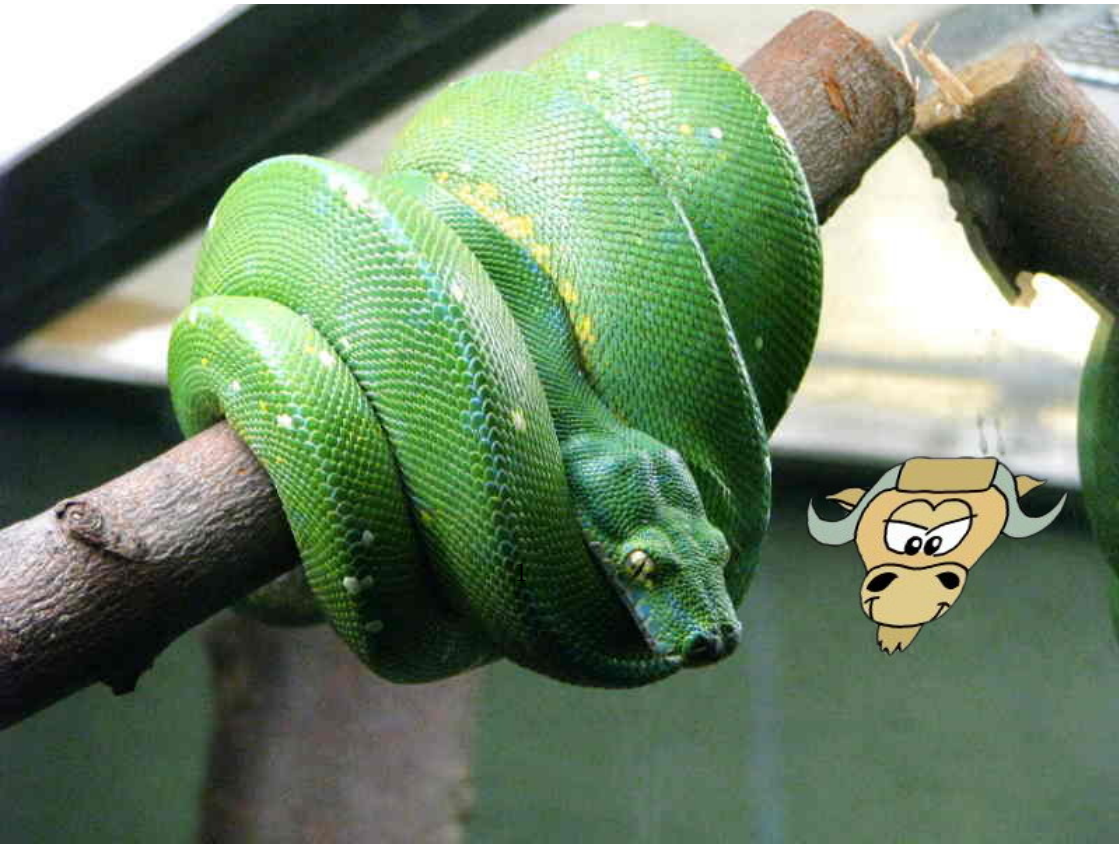
Going from Python to Guile Scheme
A natural progression

Arne Babenhauserheide

February 28, 2015

Abstract

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



The title image is built 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.

This book is licensed as copyleft free culture under the GPLv3 or later. Except for the title image, it is copyright (c) 2014 Arne Babenhauserheide.

Contents

Contents **3**

I My story **7**

II Python **11**

1 The Strengths of Python **13**
1.1 Pseudocode which runs 13
1.2 One way to do it 14
1.3 Hackable, but painfully 15
1.4 Batteries and Bindings 17
1.5 Scales up 17

2 Limitations of Python **19**
2.1 The warped mind 19
2.2 Templates condemn a language 20

2.3 Python syntax reached its limits 21
2.4 Time to free myself 23

III Guile Scheme **25**

3 But the (parens)! **29**

4 Summary **33**

5 Comparing Guile Scheme to the Strengths of Python **35**
5.1 Pseudocode 36
General Pseudocode 36
Consistency 37
Pseudocode with loops 40
Summary 44
5.2 One way to do it? 44
5.3 Planned Hackablility, but hard to discover. 50
Accessing variables inside modules 51
Runtime Self-Introspection 51
freedom: changing the syntax is the same as regular programming 58
Discovering starting points for hacking 60
5.4 Batteries and Bindings: FFI 61
5.5 Does it scale up? 63
Positional arguments and keyword arguments . . 64
Different ways to import modules 65
Adapting the syntax to the problem 67

Good practice is needed! (but not enforced) . . . 68
5.6 Summary 69

6 Guile Scheme Shortcomings 71
6.1 creating languages *and* solving problems 71
6.2 car and cdr: Implementation details in the language 72
6.3 Inconsistency, overhead and duplication 76
6.4 A common standard moves more slowly 78
6.5 Distributing to OSX and Windows 79
6.6 Summary 80

7 Guile beyond Python 81
7.1 Recursion 82
7.2 Real Threads! 84
7.3 Programming the syntax and embedded domain specific languages 85
7.4 New Readers: Create languages with completely different syntax 89
Multi-Language interface definitions 89
Developing new programming languages 91
7.5 Your own object oriented programming system . 92
7.6 Continuations and prompts 93
7.7 Summary 94

IV Conclusions **97**

V Appendix **103**

8 See also (links) **105**

9 Glossary **107**

Part I

My story

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 then started going 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.

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:

> *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.*

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. Ac-

tually I mostly just 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.

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 which answers some of the questions I faced when starting my dive into Scheme.

Part II

Python

Chapter 1

The Strengths of Python

But we need to know where we are, before we can understand new experience. So, before I go into my experience with Scheme, let's start with Python.

> Pseudocode with one right way to do it and hackable, scalable batteries.

1.1 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:

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

1.2 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.*

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 this philosophy creates nicely usable APIs and minimizes guesswork - and when you 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.

1.3 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 in real-life.

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 functions:

```
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 mostly 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:

```
# 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 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.

1.4 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.

```
import antigravity
```

Most of those 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 with it. By this 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.

1.5 Scales up

And finally, Pyhon 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 2

Limitations of Python

With all its strengths, Python is still a language with a limited syntax. It is very broadly applicable, but it has hard rules how things can be done. And these rules can hamper some tasks - they create a straightjacket you cannot escape easily. Most of the time, its chains are very convenient, and they often help when you develop code in a community, but regardless of the type of chains you choose, they can never be convenient for all problems.

2.1 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.

2.2 Templates condemn a language

```
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 for regularly needed tasks. When you start thinking about using code-templates in your editor to fullfill the requirements of your language, then something is wrong with the language.

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.[1]

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.

2.3 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 (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:

[1] 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: Text written by humans for humans, without a computer in the mix.

```
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 it since Python 2.5 (PEP-308), but it is written like this:

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

And that just looks alien. 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,[2] but in real-life I see people recommend the previous version):

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

This isn't the simply Python syntax anymore. The uniformity of the syntax is broken. And this is not necessary. For example with Scheme, where all code is just a tree of evaluated

[2] "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.

expressions, I can simply write the following - and it looks the same as all other code:

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

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

```
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)]
```

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

2.4 Time to free myself

And this brings us back to The White Belt from Apprenticeship patterns:

> *"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."*

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 was time to free myself from its shackles.

Part III

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.[3] 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 really natural to me. Actually more natural than Python.

[3] Aside from Guile, there are lots of other Scheme implementations. See the opinionated guide to scheme implementations from Andy Wingo for an overview. 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.

The expressiveness of the original syntax 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 experiment was a phenomenal success - but it is starting to break down as people expand it into more and more directions.

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 stomach suddenly says "leave out the commas!", and I actually found 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 suddenly there's a language which does not require commas to achieve that goal.

Chapter 3

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 worse with 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, 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 is currently in draft phase as Scheme Request For Implementation: SRFI-119.

(for some more background why I took that step, see my presentation Why Wisp?)

The previous code-block with parens was indented in the canonical way for lisp. If you write the same code-block without parens and 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 the wisp-preprocessor is implemented in Wisp itself. Being a simple preprocessor, it does not need to know about any features of the underlying scheme implementation, except where those features can add linebreaks which have to be ignored (like multiline strings). With this, the readability is not yet quite on the 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 the 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.

You can even use wisp in the interactive console of GNU Guile (thanks go to Mark H. Weaver and others from #guile!). Just get wisp 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.

Chapter 4

Summary

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

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

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

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 points 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: Onward to the unfair race!

5.1 Pseudocode

With the paren-preprocessing of wisp, Guile Scheme also comes close to runnable pseudocode. Maybe even closer than Python.

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.

Consistency

Similarly, consistency of code is generally considered as one of the huge strengths of Python: When you know one Python-program, you can find the patterns used there in all future Python-programs you read. And here Guile Scheme can even go a step further.

Modern Python offers a kind of dual syntax. The 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:

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

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:

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

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 Python:

```
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, so there is also some degree of duality in this Scheme example, but in Python the inner code structure works very different from 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 consistent), so I think that there is value in having to complementary ways to format your code. Both ways should use a consistent 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 just 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 pretty bad. While 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 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:

[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.

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 While loops are 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.

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

Intuitive? These constructs are not really what I consider intuitive, but they are easy and not really bad. What 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 and than any other kind of loop - so much easier, that I wrote an article about the elegance of let-recursion (in german).

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

```
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 id,name in results]
```

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

```
define names : listcomp name for (id name) in results
```

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

Great Looping Constructs in Guile Scheme

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

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

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

foof-loop and foof-loop 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)
  write element
  newline
```

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
```

To get foof-loop for guile, you need to first install guildhall, a package manager for guile. See Getting Started for a short tutorial.

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 actually 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.

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 for 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. If you program in Python, you'll soon see patterns 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 guile-2d, a really nice game-engine in Guile Scheme, modelled partly after pyglet.

Writing a movement pattern for a character in guile-2d 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 actually have to look deep into the nexus of 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.

```
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 procedure.
  call-with-prompt 'coroutine-prompt thunk handler

; this is definition of the coroutine macro
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 - 23 when 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 sim-

ply 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 meat 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 procedure.
  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.[2]

[2] `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...

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

While there are countless ways to do a task in Guile Scheme, when it comes to hackability there are big differences to Python.

Python exposes most of its inner workings via double-underscored attributes. As such, almost every 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. At the same time, hacking Guile and hacking Python are two completely different things.

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-module' or 'resolve-interface'. 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.

A rough equivalent to pythons `dir(math)` is

```
module-map
  lambda (sym var) sym
  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 from 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__`.

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

In Scheme you do not use magic attributes as in python, but 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. 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))))
  (when (or (not my-local-name) ; catch the case when the compiler
                                ; optimized away the info.
      (string-suffix? (car (command-line)) my-local-name))
    (display "you executed this as script.")))
```

The limitation never to repeate 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.

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"))
```

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

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

```
for script in runscript-shell.scm runscript-env.scm runscript-meta.scm; do
    echo real $script; for i in {1..100}; do
        time ./${script} >/dev/null;
    done
done 2>&1 | grep real | sed "s/.*0m.*\\.0//" | sed s/s$//
```

The runtimes were 20-22s for when deferring to shell, 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 3ms - which I would call mostly negligible given that the python equivalent of this script requires 20-24ms, too.

One problem could be that the startup time for the shell method is a bit unstable: On some runs it jumps up to 40ms. To check that, we can look at the average value and the standard deviation in 100 runs:[3]

- shell-deferring: 24.1 ± 4.6 ms
- env: 21.1 ± 2.7 ms
- meta-trick: 18.3 ± 2.4 ms
- python (with `if __name__...`): 23.8 ± 1.9 ms

[3] According to Andy Wingo's writeup on Elf in Guile, guile 2.2 should reduce these startup times quite a bit.

I guess that the additional uncertainty about the startup time when using the shell-deferring is due to filesystem-access and caching, but I did not trace it further. The additional 3ms of the shell-deferring is just the startup time of bash when started from bash.

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 likely 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__
# or rather
print __import__(log.__module__).__file__
```

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

```
define oip : module-ref (resolve-module '(ice-9 popen)) 'open-input-pipe
use-modules : system vm program
display : module-filename : program-module oip
newline
```

To find the absolute path, you need to search `%load-path` for the file-path (a variable holding the list of the load path).

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

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

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

Example for checking whether the file was called itself. Made easy.

This code still has serious issues: If you repeat a filename, the code will be run. But it illustrates the power you get from tweaking the syntax in Scheme. If you find youself tempted to use this, 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.

```
; 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:

```
(define-module (in))

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

this is not triggered when importing in.scm in yet another file!

```
(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.

The approach which currently works for me 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 and do a full-text search with `ctrl-s <search> ctrl-s ctrl-s` (repeating `ctrl-s` tells info to search all subtopics).

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_inc_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_inc_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 elegant. 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 very good and hard work to reach the state of Python.

To realize a consistent way for accessing foreign libraries, this also needs a definition of what constitutes Schemish code.

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 this for certain, because the scaling will only show after prolonged usage.

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

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.

And I do not yet know whether I can use properties in Guile Scheme (simple variables to which I can later add getters and setters).

On the other hand, code using let-recursion should scale much better than for-loops.

Positional arguments and keyword arguments

Different from Python, an agrument 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) ; 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
```

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. 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 arguments part of the exposed API of the function.

So I cannot really 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 func-

tion.

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` instead of using a dictionary of keyword-value pairs and a list, so it requires 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[4], it 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 than the default way. Note that I can use any prefix, so I could even unite the bindings from several modules under a common

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

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 ; note the added indentation!
    . #:renamer : symbol-prefix-proc '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 methods 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, finding the origin of a binding requires either IDE support or checking 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. 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 myself.

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 somewhat 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 up 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.

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 *` «

Guile-using development teams actually 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.

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 actually 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 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 likely 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 sim-

ilarly powerful and consistent interface as the one which numpy provides to access numerical datasets in Python. 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 actually shines, though (at least after taking care of the parens). Some of its names seem unintuitive to me, but its very consistent and easy structure make it even more accessible than Python - at least for people who do not come with a huge can of preconceptions from C-like languages. This is especially true when using the curly-infix extension (SRFI-105) which allows avoiding prefix-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 other 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 still 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. 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`.

What I am missing is a guide which combines the most elegant structures in Guile Scheme into a canonical way to solve problems. It is possible, 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

And similar as with `list-ec` and `in-list`, implementation details creep into high-level 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 soon becomes second nature, when you use Scheme a lot (because they are used all the time in recursion), their meaning is completely opaque to newcomers.

Let's show that with an example from the book The Little Schemer[1]:

[1]This example uses equal? instead of eq? to make the distinction between car/cdr and first/rest or head/tail more visible. eq? compares

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

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
       lambda : a lat
           cond
              : null? lat
                quote ()
              : equal? (first lat) a
                rest lat
              else
                cons : first lat
                       rember a : rest lat
```

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

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.

- 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 second element of a list (`(car (car l))`), and `cddr` is the tail of the list beginning with the 3rd element (`(cdr (cdr l))`).[2]

And from my own experience with The Little Schemer, `car`, `cdr` and `cons` 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 `drop list i`, which returns all but the first `i` elements from the list.

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

[2]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 they 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.

```
use-modules : ice-9 match
define l '(1 2 3)
match l
    : car cadr caddr
      . car
; => 1
match l
    : car cadr caddr
      . cadr
; => 2
match l
    : car cdr ...
      . cdr
; => (2 3)
match l
    : car cadr cddr ...
      . cddr
; => (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 allow you to recognize it more easily when you see it):

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

This is not what I would call simple, but at least it is explicit - and it follows the common theme to be 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 and duplication

And this missing polish is visible in quite a few other areas, too. From my current impression Guile 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 two different 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` actually takes a character and not a string as second argument.

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 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. Actually 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\T1\textquoteright s it. Now this works:
length "123"
; => 3
```

And using GOOPS with keyword-arguments, 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 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 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, but the advantage that the skills you learn with one Scheme can be readily 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 actually use

all the advanced features within Guile to keep interoperability to other Scheme-implementations.

Also it is harder to create a lean and elegant language if the language has to be a good fit for multiple implementations with different constraints and target groups. Consequently 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 to OSX and Windows

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 2014-02-22) 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. davexunit is working on that front with guile-2d. 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.

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 easy testing - 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 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 primitiv 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 to be found.

For some of this, there is limited flexibility due to keeping compatibility with Scheme code written for other implementations. But most of these problems can be overcome by lots of polishing, language design and focussed documentation.

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 the next few chapters will quote from the work of Ludovic Courtes (Ludo), Clinton (Unknown Lamer), 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, 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 call optimization. This means, that you can use recursion to solve problems and have the solution actually 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} ; else
```

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:

```
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).

```
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" (The Adventures of a Pythonista in Schemeland).

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} ; else
```

7.2 Real Threads!

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

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

```
use-modules : ice-9 futures
let loop ; create a long loop. Guile is ridiculously fast on math, so
         ; we have to make it expensive to see something.
    : i : iota 1000
    when : not : null? i
      let : : l : cdr : iota 1000 ; no 0
        let ; worst case: futures in the inner loop.
          : a : future : apply string-append : map number->string l
            b : future : apply string-append : map number->string : map log l
            c : future : apply string-append : map number->string : map sqrt l
          apply string-append : map touch : list a b c ; touch gets the result
      loop : cdr i
```

This runs at 232% CPU load on my machine, which is pretty good for a tight inner loop. Note that with futures guile automatically uses a thread pool.

7.3 Programming the syntax and embedded domain specific languages

Due to the integral role syntax adaptions take in Scheme, making an optimal 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 using pattern-matching.

Ludovic Courtès describes the benefits of using an embedded domain specific language for packages in GNU Guix in the description for his Fosdem-Talk Growing a GNU with Guix (video):

> 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:

```
(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
            "1y2p50zadb36izzh2zw4dm5hvdiydqf3qa88l8kav20dcmfbc5yl"))))
    (build-system gnu-build-system)
    (home-page "https://gnu.org/software/which/")
    (synopsis "Find full path of shell commands")
    (description
      "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 and can be used directly from Guile Scheme. Compare the above package definition with this ebuild and the Manifest needed for it:

```
EAPI="4"

inherit flag-o-matic toolchain-funcs

DESCRIPTION="Prints out location of specified executables that are in your path
HOMEPAGE="http://www.xs4all.nl/~carlo17/which/"
SRC_URI="http://www.xs4all.nl/~carlo17/which/${P}.tar.gz"

LICENSE="GPL-3"
SLOT="0"
KEYWORDS="alpha amd64 arm arm64 hppa ia64 m68k ~mips ppc ppc64 s390 sh sparc x8
IUSE=""

src_prepare() {
        append-lfs-flags
}

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

```
...
DIST which-2.20.tar.gz 135372 SHA256 d417b6... WHIRLPOOL 35ca3d...
...
SIGNATURE
...
```

Notes: I (the author of the article) am a longterm Gentoo user.

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

7.4 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 XML as library - which as a 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 OSs.

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 C code and header files out.

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 module (language xml-xcb spec) and looks like this:

```
(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 modules, without 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.

> This article is about the front-end only: lexer and parser, and transforming a simple AST (actually it's 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 Guile very easy to implement new languages.
>
> ... Simple, but not so simple ...
>
> 'simple' is just a simple language, maybe too simple for a serious compiler writer. Because even a front-end will take you a lot of time and hack power. No mention the backend. Fortunately, Guile provides a nice way to let language fans focus on the grammar

> rather than optimization. Nevertheless, all the languages front-end 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.

7.5 Your own object oriented programming system

GOOPS in Guile allows changing the OOP system to your liking. For example Clinton aka Unknown Lamer did some quite nifty OOP experiments with Guile. As a first stop he suggests having a look at serialize.scm, 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 Pythons object-system, this allows going outside the usual realm with ease - with mixins instead of inheritance being just one of the simplest applications. And where an often-used definition should be cumbersome, you can simply utilize macros to make it convenient again.

7.6 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: Guiles delimited continuations via prompts allow implementing advanced control structures in an efficient way. By default Guile uses them for Exception handling 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 a simple concept: In guile-2d 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 every function by passing it as argument to the game-agenda (this is a generalization of what 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 big hassle to use with Python.

7.7 Summary

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

While elegant recursion support and real threads provide incremental improvements over Python, redefinitions of the syntax and of object orientated programming allow shaping the language into something completely different. To top this, 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 without ever leaving Guile Scheme, as the example of guile-2d shows, and have them intergrated 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 the 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 completely new system.

And you do all this in the language you are also using 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 IV

Conclusions

While Python is a good choice for new programmers, thanks to providing a complete set of functionality for any kind of task, encoded in a minimal set of highly polished basic concepts and a very readable syntax, the structures of Python are almost stretched to their limit and extensions like list comprehensions make it more and more complicated for newcomers to understand code from others. It is 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.*

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 touts, but that's part of the reason why I started looking into Guile Scheme. Scaling a program from a first draft to a big application actually looks easier, and while the parentheses look odd at first, it's extensions for infix-math and for indentation-based syntax make it an even better choice for pseuddocode than Python. It's standard library is much smaller than the batteries of Python, but that is partially made up for by offering a much easier way to call C-libraries.

There are some 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 other huge challenge is deploying a guile-based program to any platform which does not have a decent package manager.

But these shortcomings are more than compensated by its strengths. Let-Rekursion (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.

Last, but not least, direct access to real threads (and consequently also truly concurrent futures) provide 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 recurision 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 already showed that it can withstand the onslaught of time, which pushed so many other languages and systems into oblivion. GNU 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.

Part V

Appendix

Chapter 8

See also (links)

- wingolog
- chaos code with scheme
- Guile Basics

Chapter 9

Glossary

Terms used by Schemers.

- Procedure: Function.
- 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.