. My gut suddenly says “leave out the commas!”, and I find myself forgetting them in Python-code. I think the commas once looked useful to me, because languages with commas helped me get rid of the quoting hassles in the shell, but now there’s a language which does not require commas to achieve that goal - and stays clean. That sounds small, but it changes radically how the language feels.

4.1 But the (parens)!

“LISP: Lots of Irritating Superfluous Parentheses.”
— popular skit.

Lisps have long been criticised for their use of parentheses. And rightly so.

Yes, the parens are horrible. I no longer see them as strongly as when I started (they faded a bit into the background), but I still remember how they horrified me when I began hacking my Emacs config - and even when I started with Guile.

```
(for
 ((beginners (readability of parens))
  (is horrible)))
```

This becomes even worse in bigger code examples - with often 5 or 6 parens closed at the end of a function. The moment your examples get bigger than the ones in the first 90% of The Little

Schemer,[3] it becomes hard to keep track of the parens without strong support from your editor.

So I started contributing to a fix. At first I joined readable (nowadays known as SRFI-110). But when the readable-developers added syntax using `$`, `\\` and `<* *>`, readable lost me. It had left the path of the minimalist elegance which fascinates me in Lisp. Therefore I began to work on a simpler solution.

That solution is Wisp: A whitespace-to-Lisp preprocessor.

Wisp also known as Scheme Request For Implementation 119: SRFI-119.[4, 5]

The previous code-block with parens was indented in the canonical way for Lisp. If you write the same code-block without parens and add some syntax for double- and inline-parens, Wisp can transform it into full-fledged scheme which you can then run:

```
for
  : beginners : readability of parens
    is horrible
```

Wisp uses the minimum indentation-sensitive syntax which can represent arbitrary Lisp-structures and is implemented in Wisp itself. With this, the readability is not yet quite on the

[3]The Little Schemer is a popular and very readable introduction to understand the nature of computing by programming with Scheme, and a great start into thinking in recursion: `https://mitpress.mit.edu/index.php?q=books/little-schemer`

[4]SRFI-119 standardizes the syntax of wisp: `http://srfi.schemers.org/srfi-119/srfi-119.html`

[5]For some more background why I took the step to create Wisp instead of using readable, see my presentation Why Wisp?: `http://draketo.de/proj/wisp/why-wisp.html`

level of Python, but it is getting close - at least close enough for me.

I started with an analysis of what indentation-sensitive syntax means, and interestingly I now find ways to write more elegant code in that syntax - ways I did not think about when I defined Wisp. I am still learning how to write nice Wisp-code, and I think that is a good sign: The syntax brings its own "natural" style.

Thanks to the flexibility of GNU Guile, you can even use Wisp in the interactive console (implemented with help from Mark H. Weaver and others from #guile on irc.freenode.net!). Just get Wisp[6] and run

```
./configure; make check;
guile -L . --language=wisp
```

After having a fix for the most pressing problem I see in Guile Scheme (the parens kill newcomers), I could proceed to testing how Guile Scheme with Wisp compares to Python - and as you'll guess, you read this book, because Guile Scheme did remarkably well.

[6]To get wisp, install Guile version 2.0.11 or later and run `wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.9.0.tar.gz; tar xf wisp-0.9.0.tar.gz ; cd wisp-0.9.0/; ./configure; make check; examples/newbase60.w 123`
if it prints 23, you are set up.

4.2 Summary

We saw how complexities crept into Python, which was written with the expressed goal to be more limited than Lisps, making it one more example of Philip Greenspuns 10th Rule:[7]

> *"Every sufficiently complex application/language/tool will either have to use Lisp or reinvent it the hard way."* — Generalization of Philip Greenspuns 10th Rule

In contrast, after some time of getting used to it and finding a fix for the parens, Scheme now feels really natural for me.

And with that, I can go on and compare Guile Scheme to the strengths of Python. We will pit Guile Scheme against Python in the areas where Python rules the place and see how Guile Scheme fares.

[7] Philip Greenspuns 10th Rule is immortalized in the venerable ContentCreationWiki: `http://c2.com/cgi/wiki?GreenspunsTenthRuleOfProgramming`

Chapter 5

Comparing Guile Scheme to the Strengths of Python

After having listed the many strengths of Python, it's time for a very unfair comparison: How does Guile Scheme stand against the strongest aspects of Python?

I ask this, because it is what Python-Programmers will ask - and because it is what **I** asked myself when looking into Guile Scheme.

Later we will see where Guile Scheme enables us to go beyond Python. But now: Let's start the race.

5.1 Pseudocode

We showed that Python is rightfully known as "Pseudocode which actually runs".

Using the indentation sensitive syntax of Wisp, Guile Scheme also comes close to runnable pseudocode. Maybe even closer than Python. Let's test this with examples.

General Pseudocode

The following shows Guile Scheme code using the Wisp-reader to leave out most parentheses. It realizes the well-known FizzBuzz game which is used in the English Wikipedia as an example for pseudocode. And different from the Wikipedia-examples, the code here actually runs.

```
;; this example needs foof-loop installed via guildhall!
;; see https://github.com/ijp/guildhall/wiki/Getting-Started
use-modules : guildhall ext foof-loop

;; Pseudocode adapted from
;; http://en.wikipedia.org/wiki/Pseudocode#Syntax
define : divisible? number divisor
         = 0 : remainder number divisor

define : fizzbuzz
  let : : print_number #f
    loop : : for i : up-from 1 : to 100
        set! print_number #t
        when : divisible? i 3
               display "Fizz"
               set! print_number #f
        when : divisible? i 5
               display "Buzz"
               set! print_number #f
        when print_number
               display i
        newline

fizzbuzz
```

I did not expect to find such simple pseudo-code in another language than Python, but Scheme actually provides it.

Uniformity

Similarly, uniformity of code is generally considered as one of the huge strengths of Python, a strength which gives normal code the readability of pseudocode: When you know one Python-program, you can find the patterns used there in all future Python-programs you read.

Modern Python however offers a kind of dual syntax. The first, simple syntax provides indentation-sensitive control flow, declarative data-definition and function calls:

```
for i in [1, 2, 3, 4]:
    if i >= 4:
       continue
    def counter():
        if i != 2:
            for j in range(i):
                yield j
    print list(counter())
```

And then there are generator-expressions, the second syntax:

```
for k in (range(i)
          if i != 2 else []
          for i in [1, 2, 3, 4]
          if i < 4):
    print k
```

(These are synthetic code examples to show you all the interacting features as little space as possible. Make them three times as long and calling into external functions at every step, then you have something close to real code I read and wrote.)

Both of these expressions are valid Python, and they yield the same result, but in the tutorials I saw over the years, newcomers mostly learn the first style, not the second - and the first style is the original Python-syntax. When you see that second syntax in raw form, it looks really bolted-on and the parentheses only barely contain, how alien it is to the original Python:

```python
a = 1
b = 1 if a != 1 else 2
c = (1 if b > 2 else 3)
```

And here Guile Scheme can go a step further towards uniformity.

The addition of generator-expressions to Python essentially creates two completely different languages mixed into one. And I expect to see huge battles between these two syntactic forms in the coming years.

In contrast, Scheme provides one single syntax: The function-call with parameters and a return value. And the symbol-binding let-syntax is quite close to the generator-style in the Python-example:

```
use-modules : guildhall ext foof-loop

loop : : for i : in-list '(1 2 3 4)
         if {i < 4}
            let : : k : cond ((not {i = 2}) (iota i)
                             (else '()))
                    display k
```

But different from Python, this is the default syntax in Scheme. A common theme between both is that the outer code uses indentation (in Scheme via Wisp) while the inner code uses parentheses. There is also some degree of duality in this Scheme example, but in Python the inner code structure works differently than the outer code while in Scheme the only change is that the code switches to using parentheses instead of indentation to mark scope. You could use indentation for the inner part, too, but that would look more busy, and parentheses everywhere would be harder to read for most people (though it would look more uniform). This is the value of having complementary ways to format your code, in Python as well as in Scheme. But both ways should use a uniform structure, and Python does not achieve that while Scheme does so easily.

And it is possible to get even closer to the generator-example in Python without breaking the basic syntax by using the syntax-adjustment capabilities Scheme provides for its users.

It misses the kind of polish which went into Python and which has the effect that after some time the generator expressions in Python look neat instead of completely alien. But the foundation of Scheme is much stronger: It can express both styles in the same structure.

So let's use this strong position to look at something which appears to be a sore spot at first:

Pseudocode with loops

When starting to look into loops in Guile Scheme, my initial impression was bleak. But don't let yourself be fooled by that: When I looked deeper into it, that impression changed: The looping constructs in basic scheme are pretty limited, but it has many syntax extensions which make looping enjoyable.

Let's first start with my initial impression of basic scheme. This is what newcomers will see (at the current state of the documentation).[1]

Initial impression

Scheme canonically only supports do-loops, while-loops and let-recursion, so loops look like the following:

Do The basic loop, similar to a for-loop in `C`.

```
do : : i 1 : 1+ i
   : > i 4
   display i
```

Note: Almost no schemer uses this.

While Looping with while is sometimes seen, but similar to do-loops mostly superceded by more elegant constructs.

```
let : : i 1
  while : < i 5
         display i
         set! i : 1+ i
```

Let-recursion Also called named let: looping via explicit recursion. Among other possibilities, let-recursion can easily be used instead of `do` or `while` loops.

[1]Hint: if you write a tutorial on Scheme, do NOT start with do-loops. Rather start with let-recursion and simple usages of SRFI-42 and foof-loop. But please don't reference their documentation documents as if they were tutorials. That would be a very rough start.

```
let loop
  : i 1 ; the initial value
  when : < i 5
         display i
         loop : 1+ i
```

Even though let-recursion is a great tool, it has quite a bit of overhead for simple loops, and it requires shifting the mental model towards recursion. But as soon as the loop gets more complex than two or three lines, this overhead fades.

Intuitive? These constructs are not really what I consider intuitive, but they are easy and not really bad. A looping construct I would consider as intuitive would be something like this:

```
for : i in '(1 5 8)
      display i
```

(you can do that with a macro - people do: see the next chapter!)

On the other hand, I experienced let-recursion to be much easier to debug than any other kind of loop - so much easier, that I wrote an article about the elegance of let-recursion.[2]

And for this gain, I accept the slight loss in readability:

[2]When I realized that let-recursion provides the simplest possible model for recursive code with initialization of the recursion start, I wrote an article about my experience: `http://draketo.de/licht/freie-software/let-rekursion` (in german)

```
let loop
  : l '(1 5 8)
  when : not : null? l
       display : car l ; first element
       loop : cdr l ; continue with rest
```

I can't deny, though, that standard scheme-loops are still a long way from Python - especially in list-comprehensions:

```
names = [name for ids,name in results]
```

But the flexibility of scheme-syntax would definitely allow defining something which looks like the following:

```
define names : list-comp name for (ids name) in results
```

And I'm pretty sure that somewhere out there someone already defined a list-comp macro which does exactly that. Let's see...

Great Looping Constructs in Guile Scheme

...and one question on IRC later (#guile @ irc.freenode.net)[3] I learned that I was right: SRFI-42 (eager comprehensions) offers list-comprehension while foof-loop provides the loop-macro.

SRFI-42 SRFI-42 allows using the compact representation from Pythons list-comprehensions:

[3]To get instant contact with the guile developers, visit the freenode IRC webchat: `http://webchat.freenode.net?randomnick=1&channels=%23guile`

```
use-modules : srfi srfi-42
list-ec (:range i 5) i ; [i for i in range(5)]
; => (0 1 2 3 4)
```

foof-loop and foof-loop[4] gives powerful looping, which starts simple

```
;; this example needs foof-loop installed via guildhall!
;; see https://github.com/ijp/guildhall/wiki/Getting-Started
loop : : for element : in-list list
  display element
```

and becomes very, very powerful:

```
;; this example needs foof-loop installed via guildhall!
;; see https://github.com/ijp/guildhall/wiki/Getting-Started
loop
  : for x : in-list '(1 2 3)
    with y 0 {y + {x * 2}}
  . => y
```

apply/fold/map In addition to providing these explicit looping constructs, Scheme developers are far more likely to use functions like apply, fold and map for the same effect, often with an anonymous (lambda) function which replaces the loop body.

[4] To get foof-loop for guile, you need to first install guildhall, a package manager for guile. See Getting Started for a short tutorial: `https://github.com/ijp/guildhall/wiki/Getting-Started`

Summary

Guile Scheme with syntax extensions does not have to look up to Python when it comes to pseudocode. Scheme code can be very elegant, readable and intuitive. With the right extensions that even holds for loops. And both foof-loop and srfi-42 are more powerful looping-constructs than the default in Python. For example `list-ec (:real-range i 0 5 0.1) i` is equivalent to the numpy-enhanced range-function. And despite that power, their code looks almost as intuitive as Python-code.[5]

But they also come with lots of additional ways solve a problem. Which brings us to the next topic.

5.2 One way to do it?

So readability can be pretty good, but when it comes to canonical code, Scheme follows a very different path than Python. Instead of providing one best way to do something, Scheme is a tool for creating the language you need to solve your problem. This means, that while Scheme code can look much clearer than Python-code, it can at the same time be much harder to understand. When you program in Python, you'll soon see patterns

[5]In my opinion, SRFI-42 still falls a small way short of Python list comprehensions. In Python, the syntax looks like the datastructure it creates, joined in a sentence. For example, `[i for i in range(5)]` can be spelled as "the list containing i for each i in range up to five". In SRFI-42 the syntax rather looks like a statement. I would spell the example `(list-ec (:range i 5) i)` as "the list eager comprehension which uses i from the range up to five as i". It does not feel quite as polished as the Python version. But it is already very close and quickly becomes natural.

in the code, so you don't actually read the full code. Rather you say "ah, yes, this is a future". In Scheme on the other hand, every programmer can use different mechanisms for doing the same task.

This sounds pretty harsh, and it warrants an example. For this, we'll turn to Sly,[6] a really nice game-engine in Guile Scheme, modelled partly after pyglet.

Writing a movement pattern for a character in Sly looks very intuitive:

```
use-modules : 2d agenda
              2d coroutine
              2d game

coroutine
  while #t
    walk 'up
    wait game-agenda 60
    walk 'down
    wait game-agenda 60
```

But when you try to understand what `coroutine` does, you have to look deep into the nexus of delimited continuations - and then mangle your mind some more to build the patterns used there. Let's do that: We take one step into coroutine.scm.

[6]Sly is a game engine for Guile Scheme which provides a dynamic live coding environment that allows games to be built interactively and iteratively: `http://dthompson.us/pages/software/sly.html`

```
define : call-with-coroutine thunk
  . "Apply THUNK with a coroutine prompt."
  define : handler cont callback . args
    . "Handler for the prompt.
       Applies the callback
       to the continuation (cont)
       in a second prompt."
    define : resume . args
        . "Call continuation
           that resumes the procedure.
           Uses the continuation
           from the handler."
        call-with-prompt 'coroutine-prompt
              lambda () : apply cont args
              . handler
    ; here the handler
    ; calls the callback with resume
    when : procedure? callback
        apply callback resume args

  ; finally call-with-coroutine
  ; calls the code (thunk).
  call-with-prompt 'coroutine-prompt
                 . thunk handler

; definition of the coroutine macro.
; Enclodes the code in a function (lambda)
define-syntax-rule : coroutine body ...
  . "Evaluate BODY as a coroutine."
  call-with-coroutine : lambda () body ...
```

Firstoff: This really is the full definition of coroutines. 11 lines of concise code (not counting blank lines, docstrings and comments). From my experience with Python, I would say “this will be easy to understand". Let's try that - beginning with the macro `coroutine` at the bottom.

```
define-syntax-rule : coroutine body ...
  . "Evaluate BODY as a coroutine."
  call-with-coroutine : lambda () body ...
```

This one is still easy: If you call `coroutine`, it simply puts all the arguments you give it inside a lambda and passes it to `call-with-coroutine`. In python, you would do that by defining a function and passing the function around. So far, so nice. Now let's get to the core and understand `call-with-coroutine`:

```
define : call-with-coroutine thunk
  . "Apply THUNK with a coroutine prompt."
  define : handler cont callback . args
    . "Handler for the prompt.
       Applies the callback
       to the continuation (cont)
       in a second prompt."
    define : resume . args
        . "Call continuation
           that resumes the procedure.
           Uses the continuation
           from the handler."
        call-with-prompt 'coroutine-prompt
              lambda () : apply cont args
              . handler
    ; here the handler
    ; calls the callback with resume
    when : procedure? callback
        apply callback resume args

  ; finally call-with-coroutine
  ; calls the code (thunk).
  call-with-prompt 'coroutine-prompt
                  . thunk handler
```

So `call-with-coroutine` first defines the internal function `handler`. That handler gets the arguments `cont`, `callback` and args. It defines the internal function `resume`. When `resume` gets called, it uses `call-with-prompt`. This isn't defined here: It uses continuations, which are something like a supercharged `yield` from Python. They allow stopping a function at any point and later resuming it from there - multiple times if needed. So this `handler` returns a function which can continue the control flow - conveniently called `resume`. And if I call a `coroutine`, I create code which can stop itself and give other functions a way to continue it where it stopped (getting from defining `resume` to passing it to other functions is a huge leap. You can read some more about this in the chapter about `prompts` in the Guile Reference Manual).

I won't go into further details of continuations here, because I cannot explain them in an easy way - it took me a few hours to actually figure out what this code does, and I still have problems wrapping my mind around all the details. The fundamental power of delimited continuations is so huge, that after I mostly understood what this code does, I wrote a short note to a fellow Science-Fiction RPG designer and told him, that the paranormal time-warping power we designed could be modelled completely with continuations and a diff-based memory implementation.

But let's finish this: On the outside, these 11 lines of code give you a way to define code you can step through - for example going one line at a time, and every step returns a function with which you can `resume` the function to run the next line.

This is elegant. I cannot even get close to describing the extent of the elegance of this approach.

But it requires me to wrap my mind around very complicated concepts to understand why the very simple code for the movement of a character works.[7]

If I were to use another library, it would likely provide a slightly different way to define coroutines. So I cannot easily

[7] `call-with-prompt` is already the simplified version of the even more complex concept of general continuations. Let's not go there for the time being...

build up patterns to quickly understand code. I have to actually read the code line-by-line and word-by-word. Then I must read up on the definition of the structures it uses. Only after doing this, I can begin to hack the code. And this is a problem for code-reuse and maintainability.

And additionally most Scheme implementations provide slightly different base functionality.

So on the grounds of providing one way to solve a problem, Scheme falls far short of Python.

Partly this is the price of Freedom for the Developer.

But another part of this is simply, that the functionality in Scheme seems to be just a tiny bit too low-level. It does not expose a well-defined set of easy functionality to build programs. Instead it provides a set of very powerful tools to build languages - but the simple ways, which are the default in Python, are mostly missing – or hidden within the wealth of low-level functionality. In the strive to provide the perfect environment to define languages, Scheme loses sight of the requirements for solving actual problems with code. It is a very elegant language for defining languages, but for solving actual problems, each library provides its own domain specific language, and that makes code harder to understand than needed.

I do not see this as an unsolvable problem, but from the outside it looks like developers are happy with the situation: They have all the freedom they need, and they can ignore the rough edges. But those edges cut the hands of new users.

Note, though, that every set of functions provides a kind of domain specific language, independent of the programming language you use. If you create a module in Python with functions for doing a given task, then the names of the functions and the arguments they form a language. But (to stay with the analogy) the shared vocabulary and grammar between those languages is much bigger in Python than in Scheme. Python-programmers mostly try to write conventional code: Code which looks just like the user expects it to look.

And in Python, this is the easiest way to write code – that is a part of the Zen of Python – while in Scheme unconventional code is just as easy to write as conventional code.

So in Scheme there is not *one way to do it.* There are countless ways, and most of them can be made to work similarly well.

5.3 Planned Hackablility, but hard to discover. . .

Similar to the many ways to do a task in Guile Scheme, there are big differences to Python, when it comes to hackability.

Python exposes most of its inner workings via double-underscored attributes and functions. As such, almost every obscure possibility is available at all times, but never elegantly.

Guile Scheme on the other hand provides explicit ways to change its inner workings, but does not give access to everything it could. Consequently, hacking Guile and hacking Python are two completely different things. We'll go through a few examples to get a feeling for the different styles.

Accessing variables inside modules

As in Python, Guile Scheme allows you to access all toplevel variables in a module. Whether you only see exported variables or all of them is a matter of whether you use 'resolve-interface' or 'resolve-module'. When defining a module, you explicitely define which values are exported. In contrast, Python uses the convention that names starting with an underscore are not exported and all others are implicitly exported.

To get a list of variables and functions in a module you can use dir:

```
import math
dir(math)
```

A rough equivalent in Guile Scheme is

```
module-map
  lambda (sym var) sym ; return the symbol (key)
  resolve-interface '(ice-9 popen)
```

Getting an exported binding (variable or function) directly can be done via `@`:

```
define oip : @ (ice-9 popen) open-input-pipe
```

To get a non-exported binding, just use `@@` instead of `@`.

If you want to get the bindings for a module referenced in some datastructure, `module-ref` might be more convenient:

```
define oip
  module-ref
    resolve-module '(ice-9 popen)
    . 'open-input-pipe
```

Runtime Self-Introspection

The title of this part is a fancy name for "getting implementation details at runtime which a clean system should not need but which many people use anyway". For example for locating image-files which are stored in a path relative to the script-file (which is evil if the program gets installed cleanly, but can come in handy during development and for deploy-by-dropping-a-folder-somewhere).

A file as module and script

In Python, you can always check whether a file was imported as module or started as script by checking for `__name__`. If that is `__main__`, then the script is the file the user executed. And you can retrieve the filename of a module with the magic attribute `__file__`.

```python
if __name__ == "__main__":
    print "you executed", __file__
```

In Scheme you do not use magic attributes as in python, but you have several ways to achieve the same.

You can explicitely check the command-line arguments for running code only when the file is called as script. This example is not yet perfect, but it does its job as long as you do not reuse the filename of modules you use in other parts of the load path. Note that for this part I switch back from Wisp to Scheme (with parens), because this makes it easier to discuss the code with scheme-hackers (and I'm pretty far out of my zone of expertise, so this is important).

```
(define (my-local-main)
  (display "I am just a dummy, why do you call me?"))
(let ((my-local-name (module-filename
                      (program-module my-local-main))))
   ; catch the case when the compiler
   ; optimized away the info.
  (when (or (not my-local-name)
            (string-suffix? (car (command-line))
                            my-local-name))
    (display "you executed this as script.")))
```

The limitation not to repeat a filename is a serious one, though: Do NOT use this code in production. It is here to show feature parity by taking a similar approach. Luckily there are safer solutions.

If you know the location of your guile interpreter, you can use the meta-switch. This is very useful for local development and for distributions, but it does not work if you need to use `#!/usr/bin/env guile` as the hashbang to make your script more portable (the meta-switch is something which is direly missing in env - even GNU env).

As simpler alternative, you can run any file from guile as a script instead of importing it as module: just call it with `guile -e main -s scriptfile.scm`. This misses the use-case by a fine margin, because it moves part of the running logic outside the script file, but it shows the way towards the most versatile solution in Guile:

Just use a hashbang for a shell script which contains the information how to run the module as script. This is possible because in guile-scheme `#!` starts a comment which lasts up to

`!#`, so when the shell re-runs the file with guile, guile will ignore the shell-part of the file.

```
#!/bin/sh
# -*- scheme -*-
exec guile -e main -s "$0" "$@"
# Thanks to exec, the following lines
# are never seen by the shell.
!#
(define (main args)
  (display "Horaay!\n"))
```

And this is technical elegance in its raw form: Interaction of different parts of the system to form something much greater than its parts by generalizing an already existing special case (the hash-bang as starting a multiline-comment). I pull my hat before whoever discovered and realized this solution.

If you want to use this in a module, you need to call the main from the module:

```
#!/bin/sh
# -*- scheme -*-
exec guile -e '(@ (py2guile runscript) main)' -s "$0" "$@"
# use @@ if your module does not export main
!#
(define-module (py2guile runscript)
                #:export (main))
(define (main args)
  (display "Horaay! Module! Horaay!\n"))
```

Using this with Wisp requires some additional arguments to add the wisp directory to the load path:

```
#!/bin/sh
# -*- wisp -*-
exec guile -L $(dirname $(dirname $(realpath "$0"))) \
     --language=wisp -e '(@ (py2guile runscript) main)' \
     -s "$0" "$@"
# use @@ if your module does not export main
; !#
define-module : py2guile runscript
              . #:export (main)
define : main args
  display "Horaay! Module! Horaay!\n"
```

To estimate the overhead of running a shell-script and then deferring to guile, I compared this script to a script started with `env` and a script which uses the meta-trick:

```
#!/usr/bin/env guile
!#
(define (main args)
  (display "Horaay!\n"))
(main 1)
```

```
#!/usr/bin/guile \
-e main -s
!#
(define (main args)
  (display "Horaay!\n"))
```

As final test I added a script which used dash as shell instead of bash (by replacing `/bin/sh` with `/bin/dash`).

I then ran the scripts with a simple for-loop:

```
for script in runscript-shell.scm runscript-dash.scm \
              runscript-env.scm runscript-meta.scm; do
    echo $script; ./${script} >/dev/null;
    time for i in {1..1000}; do
       ./${script} >/dev/null;
    done;
done
```

The runtimes for 1000 runs were 20-24s when deferring to shell, 20-24s when using dash, 18-20s when calling the script via env and 17-19s when using the meta-switch. So the overhead for running via the shell is about 3-4ms - which I would call mostly negligible given that the Python equivalent of this script requires 19-23ms, too.

One problem could be that the startup time for the shell method is a bit unstable: On some runs it jumps up to 30ms. To check that, we can look at the average value and the standard deviation and range in 1000 runs:[8]

```
for script in runscript-shell.scm runscript-dash.scm \
              runscript-env.scm runscript-meta.scm; do
    echo real $script; ./${script} >/dev/null;
    for i in {1..1000}; do
        time ./${script} >/dev/null;
    done;
done 2>&1 | grep real | sed "s/.*0m.*\\.0//" | sed s/s$//
```

- shell-deferring: 23.2 ± 1.9 ms (range from 14ms to 29ms)
- dash: 22.7 ± 2.5 ms (range from 14ms to 27ms)

[8] According to Andy Wingo's writeup on Elf in Guile, version 2.2 should reduce these startup times quite a bit: `http://wingolog.org/archives/2014/01/19/elf-in-guile`

- env: 22.3 ± 2.8 ms (range from 13ms to 26ms)
- meta-trick: 19.7 ± 2.8 ms (range from 13ms to 19ms)
- python (with `if __name__...`): 22.0 ± 1.8 ms (range from 18ms to 26ms)

These numbers are with guile 2.0.11 on a 64bit machine with a standard rotating disk (no SSD). They show that the larger spread is due to rare outliers.

I guess that the additional spread of the startup times when using the shell-deferring is due to filesystem-access and caching, but I did not trace it further. The additional 3ms of the average time with shell-deferring is just the startup time of bash when started from bash via `bash -c true`.

So on average using the feature-equal method to call a script in Guile Scheme (shell-deferring) is just as fast as the equivalent method in Python. But there are ways to decrease the startup time if you have additional information about the system.

Practical Hint (as far as I understand it): If you are a distribution maintainer, and you see shell-deferring in Guile scripts, you can speed them up with the meta-trick. But if you are a developer and you want to make your script as portable as possible, your best bet is shell-deferring. This is also what `guild` uses to create executable guile-modules, so it is very unlikely that something will break this behaviour.

For larger projects you'd be better off with defining a script-runner which runs functions imported via `(@@ (package module) function)`. A script-runner with some error information would be the following script:

```
#!/usr/bin/env guile
-*- scheme -*-
!#
; This code will run the main function of the file it is given
; as argument. This keeps the testing logic inside the project
; while keeping the overhead for each module to a minimum.
; Thanks to Mark Weaver for this code!
(use-modules (ice-9 match))

(match (command-line)
  ((_ file args ...)
   (primitive-load file)
   (let ((main (module-ref (current-module) 'main)))
     (main args)))
  ((cmd . _)
   (format (current-error-port)
           "Usage: ~a FILE [ARGS ...]\n"
           (basename cmd))))
```

Where am I?

A more complex example asks, in which file a function I am using is defined. In Python, this works roughly like this:

```
from math import log
log.__module__ # it is math
import math
print math.__file__
# this code prints /usr/lib/python2.7/lib-dynload/math.so
```

or rather

```
from math import log
print __import__(log.__module__).__file__
# this code prints /usr/lib/python2.7/lib-dynload/math.so
```

In Guile Scheme I would achieve the same effect with this:

```
use-modules : system vm program
              (ice-9 popen) #:select : open-input-pipe
; then regain the module path
display : module-filename : program-module open-input-pipe
newline ; this code prints ice-9/popen.scm
```

To find the absolute path, you need to search `%load-path` for the file-path (a variable holding the list of the load path). For example you could it like this:

```
map
  lambda : x ; lambda is an anonymous function.
             ; In code you can also use the lambda-symbol,
             ; but it breaks my latex export.
      let*
        : sep file-name-separator-string
          path : string-join (list x "ice-9/popen.scm") sep
        if : file-exists? path
           . path
  . %load-path
```

Some more information on this is available in the nala-repl interpreter hacks.[9]

For using command-line arguments and such, have a look at Guile-Scripting.[10]

As you can see in the example, the python-versions often look more hacky, but they are shorter. Yet a big difference between

[9]The nala-repl interpreter hacks provide utility functions like getting the source code of a program: `https://github.com/NalaGinrut/nala-repl/blob/master/nala/src.scm`

[10]Guile-Scripting from the Guile Manual describes how to use GNU Guile for scripts: `http://www.gnu.org/software/guile/manual/html_node/Guile-Scripting.html`

both is that in Guile Scheme you could add syntactic sugar yourself to make this nicer. And that's where we now come to.

freedom: changing the syntax is the same as regular programming

The following shows an example for checking whether the file was called itself. Made easy. Using syntax macros, it replaces `if __name__...` from python with a block within `((in=m ...))`.

This example illustrates the power you get from tweaking the syntax in Scheme, but it is not a perfect clone: If you repeat a filepath in another part of the load path, the code will be run. I consider that a minor issue, because repeating the file path is also a toxic case in Python. In Python it would result in replacing a loadable module, potentially wreaking havoc on many other programs. If you find yourself tempted to use this example for creating scripts which also serve as modules, please turn to the safer ways shown in the previous section: The ***meta-trick****, if you know the location of the guile-interpreter, or* ***shell-deferring*** *(env on steroids) if you want to be as portable as possible. Guile Scheme is not Python, so some tasks are done differently - often in a more versatile way.*[11]

[11] The `in=m` example uses parenthesized syntax, because it needed a few round-trips to get right so I needed to make it as easy as possible to discuss the implementation with other Schemers.

```scheme
; define the module
(define-module (inm)
  #:export (inm in=m))

; get introspection capabilities
(use-modules (system vm program))
; define a syntax rule.
(define-syntax-rule (in=m body ...)
  (lambda ()
    (define (my-local-main)
      (display "I am just a dummy, why do you call me?"))
    (let ((my-local-name
           (module-filename (program-module my-local-main))))
  (when (or (not my-local-name) ; catch the case
                                ; when the compiler
                                ; optimized away the info.
            (string-suffix? (car (command-line)) my-local-name))
        (begin body ...)))))
; the lambda is executed here, not in the macro!
((in=m (display "you executed inm.scm as script")
       (newline)))
```

From another module:

```scheme
(define-module (in))

(use-modules (inm))
((in=m (display "in.scm")))
```

it is not triggered when importing this in yet another file.

```scheme
(use-modules (in))
```

Note that doing stuff like this is seen as normal programming in Guile Scheme, while it is seen as hacky in Python. The scope of hack and expected usage differs between Python and Guile Scheme.

Extending the syntax of your language to better suit your problem space is a core feature of Guile Scheme.

Discovering starting points for hacking

So far my tries to change something which isn't in the explicitely supported adaptions weren't as successful as I had expected. But I'm comparing several years of intense experimenting with Python to experimenting with Guile Scheme now-and-then, so all I can say is: discovering starting points for doing something which the main developers did not anticipate requires a different approach than in Python. When searching what I can do, I'm used to simply dropping to the Python shell, typing the variable with a trailing period and hitting tab to let GNU readline completion do the rest. Since Scheme does not give access to the namespace content via the dot-syntax, I cannot use this here.

The approach which currently works for me with Guile Scheme is just asking on IRC, but that does not scale to a quickly growing userbase (when the number of users grows much faster than the number of experts).

One of the things you should definitely do when starting with Guile is getting familiar with GNU Info - either the standalone reader (just type `info` in a shell) or better still the info-mode in emacs (invoked with `C-h i`). Then just go to the guile topic (hit `m`, then type `Guile Reference`) and do a full-text search with `ctrl-s <search text> ctrl-s ctrl-s` (repeating `ctrl-s` tells info to search all subtopics). That lessens the need to ask a lot.

If you want to use a search engine, add “Guile Scheme” as identifier. “Guile” often gets the character from the Street Fighter game series and “Scheme” gets anything from URL schemes to business processes, but not how to use them *in Scheme*.

5.4 Batteries and Bindings: FFI

The Batteries and Bindings of Guile are generally weaker than those of Python. To mitigate this a bit, Guile provides a simple way to call libraries written in C: The Foreign Function Interface (FFI).

With this I can wrap a library into a Scheme-module so it appears like a native tool. To investigate, I tested the simplest case of wrapping a library I'll really need: netCDF4.

```
use-modules : system foreign
; load the netcdf library.
define libnetcdf : dynamic-link "libnetcdf"
; get the function to inquire the netcdf version.
define nc_inq_libvers
    pointer->procedure
      .  '* ; returns a pointer to a char-array.
      dynamic-func "nc_inq_libvers" libnetcdf
      list ; takes no arguments (empty list)

; test the foreign function
; and convert the returned char-array to a string.
pointer->string : nc_inq_libvers
; => "4.1.1 of Jul  1 2013 03:15:04"
```

So I can wrap a library without too much effort. As seen in the example, wrapping a simple function can be done in just 7 short lines of code.

The Guile manual provides important resources for working with FFI:

- Foreign Functions
- Void Pointers and Byte Access

It leaves quite a few open questions, though:

- Can I wrap fortran? equivalent to f2py? fwrap to wrap fortran in C and call it? Use Fortran Interoperability with C / Interfacing with C?
- How good is its Performance? bytevectors for direct memory access?
- How to wrap other languages? Is there something equivalent to Integrating Python with other languages? Or Using Python as Glue (ctypes should be equivalent to FFI)?

So while not being too complex, this is also not really painless. Guile provides the basic tools which could give it quite as many batteries as Python. With bytevectors for memory-access, it could have something similar to numpy. But to get closer to that, it would need a common API for interfacing with big C-arrays. An API which keeps the C-arrays as reference (for example using methods described in the Guile manual under Accessing Arrays from C), so they can easily be passed to libraries and returned from foreign functions with negligible runtime cost and which provides easy ways of slicing and mathematical operations (like numpy).

If every GNU library provided a Schemish interface for Guile, that would go a good way towards providing powerful batteries - especially because GNU already provides powerful mathematic libraries. So the basics are in place, but Guile will need a lot of good and hard work to reach the state of Python. And that means more exposure to programmers who use Guile for real work and complain about the remaining unpolished corners.

To realize a consistent way for accessing foreign libraries, this also needs a definition of what constitutes Schemish code. Most Guile developers follow the guidelines in Riastradh's Lisp Style Rules,[12] as well as code examples from GNU Guix.[13]

On the other hand, the FFI interface already looks similarly elegant as interfaces written in cython, but without the need to compile it.

5.5 Does it scale up?

I cannot yet say whether Guile scales up for certain, because the scaling will only show after prolonged usage.

[12]Riastradh's Lisp Style Rules is an article which provides rules with rationales for formatting Lisp code, with some parts focussed specifically on Scheme: `http://mumble.net/~campbell/scheme/style.txt`

[13]GNU Guix is a distribution built on Nix which uses GNU Guile Scheme for implementation and package definitions. It treats packages as pure functions and ensures a consistent system state even when the computer loses power during an update: `http://gnu.org/s/guix`

But I already found some information which suggests scaling properties of Guile.

Basic impression

I miss the namespace-by-default from Python, though I can add namespaces to modules and import only specific bindings. For bigger projects, I can just write my own import wrapper which adds namespaces by default - and the same goes for almost any other limitation of Guile Scheme. But every change will make it harder for newcomers to understand the code.

On the other hand, code using let-recursion should scale much better than for-loops, because it makes it easier to extract parts of the loop. Let's look at some specifics.

Positional arguments and keyword arguments

Different from Python, an argument in GNU Guile can either be a required positional argument, an optional positional argument or a keyword-argument, but not several at the same time.

```
use-modules : ice-9 optargs
define*
    example required1
          . #:optional opt1 (opt2 'opt2default)
          ; optional must come before #:key!
          . #:key named1 (named2 'named2default)
          . . rest
    display
        list 'required1: required1
             . 'opt1: opt1 'opt2: opt2
             . 'named1: named1 'named2: named2
             . 'rest: rest

example 'foo #:named2 'bar 'baz
newline ; prints (required1: foo opt1: #f opt2: opt2default
        ;         named1: #f named2: bar
        ;         rest: (#:named2 bar baz))
```

In Python, positional arguments always double as keyword-arguments, so a user of a library can explicitly state in a function call which meaning the different passed arguments will have, but there are no optional positional arguments without default value (in Guile those default to #f). This makes it very easy to call functions in a readily understandable way. On the other hand, this means that Python makes the function-internal names of positional arguments part of the exposed API of the function. Changing them means changing the API – and as such potentially breaking external code.

So I cannot decide which of these approaches is better for scaling. Python seems more convenient for the user of libraries and makes it easier to transition from positional arguments to keyword-arguments when the function signature becomes unwieldy. But locking the names of positional arguments into the API also means that a programmer can never change these names to suit changes in the implementation of the function.

So in terms of function-arguments, Python and Guile Scheme make different tradeoffs, but I cannot decide which approach will be better in the long run.

Note that `(define* (func #:key foo . rest) ...)` puts the keywords in the list `rest` (in addition to providing them as variables) instead of using a dictionary of keyword-value pairs and a list, so it can require additional parsing. I think this could benefit from some polish.

Different ways to import modules

When it comes to importing, though, the difference is clearer. GNU Guile offers quite a few different ways of importing modules, while Python sticks to a few default ways.

If you follow the default way[14], Guile gets all bindings without namespace. This is not what I would want, given my experience from Python, but since C does it the same way with #include, it's clear that this does not make it impossible to scale up. Just a bit inconvenient.

```
use-modules : ice-9 popen
```

To make it clear where the bindings in a given module come from, I can import modules with a namespace. This uses the general renamer keyword and I consider it as much more useful

[14]As default way I take the one which Guile uses if you give it no additional arguments: The easiest way for the programmer.

than the default way. Note that I can use any prefix, so I could even unite the bindings from several modules under a common prefix. This would defeat the purpose for me (finding the file from which the bindings originate by looking at the prefix), but Guile Scheme makes it possible and quite easy.

```
use-modules
  : ice-9 popen
    . #:renamer : symbol-prefix-proc 'popen-
```

To make the common case of prefixing easier, there's also a dedicated prefix option:

```
use-modules
  : ice-9 popen
    . #:prefix popen-
```

Also I can restrict the import to specific bindings and rename them individually:

```
use-modules
  : ice-9 popen
    . #:select : (open-pipe . pipe-open) close-pipe
```

And naturally these methods can all be combined.

These methods can mirror all the possibilities from Python and then a few more, but the default way is less suited for scaling up, because it litters the namespace with all the exported functions from imported modules without any prefix. Due to this choice, finding the origin of a binding requires either IDE support, checking at runtime or looking into all imported modules.

On the other hand, Guile encourages exporting only selected functions from a module as explicit API, and it allows mixing several modules under the same prefix – a capability which Python only made default in 2012 (version 3.3) as “implicit namespace packages” (PEP-340) which added a complexity to the import process which kills my usecase of simply deriving the module filename from the module name.

But still I would wish for a default which adds the name of the module to all imported bindings. On the upside, with Guile it is possible to add this default myself on a per-project basis.

identifier-syntax: getters and setters for variables

Guile Scheme provides identifier-syntax which works like Python properties: simple variables to which I can later add getters and setters, one of the big scalability assets of Python.

This does not (yet?) work with Wisp because that adapts the reader, but if you use parenthesized Scheme, you can do this:

```scheme
(define y 5)
(define-syntax x
  (identifier-syntax (var y)
                     ((set! var val)
                      (set! y (+ 1 val)))))
(write x) ; -> 5
(set! x 5)
(write x) ; -> 6!
```

This enables you to define an API with the simple variable `x` and if you later want to add constraints on the values of `x` or retrieve the value from some datastructure or hidden variable, you can easily do so without changing the exposed API.

Adapting the syntax to the problem

With Macros, Guile allows adapting most aspects of the language to the task at hand. For small projects, this can provide a solid core of tools which make programming more enjoyable. While the project is small, these serve as syntactic sugar for convenience.

When the project grows however, the impact of these tools can become much bigger, as they allow cutting unnecessary overhead at every step.

If they are well-designed, they can make it much easier to scale a project from hobby-tinkering to production quality.

When the project I wrote for evaluation in my PhD thesis grew beyond its initial scope I had to turn to very dark corners of Python to keep it maintainable. And despite the elegance of their design, even great web frameworks like django always expose a certain degree of ugliness as they struggle to realize their goals in the constraints of Python (though in my still limited experience this is much, much worse with C-based projects). I'll call these constraints warts - similar to the usage of that term in the Python-community when it comes to the core-language.

With Guile it is possible to avoid most of these warts, because the basic constraints of its syntax are much smaller than the constraints of Python.

On the other hand, this makes the design of macros more challenging, because they can affect everything you do, and badly designed macros can create much more ugliness than allowing the warts of Python to creep in. So while the macros should make it much easier to scale a project written in Guile than to scale a project written in Python, it can make more sense to rely on standard syntax in the beginning and only start playing with macros when you gathered experience with the requirements of your project. But then, to learn using macros effectively, you have to experiment with them - which is a bit of a catch-22. If you're interested in all the nifty tricks you can do with macros and syntax-transformation, it might therefore be best to start with a breakable toy.[15]

Good practice is needed! (but not enforced)

As you can see from all the different ways to import bindings and to generally tackle problems in Guile Scheme, good practice becomes much more important than in Python. While in Python you say

> *"Don't use `from os import *` "* — Good practice with Python

Guile-using development teams have to give clear code-guidelines to get a consistent codebase.

On the other hand, you can experiment with better ways to work and move programming paradigms forward without having to invent your own new language from scratch.

[15] A breakable toy is a basic building block from Apprenticeship Patterns in which you choose a project which is interesting but which is not needed in production, so you can play with it without fearing to lose something if you break it. It allows easy experimentation and faster learning due to being able to try new things without having to get them perfect right away and without having to polish them for use by others: `http://chimera.labs.oreilly.com/books/1234000001813/ch05.html#breakable_toys`

With this, Guile Scheme allows you to make code-structures scale which do not scale well in Python.

5.6 Summary

A first look at Guile Scheme through the lens of the strengths of Python shows a much less polished language. Instead of having one easy way to do something, every developer can take his or her own path, making it harder to understand code from your fellows, and where Python greets the beginning programmer with readily accessible default structures, Scheme exposes minimal features which beg to be extended, while many advanced but easily understandable structures are hidden in modules people have to find before they can use them.

On hackability Python actually makes it easier than Guile Scheme to find ways for hacking on deep concepts by simply using autocompletion in the shell – and with very nice documentation – but when we look at the reasons why such hacks are used, Scheme becomes far easier, because many of the actions which feel like hacks in Python are cleanly integrated in Guile Scheme, or can be realized using only core functionality.

Following the lifecycle of a program also looks more feasible in Scheme, though I guess that it requires deviating from the simplest way of doing something. When programs get bigger, the syntax-adjustments in Scheme should start to pay off more and more, though these require discipline from the programmers to avoid locking themselves into a bubble of language concepts which are alien to newcomers.

Similarly while Guile provides fewer batteries, it is possible to build more batteries with the simple Foreign Function Interface (FFI). With this method, wrapping a library is about as convenient as doing the same with cython. I did not find a similarly powerful and consistent interface as the one which numpy provides to access numerical datasets in Python, though bytevectors might provide a good base to start. All the basics seem to be in place, but they need lots of solid work to get close to Python in terms of directly usable and consistent bindings.

In terms of executable pseudocode, Scheme shines (at least after taking care of the parens). Some of its names seem unintuitive to me, but its very consistent and easy structure makes it even more accessible than Python - especially for people who do not come with a huge can of preconceptions from C-like languages. This is enhanced when using the curly-infix extension (SRFI-105) which allows using regular infix notation for mathematics.

After that initial very good impression, the ride gets a little bumpy with unusual naming and some really mindbending features, until the advanced capabilities of Scheme come into bearing and allow clean and simple solutions to challenges which in Python require ugly hacks.

But before we go into these neat solutions and take programming beyond Python, there are some dark places to visit.

Chapter 6

Guile Scheme Shortcomings

Guile Scheme is a solid language, and it can compete quite well with Python, even in the areas where Python is strongest. But there are some dark corners I did not mention yet. Here I will explore the worst shortcomings I found in Guile Scheme.

6.1 creating languages *and* solving problems

As written in `One way to do it?`, Guile Scheme is harder for newcomers than Python. And I think I can see (part of) the reason for that.

Different from Python, which is clearly focussed on solving problems, Guile Scheme has a dual focus: Creating new language structures and solving problems with the same language. And from my current experience, the focus on creating languages is stronger than the focus on solving problems.

This leads to a mix of high-level and very low-level operations and less than ideal naming of functions.

Guile Scheme is a wonderful ontology which encodes the common tasks for creating new language structures, but its structures for solving general problems are ripe with inelegancies like using the name `list-ec` (collect into a list) for list comprehensions or `in-list` for looping over a list instead of simply using `in`.

To get this polished, it will need lots of real life usage to straighten out rough edges and find which convenience functions are needed in practical work.

What I am also missing which could make this much easier is a guide which combines the most elegant structures in Guile Scheme into a canonical way to solve problems. It is possible that it already exists, but I did not see it yet - and such a guide should be the first point of contact for a new Schemer.

6.2 car and cdr: Implementation details in the language

Similar as with `list-ec` and `in-list`, implementation details creep into high-level Scheme code at many points. The most visible example are `car` and `cdr` (read as “coulder”).

`car` and `cdr` are used in Scheme for accessing the head and the tail of a list (and some other data structures), and while their use quickly becomes second nature (because they are used all the time in recursion), their meaning is completely opaque to newcomers.

Let’s show that with the example of `rember` (remove member) from the book The Little Schemer[1, 2] (slightly adjusted to make it accessible without context):

```
define : rember a l
    cond
       : null? l
         quote ()
       : equal? (car l) a
         cdr l
       else
         cons : car l
                rember a : cdr l
```

This function walks through the list `lat` and removes the first occurrence of `a`.

Compare this to the same function using `first` and `rest`:

```
define : rember a l
    cond
       : null? l
         quote ()
       : equal? (first l) a
         rest l
       else
         cons : first l
                rember a : rest l
```

[1]This example uses equal? instead of eq? to make the distinction between car/cdr and first/rest more visible. eq? compares object identity while equal? compares the content of a variable. In The Little Schemer `rember` uses eq? because it is defined at a point in the book where equal? is still unknown.

[2]The Little Schemer is a popular and very readable introduction to programming with Scheme and a great start into thinking in recursion: https://mitpress.mit.edu/index.php?q=books/little-schemer

So why are `car` and `cdr` used instead of more speaking names? The origin of their names lies in early machine language:

- car: Contents of the Address part of Register number
- cdr: Contents of the Decrement part of Register number

So their use began as an implementation detail.

According to Wikipedia, they continue being used, because their structure makes it easy to define visually similar functions for multiple applications. For example `caar` is the first element of the first element of a list `(car (car l))`, `cadr` is the second element in a list `(car (cdr l))`, and `cddr` is the tail of the list beginning with the 3rd element `(cdr (cdr l))`.[3]

And from my own experience with The Little Schemer, `car` and `cdr` quickly become something like invocations - an integral part of the "sound" of the code. That doesn't make them less opaque to newcomers, though.

A partial solution to the more speaking names is using `srfi-1`, which provides first, second, third, and so forth - alternatives to car, cadr and caddr. It does not have an equally simple alternative to cdr, though. You have to use the 2-argument function `(drop list i)`, which returns all but the first `i` elements from the list. It would be trivial to define a `drop-first` function which is equivalent to `cdr`, but this is not part of the `srfi-1`,

[3]Going from `cdr` to `cddr` is similar to the way how derivation is written in mathematics: `dx/dt` for the first derivative, `d²x/dt²` for the second, and so forth. In ASCII this can be simplified to `xddt` - In the last 2 years I read plenty of fortran code using variable names like `mdddx`: The third derivative of the mass in x-direction.

and consequently you have to define it yourself for each project or stick to `cdr`.

To also replace `caar`, `cadr` and `cddr`, Guile provides a more flexible alternative in the match-module via Pattern Matching.

```
use-modules : ice-9 match
define l '(1 2 3)
match l
    : first rest ...
      . first
; => 1
```

```
use-modules : ice-9 match
define l '(1 2 3)
match l
    : car cadr caddr
      . cadr
; => 2
match l
    : car cdr ...
      . cdr
; => (2 3)
match '(1 (11 12) 2)
    : car (caadr cadadr) caddr ...
      . cadadr
; => 12
```

Or, for that matter:

```
use-modules : ice-9 match
define l '(1 2 3)
match l
    : fubbly dubbly duh
      . fubbly
; => 1
```

In parenthesized scheme, match-usage looks like this (included here to make it easy to recognize when you see it):

```scheme
(use-modules (ice-9 match))
(match '(1 2 3)
  ((car cadr caddr)
   cadr))
   ; => 2
(match '(1 2 3)
  ((car cadr cddr ...)
   cddr))
  ; => '(3)
```

In general this is not what I would call simple, but it is explicit - and it follows the common theme of being much more powerful than anything I had imagined.

As in other areas, Guile Scheme provides the features I need to easily define my own language while slightly missing the sweet spot for solving common problems. It creates the tools needed to squarely hit the sweet spot, but then does not utilize this power to provide the last 10% of polish for problem-solving.

6.3 Inconsistency, overhead, duplication

And this missing polish is visible in quite a few other areas, too. From my current impression, Guile Scheme is a language of power and necessity, but when it comes to convenient usage, it has quite a few rough corners.

This starts with inconsistent ordering of function arguments, shows up in duplication of functionality for different types and leads to overhead in the function specification to make it usable for multiple usecases.

For example `string-index s char_pred` searches for the second argument within the first, while `string-prefix? s1 s2` checks whether the first argument is a prefix of the second. Also `string-index` takes a character and not a string as second argument, while `is-a?` is used to ask `is-a? 0 <number>`, but not in the ordering `is-a? <number> 0` which would sound more like a typical sentence.

And the duplication shows in `length` and `string-length`: `length` gives the length of a list, but to operate on a string, you have to use `string-length`, even though Guile with GOOPS (the Guile Object Oriented Programming System) is perfectly capable of matching different implementations for different types to the same name.

Finally the overhead can be seen with `format`: Its first argument defines where the formatted string should be written, with `#f` saying that it should return a formatted string.

At least the two later issues could easily be avoided. Adding generic behavior for length just requires 4 lines of code:

```
use-modules : oop goops ; get goops functionality
define-generic length ; allow specialization of length

define-method : length (s <string>)
    string-length s ; just reuse string-length
; that's it. Now this works:
length "123"
; => 3
```

And using GOOPS, format could be crisp for the usual case (I'll arbitrarily take that as returning a string) while providing all flexibility for the general case:

```
use-modules : oop goops
define-generic format
define-method : format (s <string>) . rest
                apply format #f s rest
```

This could be made nicer by adding the `destination` as a keyword argument, but `(ice-9 optargs)` does not define a `define-method*` with keyword argument parsing similar to `define*`. Which is another case where inconsistency crept in.

All of these are just minor stumbling blocks (and I am not the first to write about them: I recently found an article, where Mark Engelberg complained[4] about similar issues in Racket Scheme compared to Clojure). But they make Guile Scheme as a language feel less polished than Python.

6.4 A common standard moves more slowly

Some of the issues in Guile Scheme cannot be solved in Guile itself, because a big part of Guile Scheme is an implementation of the standardized Scheme language, which allows interoperability between different Schemes. This has the disadvantage, that is quite a bit harder to change than a language like Python which has one reference implementation that leads the language design,

[4]In Racket vs. Clojure Mark Engelberg explains why he uses Clojure instead of Racket for professional work: `http://programming-puzzler.blogspot.de/2010/08/racket-vs-clojure.html`

but it gives you the advantage that the skills you learn with one Scheme can readily be adapted for other Schemes, some of which support vastly different problem-domains, like creating tiny standalone binaries for embedded platforms.

The shortcoming this creates compared to Python as language is that many parts of Guile Scheme do not use the advanced features within Guile to keep interoperability to other Scheme-implementations.

Also it is harder to create a lean and elegant standard if this has to be a good fit for multiple implementations with different constraints and target groups. This is one more reason why the language-creation functionality which all the different Schemes need in the same way is top notch, while the problem-solving can be a bit cumbersome in comparison to Python.

6.5 Distributing and Packaging to OSX and Windows

Now we come to the darkest place in Guile development. There is no readymade plan for distributing programs using Guile Scheme to platforms without proper package manager.

For Windows, there are patches for relocatable Guile modules, but these are not yet (as of version 2.0.11) part of the regular Guile install.

Also there are no solutions to distributing games or similar artwork-rich programs which use Guile Scheme as implementation language instead of as extension language. David Thompson (davexunit) is working on that front with Sly. But easy distribution looks different.

This also is a problem with Python, which tools like PyInstaller only solve partially - for example PyInstaller still requires me to run OSX to create an installer for MacOSX - but with GNU Guile it is even more serious. While for Python the shared package repository PyPI makes it very easy to share code with people who can run or modify some code, and most of its packages just work on any platform (which might explain its popularity with web developers and scientists), sharing code for GNU Guile still requires external infrastructure like guildhall,[5] and using it on anything but GNU/Linux is an unsolved problem.

I cannot easily give Guile Scheme programs to people who do not use GNU/Linux, and even for those who do, a fast beta-cycle to non-developers will be hard to achieve.

This could well be the worst shortcoming for my usecase. I'll see how it goes, though. Making Wisp usable directly from the REPL was remarkably easy, so there might be similarly easy ways to enable quick testing which I just don't know yet - if necessary by utilizing autotools (that's what I do in Wisp).

6.6 Summary

Despite being nice to use most of the time, Guile Scheme has some severe shortcomings. Most of them stem from having a

[5] Guildhall provides functionality like PyPI but is not integrated with Guile yet. It adds package management capabilities to the guild command: `https://github.com/ijp/guildhall`

much broader focus than Python: not only on solving concrete problems, but also on tweaking the very core of Guile to make it better suited for the problem. This leads to a mix of primitive and sophisticated structures: there is an extremely flexible object oriented programming system with powerful module methods next to simplistic list modification structures. There is an efficient foreign function interface which allows calling into any C library, but distributing a program written in or with Guile to anything but a GNU/Linux system with a good package manager is a nightmare. And no best practices guide is to be found.

Due to this, Guile Scheme is still quite a few steps away from reaching a point where I can wholeheartedly recommend it to people at my institute.

For some of these points, there is limited flexibility due to keeping compatibility with Scheme code written for other implementations. But most of them can be overcome by lots of polishing and focussed documentation without breaking the language.

And now, after looking into the dark corners of Guile Scheme, it is finally time to uncover its sparkling jewelry: Going beyond Python.

Chapter 7

Guile beyond Python

Where Python takes you on a smooth path from Beginner to Experienced Programmer, Guile accompanies you far beyond that, after you cross over its initial bumps.

I am still a beginner of Guile Scheme myself, so I cannot show you all the ways of taking programing beyond Python with GNU Guile. That's why I invited a few experienced Schemers as guest authors to fill the gap and give you a glimpse into the vast possibilities Guile offers for programmers.

So in addition to my own experience, the next few chapters will quote from the work of Ludovic Courtes (Ludo), Clinton (Unknown Lamer), David Thompson (davexunit), Mark Witmer and Mu Lei (NalaGinrut) to give you an idea of their experience with using Guile Scheme in fancy ways and taking programming beyond the limitations of Python.

We'll begin with recursion and exact math, come to full posix threads and then go deep into the heart of Guile with programmable syntax, definition of completely new but interoperable languages, flexible object oriented programming and definition of control flow operators with continuations and prompts.

7.1 Recursion

Guile Scheme provides beautiful recursion features along with full tail recursion. This means, that you can use recursion to solve problems and have the solution look good. Here is a Guile example (taken from one of my german articles):

```
define : fib n
    let rek : (i 0) (u 1) (v 1)
        if {i >= {n - 2}}
            . v
            rek {i + 1} v {u + v}
```

The equivalent Python-Program looks like this:

```
def fib(n):
    def rek(i=0, u=1, v=1):
        if i >= n-2:
            return v
        return rek(i+1, v, u+v)
    return rek()
```

Time to test them. Let's start with Guile:

```
fib 1
; => 1
fib 11
; => 89
fib 101
; => 573...[18 further numbers]
fib 1001
; => 7033...[205 further numbers]
```

And follow up with Python:

```python
fib(1)
# => 1
fib(11)
# => 89
fib(101)
# => 573...[18 further numbers]
fib(1001)
# => ... RuntimeError: maximum recursion depth exceeded
```

OK, so we cannot do this…

When we want the Python-code to accept bigger input, we must convert the recursion to a for-loop (or change the maximum recursion depth - but that only delays the problem until our memory dies).

```python
def fibloop(n):
   if n in (1, 2):
      return 1
   u = 1
   v = 1
   for i in range(n-2):
      tmp = v
      v = u+v
      u = tmp
   return v
```

This works now. But compared to the beauty of let-recursion it is damn ugly. You could say, that let-recursion in Scheme is just syntactic sugar, because tail-recursion simply does this conversion automatically. But then, *"All Turing-complete languages differ solely in syntactic sugar."* — Michele Simionato in The Adventures of a Pythonista in Schemeland[1]

Let us finish this by repeating the beautiful Guile code:

```
define : fib n
    let rek : (i 0) (u 1) (v 1)
        if {i >= {n - 2}}
            . v
            rek {i + 1} v {u + v}
```

7.2 Exact Math

Where tail recursion lifts limitations in the interaction of functions, exact math lifts limitations for working with numbers. If you ever stumbled into the limits of floating point precision and language, compiler and hardware dependent rounding errors, you'll know it as a really dark place, best to be avoided.

I got hit by these issues when doing binomial calculations with large numbers to estimate the probabilities of finding sufficient numbers of close neighbors in Freenet. My trusty Python script, once written to support a forum entry about the probability that my roleplaying group will have enough people to play, broke down before 4000 elements with

```
OverflowError: integer division result too large for a float
```

[1]The Adventures of a Pythonista in Schemeland is a web-based guide to Scheme for Pythonistas written and published by Michele Simionato.

The reason is simple: There are some intermediate numbers which are much larger than what Python can represent with a floating point number.

Knowing that Guile Scheme provides exact numbers, I ported the script to Guile, and it just worked.

It just worked and used less then 200 MiB of memory - even though intermediate factorials return huge numbers. And huge means huge. Guile Scheme effortlessly handled numbers with a size on the order of 10^{8000}. That is 10 to the power of 8000 - a number with 8000 digits.[2]

Most of the time, such capabilities aren't needed. But there are the times when you simply need exact math. And in these situations Guile Scheme is a lifesaver.

7.3 Real Threads!

Different from Python, Guile uses real operating-system threads. Where a threaded Python program becomes slower with more processors due to issues with synchronization between processors, Guile can fully utilize todays multicore computers.

Starting is really easy: Just use futures. Here's an example:

[2]The ease of using exact math in Guile impressed me so much, that I wrote an article about my experience: Exact Math to the rescue: `http://draketo.de/english/exact-math-to-the-rescue`

```
use-modules : ice-9 futures
              srfi srfi-1 ; for better iota
define : string-append-number l
         apply string-append : map number->string l
let loop ; create a long loop. Guile is ridiculously fast
         ; with math, so we have to make this expensive
         ; to see an effect of concurrency.
    : i : iota 1000
    when : not : null? i
      let : : l : iota 1000 1 ; starts at 1
        let ; worst case: futures in the inner loop.
          : a : future : string-append-number l
            b : future : string-append-number : map log l
            c : future : string-append-number : map sqrt l
          ; touch gets the result of the future
          apply string-append : map touch : list a b c
      loop : cdr i
```

This code runs at 220% to 240% CPU load on my 4-core machine (ideal would be 300%) and the runtime decreases by roughly 50% compared to a strictly sequential program, which is pretty good for a tight inner loop. Note that with futures Guile automatically uses a thread pool.

7.4 Programming the syntax and embedded domain specific languages

Due to the integral role syntax adaptions take in Scheme, making an optimal domain specific language (DSL) with minimal effort while leveraging all the power of Guile Scheme to form an *embedded domain specific language (EDSL)* is just a matter of pattern matching.

Ludovic Courtès describes the benefits of using an EDSL for packages in GNU Guix in the description for his Fosdem-Talk Growing a GNU with Guix (video):[3]

> Packages are declared in a high-level fashion, using a domain-specific language embedded in the Scheme programming language. This is the first step in making it hackable to our eyes: packagers do not even need to know Scheme to write a package definition, yet the full power of Scheme, of GNU Guile, and of the Geiser programming environment is available.
>
> From a programming viewpoint, Guix and the GNU distribution are just a bunch of "normal" Guile modules, some of which export "package" objects—one can easily write Guile code that builds atop the distribution, customizes it, or otherwise fiddles with packages.

The GNU Guix embedded domain specific language looks like this:

[3] Growing a GNU with Guix at FOSDEM 2014: `https://fosdem.org/2014/schedule/event/gnuguix/`, video: `http://video.fosdem.org/2014/H1302_Depage/Sunday/Growing_a_GNU_with_Guix.webm`

```
(define-module (gnu packages which)
  #:use-module (guix licenses)
  #:use-module (guix packages)
  #:use-module (guix download)
  #:use-module (guix build-system gnu))

(define-public which
  (package
    (name "which")
    (version "2.20")
    (source
      (origin
        (method url-fetch)
        (uri (string-append "mirror://gnu/which/which-"
                            version ".tar.gz"))
        (sha256
         (base32 (string-append
                  "1y2p50zadb36izzh2zw4dm5hvd"
                  "iydqf3qa88l8kav20dcmfbc5yl")))))
    (build-system gnu-build-system)
    (home-page "https://gnu.org/software/which/")
    (synopsis "Find full path of shell commands")
    (description (string-join
      "which is a program that prints the full paths"
      "of executables on a system."))
    (license gpl3+)))
```

This realizes a similar goal as ebuild files for Gentoo, but where the ebuild files are run with a specialized scriptrunner providing domain-specific functionality, Guix packages are just standard Scheme code. Compare the above package definition with this ebuild and manifest:

```
# Copyright 1999-2014 Gentoo Foundation
# Distributed under the terms of the GNU General Public License v2
# Header: ...

EAPI="4"

inherit flag-o-matic toolchain-funcs

DESCRIPTION="Prints out location of specified executables \
that are in your path"
HOMEPAGE="https://carlowood.github.io/which/"
SRC_URI="https://carlowood.github.io/which/${P}.tar.gz"

LICENSE="GPL-3"
SLOT="0"
KEYWORDS="alpha amd64 arm arm64 hppa ia64 m68k ~mips ppc \
          ppc64 s390 sh sparc x86 ~ppc-aix ~x64-freebsd ..."
IUSE=""

src_prepare() {
        append-lfs-flags
}

src_configure() {
        tc-export AR
        default
}
```

```
...
DIST which-2.20.tar.gz 135372 SHA256 d417b... WHIRLPOOL 35ca3...
...
SIGNATURE
...
```

Note: I (the author of the book) am a longterm Gentoo user.

For details on adapting the Syntax in GNU Guile, see Syntax-Rules in the Guile reference manual.

7.5 New Readers: Create languages with completely different syntax

Guile allows defining new languages with completely different syntax and using them in concert with the existing languages in Guile. Examples include Javascript, Emacs Lisp and Wisp at the REPL. Guile realizes this by defining new readers: The first processing step in parsing code. But there are usages apart from just making it possible to use popular languages inside Guile.

Multi-Language interface definitions

As a really cool example which highlights the potential of extending the reader to solve real-life challenges, Mark Witmer, author of guile-xcb, describes his experience from implementing asynchronous X11-bindings by directly using the XML definition files as library - which as task description sounds almost unreal. But let's give him the stage:

Guile-XCB is a library that provides Scheme bindings for the X11 protocol, the foundational layer of graphical user interfaces in most Unix-like operating systems.

The X11 protocol is a format for sending messages back and forth between a client that uses graphics and input devices, and a server that manages the hardware. These messages are defined in a very long and detailed English-language document. That raises the question: what is the easiest way to turn this document into working code?

Some clever and dedicated people created a set of XML files that describe the binary format used in the core protocol and many of its extensions. This is the heart of the XCB (X protocol C-language Bindings) project. To make a C library that uses the XML files, they wrote a Python library that reads the XML files in and spits out C code and header files.

Things are a little different in Guile-XCB. Thanks to Guile's support for new language implementations, the XML files themselves are source code for a language that compiles down to the same object code format that regular Guile programs use. No need for a separate Python script or complicated non-standard build logic.

The entry point to defining the new language is in the module `(language xml-xcb spec)` and looks like this:

```scheme
(define-language xml-xcb
  #:title "xml-xcb"
  #:reader custom-read
  #:compilers `((scheme . ,compile-scheme))
  #:make-default-environment make-default-environment
  #:printer write)
```

The procedure custom-read turns XML into s-expressions using the built-in sxml library and the procedure compile-scheme runs through the expression and generates record types for all the requests, replies, and complex data types that are defined in the XML files. All that's needed to compile an XML file is this command at the terminal:

```
guild compile xproto.xml --from=xml-xcb --output=xproto.go
```

With the help of a few modules that handle X connections and send and receive requests and replies, Guile-XCB turns the XML files into modules that you can load just like any other Guile module, without requiring any FFI or C-language bindings.

Developing new programming languages

A more experimental usage of reader extensions is development of completely new languages - or reviving old languages by giving them access to the full capabilities of GNU Guile. As a practical example, Mu Lei aka NalaGinrut describes his experience with implementing a simple math language: Simple, but not so simple.[4]

> This article is about the front-end only: lexer and parser, and transforming a simple AST (actually it's a list type in Scheme) to another kind of AST, tree-il, the first level of Guile intermediate language. After the tree-il was generated, the rest of the compiling work would be taken by Guile.
>
> So we don't have to face the complicated compiling optimization stuffs. This feature makes it very easy to implement new languages in Guile.
>
> ... Simple, but not so simple ...

[4]Simple, but not so simple is a blog post about implementing a new language in Guile with 50 lines of code: `http://nalaginrut.com/archives/2014/04/15/simple,-but-not-so-simple`

’simple’ is just a simple language, maybe too simple for a serious compiler writer. Formally even a front-end would take you a lot of time and hack power. Not to mention the backend. Fortunately, Guile provides a nice way to let language fans focus on the grammar rather than optimization. Nevertheless, all the language front-ends can call from each other, If you’re interested in this feature, please read Wingo’s post on Ecmacript in Guile, and inter calling between Ecmascript and Scheme.[5, 6]

7.6 Your own object oriented programming system

The Guile Object Oriented Programming System (GOOPS) allows changing the OOP system to your liking. For example Clinton aka Unknown Lamer did some quite nifty OOP experiments

[5] In the article ecmascript for guile Andy Wingo explains in fun and approachable style how to run (strictly written) JavaScript in Guile and access Guile features from Javascript: `http://wingolog.org/archives/2009/02/22/ecmascript-for-guile`

[6] Using the ECMAscript language in Guile allows executing (clean) Javascript while retaining the full power of Scheme, including all functionality written in other Guile languages. To experiment with it, start a recent Guile (2.0.11 or newer) as `guile --language=ecmascript`. This enables you to do this: `require("srfi.srfi-1").iota(10, 5, 7); //` *ftagn*.

with Guile. As a first stop he suggests having a look at serialize.scm,[7] which contains *"some grade A GOOPS abuse"*:

> Multimethods? The meta-object protocol allowing you to mold the object system to your needs?
>
> Using GOOPS, you can define your own object systems! Who wouldn't want to change the fundamental behavior of the language ;-)

Compared to the complexities of adjusting Python's object-system, this allows going outside the usual realm with ease - with mixins instead of inheritance being just one of the simplest applications. And when an often-used definition gets cumbersome, you can utilize macros to make it convenient again.

7.7 Continuations and prompts

In the chapter *One way to do it?* I complained that solving tasks in a multitude of ways in Scheme makes it harder to read code. Now it's time to turn this around: Guile's delimited continuations via prompts allow implementing advanced control structures in an efficient and elegant way. By default Guile uses

[7] serialize.scm by Unknown Lamer uses GOOPS to serialize and unserialize objects of various types: `http://unknownlamer.org/darcsweb/browse?r=guile-web;a=headblob;f=/src/serialize.scm`

them for exception handling[8] via throw and catch, but much more is possible.

One example for those structures are coroutines: Functions which cooperatively share processor time by stopping their execution at given points and deferring to another function, until that other function passes control back to the original function.

Many words for simple code: In Sly[9] you can define the movement of a character as follows:

```
use-modules : 2d agenda
              2d coroutine
              2d game

coroutine
  while #t
    walk 'up
    wait game-agenda 60
    walk 'down
    wait game-agenda 60
```

In short: walk up, then defer to the function `game-agenda` and ask it to pass back control to the coroutine-code 60 seconds later. When `game-agenda` passes control back to the coroutine, walk down and pass back control to the `game-agenda`.

If I wanted to do something similar in Python, I would have to create an iterator which is called by the game-agenda and *yields* the time to wait after every step. I'd then have to run

[8] In the chapter exception handling the Guile manual explains what exceptions are and how to use exception handling in GNU Guile: `http://www.gnu.org/software/guile/manual/html_node/Exceptions.html`

[9] Sly is a game engine for Guile Scheme: `http://dthompson.us/pages/software/sly.html`

every function by passing it as argument to the game-agenda (this is a generalization of what the Python game library pyglet does for scheduling).

Guile does not have this limitation. A function can actually call another function to defer its control flow to that other function. “Hey game agenda, it’s your turn now. Please pass control back to me in 60 seconds”. And that allows using programming constructs easily which are a hassle to use with Python.

7.8 Summary

Guile Scheme provides functionality which makes it easy for every programmer to go far beyond the limitations of Python.

While elegant recursion support, exact math and real threads provide incremental improvements over Python, redefinitions of the syntax and of concepts from object oriented programming allow shaping the language into something very different. To go even further, continuations and prompts make it possible to create completely new control flow paradigms without ever exiting from Scheme.

There is no need to wait for something like the new `yield from` keyword in Python 3.3: In GNU Guile you can add such new control flow operators yourself, as the example of Sly shows, and have them integrated in the language just as nicely as all the constructs from the mainainers of GNU Guile. And with this, you can turn it into the perfect solution for the task you want to solve. GNU Guile gives programmers a freedom similar to that which users gain from running free software: Independence from the language designer. If something does not behave as you need it to, you can fix it without having to switch to a new system.

And you do all this in the language you also use for general programming. There is little need for mental context switches while you are working. No matter whether you write a simple string-processor or modify the very core of your programming environment: You are always using GNU Guile.

Part III

Conclusions

Chapter 8

Guile Scheme is coming back

While Python is a good choice for new programmers and a great tool for solving practical problems, thanks to providing a complete set of functionality for any kind of task, encoded in a minimal set of highly polished basic concepts using very readable syntax, the structures of Python are almost stretched to their limit and extensions like list comprehensions and decorators make it more and more complicated for newcomers to really understand code from others. This makes Python yet another example for Greenspuns 10th Rule:

> *"Every sufficiently complex application/language/tool will either have to use Lisp or reinvent it the hard way."* — Greenspuns 10th Rule

Guile Scheme on the other hand has a higher barrier of entry and suffers from some rough edges. But when actually doing a comparison between Guile and the strongest points of Python, Guile looks quite good. It does not have the *one best way to do it*, which Python promotes, but that's part of the reason why I started looking into Guile Scheme: The wish to go beyond the way of Python. Scaling a program from a first draft to a big application looks easier with Guile, and while the parentheses look odd at first, it's extensions for infix-math and for indentation-based syntax make it a better choice for pseudocode than Python. Its standard library is much smaller than the batteries of Python, but that is partially made up for by offering an easier way to call C-libraries.

There are severe shortcomings, though. Some come from pursuing two goals at the same time: Language design and solving problems. This leads to a mix of low-level, high-level and deprecated concepts baked into the language on equal footing - as well as some baggage from compatibility to the Scheme-standard which does not allow using Guiles advanced features like easy keyword-arguments throughout. And there is no best practices guide to be found. The biggest challenge, however, is deploying a Guile-based program to platforms which do not have a decent package manager.

But these shortcomings are more than compensated by its strengths. Let-recursion (named let) is not only a testament to the elegance of recursion, but also to the reward for letting programmers define the building blocks of their language as they use them, from basic tools like elegant loop structures up to generic functions, exception handling, coroutines and boundless other possibilities. And new readers allow providing all these capabilities to new languages for specialized tasks - like parsing XML files to implement protocols directly from specifications as done in guile-xcb - while exact math has your back when other systems would fail.

Last, but not least, direct access to real threads (and consequently also truly concurrent futures) provides crucial capabilities in the current times in which even mobile phones come with multiple processors.

As a related note, learning Scheme with The Little Schemer made understanding C++ Template recursion as for example described in “Modern C++ design” (my latest reading) a breeze. So even if you don’t expect to be using Scheme to solve problems in real life, I can wholeheatedly recommend learning it to get an understanding of the possibilities programming can offer beyond Python. I expect to see more and more of its features turn up in other languages, so, if nothing else, learning Scheme will be a very worthwhile investment to prepare for the future of your favorite language.

And when it comes to Scheme, GNU Guile is a very good choice which showed that it can withstand the onslaught of time which pushed so many other languages and systems into oblivion. Guile already had many different maintainers, and it is likely that it will keep being improved in the forseeable future – similar to GNU Emacs, which is still moving forward after 30 years of development. GNUs may not always be the fastest movers, but they sure are stubborn - and that’s a very good quality to have in the core of our systems.

To sum this up there’s nothing better than the well-known quote from Victor Marie Hugo:

Nothing is as powerful as an idea whose time has come. — Victor Marie Hugo

Keep an eye on Guile Scheme: It is coming back.

Part IV

Appendix

Appendix A

See also

A.1 Tools, Projects, Articles

- GNU Guile: `http://gnu.org/s/guile`
- Guile Basics: `http://draketo.de/proj/guile-basics/`
- wingolog: `http://wingolog.org/`
- chaos code with scheme:
  `http://www.nalaginrut.com/archives/2014/03/27/a-way-to-write-shi`

A.2 Recommended Reading

- Learn Scheme in 15 minutes from the Artanis web framework: `http://web-artanis.com/scheme.html`
- A Rundown of Lisp: `http://taylanub.github.io/doc/lisp-rundown.txt`
- The Guile Manual: `http://www.gnu.org/software/guile/manual/html_node/index.html`

- The Little Schemer, The Reasoned Schemer, The Seasoned Schemer and The Little Prover by MIT Press.
- sicm with guile-sicm (exploring classical mechanics with Guile Scheme):
  `http://www.cs.rochester.edu/~gildea/guile-scmutils/`

Appendix B

Glossary

Terms used by Schemers.

- Procedure: Function.
- Parens: Parentheses (round brackets)
- Thunk: One block of code. Enclosed by parens.
- Body: All the forms in a procedure.
- Form: Something which appears in a body: A definition or an expression.
- Definition: A form which starts with `(define`.
- Expression: Any form which can appear in the function body which is not a definition. See R4RS for the clear definition.
- RnRS: Revision n Report for Scheme

- SRFI: Scheme Request for Implementation (spelled as "surfie"). Like PEP.

Appendix C

Solution Map

C.1 File as Module and Script

In Python you use a runtime switch with magic variables at the bottom:

```
if __name__ == "__main__":
    # your code
    pass
```

In Guile Scheme you use shell deferring at the top:

```
#!/bin/sh
# -*- scheme -*-
exec guile -e main -s "$0" "$@"
# Thanks to exec, the following lines
# are never seen by the shell.
!#
(define (main args)
  (display "Hello World!"))
```

C.2 Output a datastructure to console to put it in the interpreter

In Python you use print and paste the result into the shell:

```
print [1, 2, 3]
```

```
print eval("[1, 2, 3]")
```

In Scheme you use write and paste the result into a `(quote ...)` form:

```
(write '(1 2 3))
(newline)
```

```
(write (with-input-from-string "(1 2 3)" read))
```

Both versions have corner cases, but work well for many situations. For custom classes Python requires defining a `__repr__` or `__str__` function which returns a string that can be `eval()`'ed to the same class.

C.3 help in interpreter

In Python you call `help(...)`

```
help(help)
```

In Guile Scheme you also call `(help ...)`

```
(help help)
```

Note: `(help help)` does not work within org-mode.

C.4 Profiling

In Python you can use `python -m profile -s cumtime path/to/file.py` or timeit:

```python
import timeit
print timeit.timeit("1 + 1", number=1000000)
```

In Guile Scheme you can use ,profile in the interpreter. For fast calls you need to use a loop. Alternatively you can use the statprof module:

```
use-modules : statprof
with-statprof #:loop 1000000
    + 1 1
```

Appendix D

GNU General Public License (GNU GPL)

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

1. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

2. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

3. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

4. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

5. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

6. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
   b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

7. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

8. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

   b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

   c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

   d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

   e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

   f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

12. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or

16. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

17. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

18. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

    End of Terms and Conditions

    How to Apply These Terms to Your New Programs

    If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

    To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
```

It starts with a laudation for Python, the first programming language I loved.

In my first years of programming I thought that I'd never need anything else.

‘Beyond Python.’

Then it dives into Guile Scheme.

Where Python takes you on a smooth path from Beginner to Experienced Programmer, Guile accompanies you far beyond after you cross over its initial bumps.

Join me on my path into Guile.

• Cover Illustration by VHR, Michael Gil, Martin Grabmüller, Arne Babenhauserheide •