From Python to PLT Scheme

Philippe Meunier　　Daniel Silva

College of Computer and Information Science
Northeastern University
Boston, MA 02115
{meunier,dsilva}@ccs.neu.edu

ABSTRACT

This paper describes an experimental embedding of Python into DrScheme. The core of the system is a compiler, which translates Python programs into equivalent MzScheme programs, and a runtime system to model the Python environment. The generated MzScheme code may be evaluated or used by DrScheme tools, giving Python programmers access to the DrScheme development suite while writing in their favorite language, and giving DrScheme programmers access to Python. While the compiler still has limitations and poor performance, its development gives valuable insights into the kind of problems one faces when embedding a real-world language like Python in DrScheme.

1. INTRODUCTION

The Python programming language [13] is a descendant of the ABC programming language, which was a teaching language created by Guido van Rossum in the early 1980s. It includes a sizeable standard library and powerful primitive data types. It has three major interpreters: CPython [14], currently the most widely used interpreter, is implemented in the C language; another Python interpreter, Jython [11], is written in Java; Python has also been ported to .NET [9].

MzScheme [8] is an interpreter for the PLT Scheme programming language [7], which is a dialect of the Scheme language [10]. MzScheme compiles syntactically valid programs into an internal bytecode representation before evaluation. MrEd [6] is a graphical user interface toolkit that extends PLT Scheme and works uniformly across several platforms (Windows, Mac OS X, and the X Window System.) Originally meant for Scheme, DrScheme [5] is an integrated development environment (IDE) based on MzScheme—it is a MrEd application—with support for embedding third-party extensions. DrScheme provides developers with useful and modular development tools, such as syntax or flow analyzers. Because MzScheme's syntax system includes precise source information, any reference by a development tool to

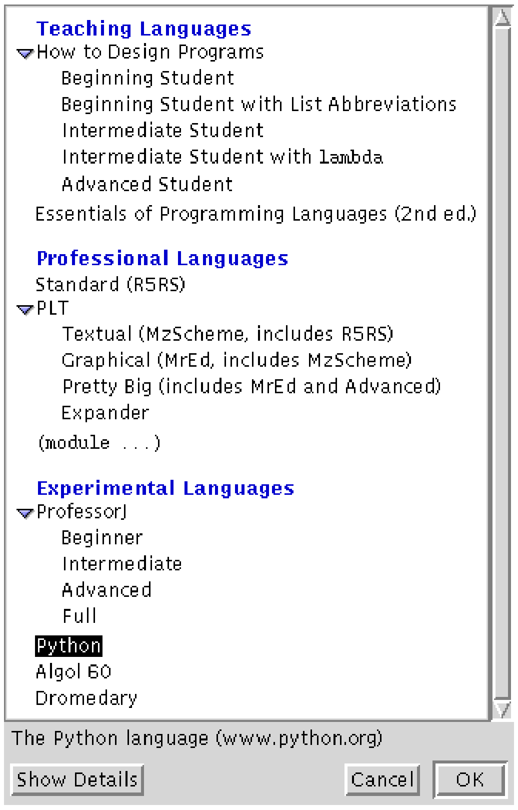
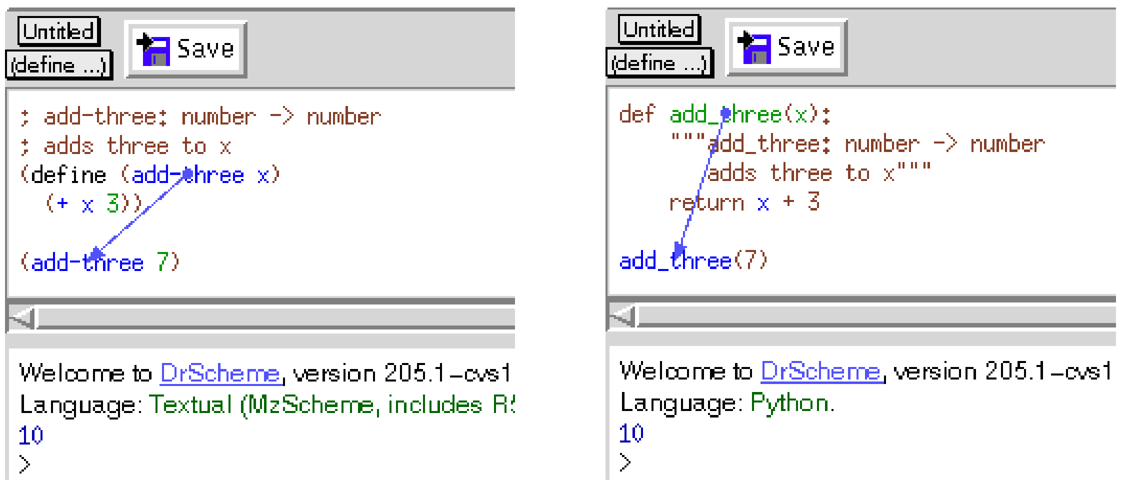

Teaching Languages
How to Design Programs
Beginning Student
Beginning Student with List Abbreviations
Intermediate Student
Intermediate Student with lambda
Advanced Student
Essentials of Programming Languages (2nd ed.)
Professional Languages
Standard (R5RS)
PLT
Textual (MzScheme, includes R5RS)
Graphical (MrEd, includes MzScheme)
Pretty Big (includes MrEd and Advanced)
Expander
(module ...)
Experimental Languages
ProfessorJ
Beginner
Intermediate
Advanced
Full
Python
Algol 60
Dromedary
The Python language (www.python.org)
Show Details
Cancel
OK

Figure 1: DrScheme language selection menu

such data can be mapped back to a reference to the original program text.

DrScheme is thus no longer just a development environment for Scheme. It can now potentially play the role of a program development environment for any language, which users can select from a menu (figure 1). When using any language from within the IDE, the program developer may use DrScheme's development tools, such as Syntax Check, which checks a program's syntax and highlights its bindings (figure 2), or MrFlow, which analyses a program's possible flow of values (MrFlow is still under development though). Also, any new tool added to the DrScheme IDE is supposed to work automatically with all the languages that DrScheme supports (figure 2).



```
; add-three: number -> number
; adds three to x
(define (add-three x)
  (+ x 3))

(add-three 7)
```

```
Welcome to DrScheme, version 205.1-cvs1
Language: Textual (MzScheme, includes R!
10
>
```

```
def add_three(x):
    """add_three: number -> number
    adds three to x"""
    return x + 3

add_three(7)
```

```
Welcome to DrScheme, version 205.1-cvs1
Language: Python.
10
>
```

Figure 2: Evaluation and Syntax Check for Scheme and Python

To support a new language, however, DrScheme needs a translator for programs written in that language. In the case of adding Python support to DrScheme, this is the task of the Python-to-Scheme compiler described in this paper. The compiler is packaged as a DrScheme language tool, thus introducing Python as a language in DrScheme's list of choices (figure 1).

The compiler was created as an experiment in porting a language like Python to DrScheme. With Python available as a DrScheme language, Python programmers can use the DrScheme IDE and its accompanying tools to develop Python programs. It also gives Scheme programmers access to the large amount of Python code available on the Internet. The compiler still suffers from several limitations though, primarily relating to the runtime support. The performance of the generated code is also currently poor compared to CPython. While we expect some of the limitations to disappear in the future and the performance to get better as the generated code is optimized, we already consider the experiment to be successful for the insights we have gained about the problem of embedding a real-world language into DrScheme.

Section 2 of this paper presents the overall architecture of the compiler system, including details about code generation and the runtime system. Section 3 describes the current status of the compiler, gives an idea of the current performance of the generated MzScheme code, and evaluates the successfulness of the whole experiment. Section 4 relates other works to this paper. Section 5 lists some of the major parts that still need to be worked on, and we conclude in section 6.

2. ARCHITECTURE

This section describes the architecture of the Python-to-Scheme compiler. The compiler has a conventional structure with three major components: the front-end, which uses a lexical analyzer to read program text and a parser to check the syntax of the tokens produced by the scanner; the back-end, which is a code generator using the parser's output to create MzScheme code; and the runtime system, which provides low-level functions that the generated code makes use of. This section delineates these three components. Section 2.1 describes the scanner and parser; section 2.2, the code generator; and section 2.3, the runtime system.

Note that, even though CPython is based on a virtual machine, we did not consider compiling CPython byte code instead of compiling Python source code. While compiling CPython byte code to Scheme is certainly doable, the semantic mismatch between the stack-based byte code and Scheme is big enough that DrScheme's tools would most likely give poor results on byte code (in addition to the problem of mapping those results for the byte code back into results for Python source code, since, unlike DrScheme, CPython does not preserve in the byte code much information about the source code to byte code transformation).

2.1 Lexical and Syntax Analysis

Python program text is read by the lexical analyzer and transformed into tokens, including special tokens representing indentation changes in the Python source code. From this stream of tokens the parser generates abstract syntax trees (ASTs) in the form of MzScheme objects, with one class for each Python syntactic category. The indentation tokens are used by the parser to determine the extent of code blocks. The list of generated ASTs is then passed on to the code generator.

2.2 Code Generation

The code generator produces Scheme code from a list of ASTs by doing a simple tree traversal and emitting equivalent MzScheme code. The following subsections explain the generation of the MzScheme code for the most important parts of the Python language. They also describe some of the problems we encountered.

2.2.1 Function Definitions

Python functions have a few features not present in Scheme functions. Tuple variables are automatically unpacked, arguments may be specified by keyword instead of position, and those arguments left over (for which no key matches) are placed in a special dictionary argument. These features are implemented using a combination of compile-time rewriting (e.g. for arguments specified by keywords) and runtime processing (e.g. conversion of leftover arguments into a Python tuple). Default arguments are not yet implemented. Python's `return` statement is emulated using MzScheme escape continuations. For example the following small Python function:

```
def f(x, y, z, *rest, **dict):
    print dict
```

is transformed into the following Scheme definition:

```
(namespace-set-variable-value! 'f
  (procedure->py-function%
   (opt-lambda (dict x y z . rest)
     (let ([rest (list->py-tuple% rest)])
       (call-with-escape-continuation
        (lambda (return10846)
          (py-print #f (list dict))
          py-none))))
   'f (list 'x 'y 'z) null 'rest 'dict))
```

2.2.2 Function Applications

Functions are applied through `py-call`. A function object is passed as the first argument to `py-call`, followed by a list of supplied positional arguments (in the order they were supplied), and a list of supplied keyword arguments (also in order). So, for example, the function call `add_one(2)` becomes:

```
(py-call add_one
         (list (number->py-number% 2))
         null)
```

The `py-call` function extracts from the `add_one` function object a Scheme procedure that simulates the behavior of the Python function when it is applied to its simulated Python arguments by `py-call`.

2.2.3 Class Definitions

In Python classes are also objects. A given class has a unique object representing it and all instances of that class use a reference to that unique class object to describe the class they belong to. The class of class objects (i.e. the type of an object representing a type) is the `type` special object / class. The type of `type` is `type` itself (i.e. `type` is an object whose class is represented by the object itself). With this in mind consider this small Python class, which inherits from two classes `A` and `B` that are not shown here:

```
class C(A, B):
    some_static_field = 7
    another_static_field = 3

    def m(this, x):
        return C.some_static_field + x
```

In this class `C`, three members are defined: the two static fields and the method `m`, which adds the value of the first static field to its argument. This class is converted by the code generator into a staticmethod call to the `__call__` method of the `type` class (which is also a callable object). This call returns a new class object which is then assigned to the variable `C`:

```
(namespace-set-variable-value! 'C
  (python-method-call type '__call__
   (list
    (symbol->py-string% 'C)
    (list->py-tuple% (list A B))
    (list
     (lambda (this-class)
       (list 'some_static_field
             (number->py-number% 7)))
     (lambda (this-class)
       (list
        'another_static_field
        (let-values
            ([(some_static_field)
              (values (python-get-member
                       this-class
                       'some_static_field #f))])
          (number->py-number% 3))))
     (lambda (this-class)
       (list
        'm
        (procedure->py-function%
         (opt-lambda (this x)
           (call/ec
            (lambda (return)
              (return
               (python-method-call
                (python-get-attribute
                 C 'some_static_field)
                '__add__
                (list x)))
              py-none)))
         'm (list 'this 'x) null #f #f)))))))
```

All instances of the class `C` then refer to that new class object to represent the class they belong to.

At class creation time member fields (but not methods) have access to the previously created fields and methods. So for example `some_static_field` must be bound to its value when evaluating the expression used to initialize the field `another_static_field` in the class `C` above. To emulate this the generated Scheme code that initializes a field must always be a function that receives as value for its `this-class` argument the class object currently being created, to allow for the extraction of already created fields and methods from that class object if necessary.

Note that a class's type is different from a class's parent classes. The parents of a class (the objects `A` and `B` representing the parent classes of `C` in the example above) can be accessed through the `__bases__` field of a class. The `__class__` field of an "ordinary" object refers to the object representing that object's class while the `__class__` field of an object representing a class refers to the `type` object (figure 3). This second case includes the `__class__` field of the top `object` class, even though `type` is a subclass of `object`. The class of an object can be changed at runtime by simply assigning a new value to the `__class__` field of that object.

The Python object system also allows fields and methods to be added to an object at runtime. Since classes are themselves objects, fields and methods can be added at runtime

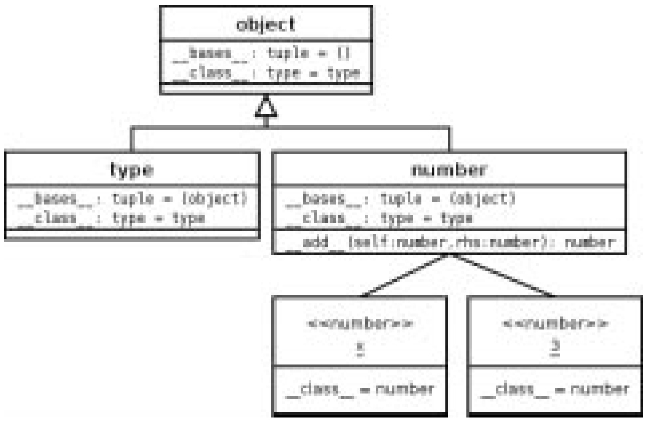


Figure 3: A simple Python class

to a class, which is then reflected in all the existing instances of that class.

Since the MzScheme object system segregates classes and objects, does not allow either to be modified at runtime, and does not support multiple inheritance, the Python object system could not be mapped to the MzScheme one. All Python object are therefore emulated using MzScheme hash tables (which is also what they are internally in CPython).

2.2.4 Variable Assignments

Identifiers are normally bound either at the top level or inside functions. Identifiers from imported modules are bound differently (see section 2.2.5).

Assignments at the top level are translated into `define`s for first assignments or `set!`s for mutative assignments. In the following Python listing, the first line defines `x`, while the second line mutates `x` and defines `y` as the same value `2` (which is only evaluated once).

```
x = 1
x = y = 2
```

Identifiers defined inside functions are bound using `let`. For example, consider the following function that uses a single variable, `x`, defined on the fly.

```
def f():
    x = 1
```

Its body is translated into this Scheme equivalent (omitting the escape continuation code used to handle possible `return` statements):

```
(namespace-set-variable-value! ’f
  (opt-lambda ()
    (let ([x (void)])
      (let ([rhs1718 (number->py-number% 1)])
        (set! x rhs1718))
      py-none)))
```

As a current shortcoming of the compiler, all variables defined throughout the body of a Python function are defined at once in a single `let` at the start of the corresponding Scheme function. To ensure that using a variable before it is defined still results in a runtime error the `let`-bound variables have to be given the value `void`. While this works fine in practice, it does not provide for good error messages though. This will be fixed in the future (see section 5).

When a `global` statement names any variable, the named variable is simply omitted from the Scheme function's initial `let` bindings, thereby allowing assignments to said variable to mutate an identifier existing in an outer scope (if it exists, otherwise a runtime error occurs).

2.2.5 Importing Modules

Unlike MzScheme modules, Python modules allow assignments to identifiers defined in other modules. Python also allows cycles between modules. It was therefore not possible to map Python modules to MzScheme modules. Rather Python modules are emulated using MzScheme namespaces.

In order to import a Python module at runtime—and, in fact, to initialize the environment at startup—the runtime system creates a new MzScheme namespace and populates it with the built-in Python library. The runtime system then compiles the requested module and evaluates it in this new namespace. Finally, new bindings for the necessary values are copied from that namespace into the original namespace of the module importer. For example, when evaluating the statement `import popen from os`, only the binding for `popen` is copied into the original namespace from the new one created to compile the `os` module. A module is always only compiled once, even if it is imported multiple times.

Since `import m` only copies over a reference to module `m` and its namespace, references to values in module `m`, such as `m.x`, are shared between modules importing `m`. However, a statement of the form `from m import x` copies the value of `x` into the current module namespace. There is no sharing of `x` between modules then.

2.3 The Runtime System

The Python runtime system can be divided into two parts: modules that are written in Python and modules that are written in C. The code generation described above can be applied to both user code and the parts of the Python runtime that are written in Python. This means that Python programmers can use these runtime modules as they would normally do. This also means that Scheme programmers have access to the parts of the Python runtime written in Python by simply invoking the compiler on them and evaluating the resulting MzScheme code (although there is currently no simple API provided to do that).

The C-level modules of the Python runtime can be dealt with in several ways. Some of these modules use C macros to abstract the runtime code over the actual internal representation of Python objects. These modules can therefore in principle be directly reused by modifying the appropriate C macros to work on MzScheme values instead of Python objects. The use of C macros is not systematic throughout the Python runtime code though, so some changes to the code are required to make it completely abstract and there does not seem to be any simple automated way to do this. As an experiment the Python String class code and macros were modified in this manner and the class is now usable by the DrScheme Python programmer.

For the Python modules written in C that are poorly, or not at all, abstracted over the representation of Python object, the most elegant solution would be to convince the CPython developers to rewrite these core modules in a more abstract way using C macros, thereby allowing the two systems to share that runtime code. We do not expect this to happen in the foreseeable future though, so one alternative solution is to replace these C modules with equivalent MzScheme code. Calls to Python runtime functions can be transformed by the code generator into calls to MzScheme functions when the Python functions have direct MzScheme equivalents (e.g. `printf`). Python functions that do not have any direct MzScheme equivalent must be rewritten from scratch, though this brings up the problem of maintaining consistency with the CPython runtime as it changes. We are currently examining the Python C code to determine how much of it can be reused and how much of it has to be replaced. Another possible solution is to use an automated tool like SWIG [3] to transform the Python C modules into MzScheme extensions. The code generator can then replace calls to the original C modules by MzScheme function calls to the SWIG-generated interface. This approach is also under investigation.

Note that there is currently no way for the Python programmer using DrScheme to access the underlying MzScheme runtime. Giving such access is easy to do through the use of a Python module naming convention that can be treated as a special case by the code generator (e.g. `import mzscheme` or `import ... from mzscheme`).

3. STATUS AND EVALUATION

Most of the Python language has been implemented, with the exception of the `yield` and `exec` statements, and of default function parameters (as explained in section 2.2.1). The Python `eval` function has not been implemented yet either but since `import` is implemented and since it evaluates entire Python files, the necessary machinery to implement both `exec` and `eval` is already available. There is no plan to support Unicode strings, at least as long as MzScheme itself does not support them. There is also currently no support for documentation strings. As described in section 2.3 access to the parts of the Python runtime system written in C is still a problem.

Because Python features like modules or objects have very dynamic behaviors and therefore must be emulated using MzScheme namespaces and hash tables (respectively), the code generated by our system is in general significantly bigger than the original Python code. See for example the simple Python class from section 2.2.3 that expands into about 30 lines of MzScheme code. In general a growth factor of about three in the number of lines of code can be expected. The generated code also involves a large number of calls to internal runtime functions to do anything from continually converting MzScheme values into Python values and back (or more precisely into the internal representation of Python values our system is using and back) to simulating a call to the `__call__` method of the `type` class object. Finally, each module variable, class field or method access potentially involves multiple namespace or hashtable lookups done at the Scheme level. As a result the performance of the resulting code is poor compared to the performance of the original Python code running on CPython. While no systematic performance measurement has been made yet, anecdotal evidence on a few test programs shows a slowdown by around three orders of magnitude.

Using DrScheme tools on Python programs has given mixed results. Syntax Check, which checks a program's syntax and highlights its bindings using arrows, has been successfully used on Python programs without requiring any change to the tool's code (figure 2). Some features of the Python language make Syntax Check slightly less useful for Python programs than for Scheme programs though. For example, since an object can change class at runtime, it is not possible to relate a given method call to a specific method definition using just a simple syntactic analysis of the program. This is a limitation inherent to the Python language though, not to Syntax Check.

A tool like MrFlow, which statically analyzes a program to predict its possible runtime flow of values, could potentially be able to relate a given method call to a given method definition. While MrFlow can already be used on Python programs without any change, it does not currently compute any meaningful information: MrFlow does not know yet how to analyze several of the MzScheme features used in the generated code (e.g. namespaces). Even once this problem is solved, MrFlow will probably still compute poor results. Since all Python classes and object are emulated using MzScheme hash tables, and since value flow analyses are unable to differentiate between runtime hash table keys, MrFlow will compute extremely conservative results for all the object oriented aspects of a Python program. In general there is probably no easy way to statically and efficiently analyze the generated code. In fact there is probably no way to do good value-flow analysis of Python programs at all given Python's extremely dynamic notion of objects and modules.

Another DrScheme tool, the Stepper, does not currently work with Python programs. The Stepper allows a programmer to run a program interactively step by step. To work with Python the Stepper would need to have access to a decompiler, a program capable of transforming a generated but reduced MzScheme program back into an equivalent Python program. Creating such an decompiler is a non-trivial task given the complexity of the code generated by the compiler.

Due to the difficulties encountered with the Python runtime and due to the current poor performance of the code generated, the compiler should be considered to be still at an experimental stage. The fact that most of the Python language has been implemented and that a DrScheme tool like Syntax Check can be used on Python programs without any change is encouraging though. The experience that has been gained in porting a real-world language to DrScheme is also valuable. We therefore consider the experiment to be successful, even if a lot of work still remains to be done.

4. RELATED WORK

Over the past years there have been several discussions [1, 2] on Guile related mailing lists about creating a Python to Guile translator. A web site [4] for such a project even exists, but does not contain any software. Richard Stallman indicated [12] that a person has been working on "finishing up a translator from Python to Scheme" but no other information on that project could be found.

Jython, built on top of the Java Virtual Machine, is another implementation of the Python language. The implementation is mature and gives users access to the huge Java runtime. All the Python modules implemented in C in CPython were simply re-implemented in Java. Maintaining the CPython and Jython runtimes synchronous requires constant work though.

Python for .NET is an exploratory implementation of the Python language for the .NET framework and has therefore severe limitations (e.g. no multiple inheritance). Like Jython it gives access to the underlying runtime system and libraries. Only a handful of modules from the Python runtime have been implemented. Among those, the ones written in Python became accessible to the user after being modified to fit within the more limited Python language implemented by the interpreter. A few modules originally written in C in CPython were re-implemented using the C# language.

Compilers for other languages beside Python are being developed for DrScheme. Matthew Flatt developed an implementation of the Algol60 language as a proof of concept. David Goldberg is currently working on a compiler for the OCaml language called Dromedary, and Kathy Gray is working on a DrScheme embedding of Java called ProfessorJ.

5. FUTURE WORK

In its present state the biggest limitation of the compiler is the lack of access to the C-level Python runtime. As such we are currently focusing most of our development efforts in that area, investigating several strategies to overcome this problem (see section 2.3).

While the performance of the generated code is poor, no attempt has yet been made at profiling it. The performance will be better once the code generator has been modified to create more optimized code, although it is unclear to us at this stage how much improvement can be expected in this regard. The need to simulate some of the main Python features (e.g. the object and module systems) and the large number of runtime function calls and lookups involved means than the generated code will probably never have a performance level on par with the CPython system although an acceptable level should be within reach.

As described in section 3, a few parts of the Python language remain to be implemented. We do not anticipate any problem with these. There is also a general need for better error messages and a more complete test suite.

6. CONCLUSION

A new implementation of the Python language is now available, based on the MzScheme interpreter and the DrScheme IDE. While most of the core language has been implemented a lot of work remains to be done on the implementation of the Python runtime and on improving the performance. Despite this Python developers can already benefit from some of DrScheme's development tools to write Python code, and Scheme programmers start now to have access to the large number of existing Python libraries.

7. ACKNOWLEDGMENT

Many thanks to Scott Owens for developing the DrScheme parser tools and creating the first version of the MzScheme lexer and parser grammar for the Python language.

8. REFERENCES

[1] Guile archive. `http://www.cygwin.com/ml/guile/2000-01/threads.html#00382`, January 2000.

[2] Guile archive. `http://sources.redhat.com/ml/guile/2000-07/threads.html#00072`, July 2000.

[3] David M. Beazley. SWIG Users Manual. `www.swig.org`, June 1997.

[4] Thomas Bushnell. `http://savannah.gnu.org/projects/gpc/`.

[5] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A Pedagogic Programming Environment for Scheme. Programming Languages: Implementations, Logics, and Programs, 1292:369–388, September 1997.

[6] Matthew Flatt. PLT MrEd: Graphical Toolbox Manual PLT, December 2002.

[7] Matthew Flatt. PLT MzScheme: Language Manual PLT, December 2002.

[8] Matthew Flatt. The MzScheme interpreter. `http://www.plt-scheme.org/`, December 2002. Version 203.

[9] Mark Hammond. Python for .NET: Lessons learned. ActiveState Tool Corporation, November 2000.

[10] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 Report on the Algorithmic Language Scheme. ACM SIGPLAN Notices, 33(9):26–76, 1998.

[11] Samuele Pedroni and Noel Rappin. Jython Essentials O'Reilly, March 2002.

[12] Richard Stallman. Invited Talk: My Lisp Experiences and the Development of GNU Emacs. In Proc. International Lisp Conference, October 2002.

[13] Guido van Rossum. Python R eference Manual PythonLabs, February 2003.

[14] Guido van Rossum. The Python interpreter. PythonLabs, February 2003. Version 2.3a2.