Psyche Manual

Yigal Duppen

November 5, 2002

Contents

1 Introduction **3**

1.1 What is Psyche? 3

1.2 Why is Psyche not Scheme? 3

1.3 About This Document 4

I Developer's Guide **5**

2 Installing Psyche **6**

3 Running the Interpreter **7**

4 Embedding the Interpreter **8**

4.1 Evaluating Scheme Expressions 8

4.2 Dynamically Constructing Scheme Expressions 9

5 Scheme Types in Psyche **10**

5.1 Symbols 10

5.2 Strings 11

5.3 Pairs and Lists 12

6 Scheme Procedures in Psyche **13**

6.1 Name Mangling 13

6.2 Calling Scheme Procedures from Python 14

7 Extending Scheme with Python Functions **15**
7.1 General Process . . . 15
7.2 Example: Adding a Dictionary to Scheme . . . 15
7.2.1 Defining the Scheme Procedures . . . 16
7.2.2 Implementing the Python functions . . . 17
7.2.3 Using the New Procedures . . . 18

II The Psyche API **20**

8 `psyche.interpreter` **21**
8.1 Environment Objects . . . 22
8.2 Interpreter Objects . . . 22
8.3 Shell Objects . . . 23

9 `psyche.types` **24**

III Appendix **25**

A R5RS Compliance **26**
A.1 Extra Features . . . 28

B GNU Free Documentation License **29**
B.1 Applicability and Definitions . . . 30
B.2 Verbatim Copying . . . 31
B.3 Copying in Quantity . . . 31
B.4 Modifications . . . 32
B.5 Combining Documents . . . 34
B.6 Collections of Documents . . . 34
B.7 Aggregation With Independent Works . . . 34
B.8 Translation . . . 35
B.9 Termination . . . 35
B.10 Future Revisions of This License . . . 35

Chapter 1

Introduction

1.1 What is Psyche?

Psyche is a Scheme interpreter written in Python. It allows you to embed a Scheme interpreter in any Python program; it's very easy to extend the interpreter with new commands, written in Python.

I decided to build Psyche when I was reading *Structure and Interpretation of Computer Programs*[2]; this book uses Scheme as its example language and it looked like a nice language to toy around with. Soon, this got out of hand, resulting in this product.

The name is a combination of the first two letters of Python and the first four letters of Scheme. Originally, this program was called `pyscheme`, until I discovered this program already exists[4]. However, since `pyscheme` is not fully Scheme compliant either, I felt confident enough to continue with Psyche.

1.2 Why is Psyche not Scheme?

In its current stage, Psyche is not yet a real Scheme interpreter since it does not fully implement the Scheme standard R5RS[1]. Full R5RS compliance is currently the main goal of Psyche.

The most interesting features that are missing are:

- Correct numeric support
- Continuations
- Hygienic Macros

Appendix A contains a complete overview of the R5RS compliance.

1.3 About This Document

This document consists of two parts. The first part is a cross between a tutorial and a developer's guide. It focuses on the different aspects of Psyche. After reading this part, you should be able to use Psyche to its full potential.

The second part is an API Reference of the Psyche modules. It covers the `psyche.interpreter`, `psyche.types` and `psyche.schemefct` modules.

Part I

Developer's Guide

Chapter 2

Installing Psyche

Psyche has only one requirement: Python, version 2.2 or later. Python can be obtained from `http://www.python.org`.

Psyche uses the Python `distutils` package for its distribution. Read the `INSTALL` file in your Psyche distribution for more information.

Psyche uses Plex[3] for its lexical analyzer; it uses a modified version of Spark[5] for its parser and semantic analyzers. The Psyche distribution includes both Plex and the modified version of Spark.

Chapter 3

Running the Interpreter

Psyche contains a simple interactive interpreter. The interpreter can be started with the command `/usr/local/bin/psyche`, assuming that you installed Psyche in `/usr/local`.

The interpreter uses the console for its I/O; no fancy windows or anything. When possible, it uses the GNU `readline` interface for its input.

Example 3.1 A session with the interpreter

```
$ /usr/local/bin/psyche
Psyche version X.Y, Copyright (C) 2002 Y. Duppen

Psyche comes with ABSOLUTELY NO WARRANTY.  This is free software, and
you are welcome to redistribute it under certain conditions; read the
attached COPYING for details.

psyche> (define x 5)

psyche> (define (square n)
......>         (* n n))

psyche> (square x)
25

psyche> (square 4)
16

psyche> (square "hello")
Error: unsupported operand type(s) for *: 'str' and 'str'

psyche> (quit)

$
```

Chapter 4

Embedding the Interpreter

While running the interpreter is nice, it is not a good example of what Psyche can do. There are more interpreters, and most of them are faster, easier to use, or better in other respects. Psyche's strength lies in embedding the interpreter in a Python program.

Psyche consists of a small number of Python modules. These modules and a short description of them are described in Table 4.1.

4.1 Evaluating Scheme Expressions

Let's start with a small example from the interactive Python shell. In this example we shall create a Scheme interpreter and use it to calculate the square of 5.

Example 4.1 Computing squares

Module	Purpose
`psyche`	Top level module for Psyche
`psyche.analyzers`	Semantical Analyzers
`psyche.ast`	Nodes for Abstract Syntax Tree
`psyche.function`	Classes for executing Scheme functions
`psyche.interpreter`	The interpreter, the shell and environments
`psyche.lexer`	Lexical Analyzer and Token class
`psyche.parser`	Parser
`psyche.schemefct`	Implementation of Scheme procedures
`psyche.types`	Implementation of Scheme types

Table 4.1: Psyche Modules

```
Python 2.2.1 (#1, Apr 21 2002, 08:38:44)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from psyche.interpreter import Interpreter
>>> i = Interpreter()
>>> i.eval('(define (square x) (* x x))')
>>> i.eval('(square 5)')
25
>>> i.eval('(square -1)')
1
>>> i.eval('(square "hello")')
Traceback (most recent call last):
    ...
TypeError: unsupported operand type(s) for *: 'str' and 'str'
>>>
```

Let's step through this example and see what happens.

The first statement imports the `Interpreter` into our namespace. The second statement creates an actual `Interpreter` object and names it `i`.

The other statements use the `eval` method to execute Scheme commands. `eval` is the most interesting method in the `Interpreter`; it expects a string containing a legal Scheme command, evaluates the command and returns the result of the evaluation.

The third statement evaluates the Scheme command `(define...)`. The fourth and fifth statements again evaluate a Scheme expression, using the `square` procedure defined in the third statement.

At this point, two interesting observations can be made. Firstly, notice how the interpreter keeps its state; in the fourth and fifth statements, it remembers the definition of `square` from the third statement.

Secondly, the latter two evaluations print a (correct) return value, while the first evaluation does not. Understanding this requires some knowledge the Scheme specification, which explicitly states the return values of each Scheme command. However, for some commands such as `define`, the R5RS specifies that the return value is undefined. In these cases, Psyche will return `None`, which is silently ignored by the Python shell.

The last statement tries to evaluate a Scheme expression resulting in an error. Psyche uses the Python exception handling mechanism to signal errors. In this case, a `TypeError` is raised.

4.2 Dynamically Constructing Scheme Expressions

Warning: Unfinished

Chapter 5

Scheme Types in Psyche

Warning: Unfinished

Scheme has a number of well-defined types; unfortunately, those types do not map exactly on the Python types. In order to return meaningful results from the `eval` method, Psyche wraps some of those types in Python objects. These objects can be found in the `psyche.types` module.

Table 5.1 shows how Scheme types are mapped on Python types and Psyche objects.

5.1 Symbols

Scheme supports the notion of symbols, something that does not exist in Python. Therefore, Psyche provides the class `Symbol` that models Scheme symbols.

There are two kinds of symbols in Scheme:

1. Symbol literals

Scheme type	Python type	Psyche object
`Number`	`int`, `long`, `float`, `complex`	`Fraction`
`Boolean`		`Boolean`
`Pair`		`Pair`
`Symbol`		`Symbol`
`Character`		`Character`
`String`	`string`	`MString`
`Vector`		`Vector`

Table 5.1: Scheme types, Python types and Psyche objects

2. Symbols created by `string->symbol`

Every symbol also has a string representation, which can be obtained by `symbol->string`. Two symbols are identical if and only if they have an identical string representation.

The string representation differs between literal symbols and symbols created by `string->symbol`: the former are always represented in lower case[1], while the latter are represented exactly like the string they were derived from.

The Psyche class `Symbol` implements this by providing a constructor with two arguments: `name` and the optional `fromString`. Two symbols are equal iff

1. both `Symbol`s have the same lowercased name
2. neither `Symbol` has a `fromString` or both `Symbol`s have the same `fromString`

Since symbols occur frequently in Scheme programs, literal `Symbol`s are interned using the Flyweight pattern. The Flyweight pool uses weak references, assuring that `Symbol`s are garbage collected as soon as there are no other references left.

The result of this is that the `eqv?` on literals is identical to using the Python `is` comparison keyword.

5.2 Strings

In Scheme, there are three kinds of strings:

1. Strings literals
2. Strings returned by `symbol->string`
3. Strings returned by other Scheme functions

String literals are immutable and are represented by the Python type `str`.

Strings returned `symbol->string` are immutable as well; however, they have the added functionality that calling `string->symbol` on such a string returns an interned symbol (see 5.1 for more information on such symbols). Strings returned by `symbol->string` are represented by the Psyche class `SymbolString`, a subclass of `str`.

`SymbolString`s are identical to `str` in all respects, apart from an extra field used by the `string->symbol` function.

All other strings in Scheme are mutable. They are represented by the Psyche class `MString`, also a subclass of `str`. `MString`s behave exactly like `str`, with the added functionality of supporting `__setitem__`.

`MString.__setitem__` only accepts single character strings. All other values raise a `TypeError`.

[1]This is more specific than R5RS, which only requires a standard case

5.3 Pairs and Lists

Scheme has pairs as a built-in datatype. Pairs are used to implement lists, by setting the cdr of a pair to another pair or the empty list.

Chapter 6

Scheme Procedures in Psyche

All procedures for Scheme described in R5RS[1] are implemented in Python in the module `psyche.schemefct`. Since Scheme identifiers are a superset of Python identifiers, some name mangling was necessary. Since the complete list of Scheme procedures is already covered in R5RS, this Chapter will focus on the name mangling.

6.1 Name Mangling

This Section describes the algorithm used for name mangling. In this discussion, "Scheme procedure" refers to the procedure from R5RS while "Python function" refers to the implementation of a Scheme procedure in `psyche.schemefct`.

Table 6.1 contains some examples.

case Scheme procedures are all in lower case, with separate words divided by dashes. Python functions use mixed case, where the first word is lowercase and the first character of each following word is capitalized.

operators In Scheme, all operators are actually identifiers or parts of identifiers. The Python functions use the same naming convention as the module `operators`.

`?` Scheme procedures implementing predicates all end with a question mark. Python functions use the standard Python naming scheme by prepending the name with `is` (and using appropriate capitalization).

`!` Scheme procedures use the exclamation mark to specify state-changing operations. Python functions omit the exclamation mark.

Scheme procedure	Python function
`current-input-port`	`currentInputPort`
`+`	`add`
`number?`	`isNumber`
`string-set!`	`stringSet`
`string->number`	`stringToNumber`
`char-ci<=?`	`isCharCiLe`

Table 6.1: Some name mangling examples

`->` Scheme procedures use `->` for functions that convert one type to another. Python uses the word `to` (with appropriate capitalization).

reserved names Sometimes, after applying the previous mangling steps, the resulting name is not appropriate in Python. It can be a keyword or the name of a builtin or the name of a well-known module. These words are suffixed with an underscore. So the Scheme procedures `not`, `list` and `string` are implemented by the Python functions `not_`, `list_` and `string_`

6.2 Calling Scheme Procedures from Python

If there is need to call one of the Scheme procedures in Python, there are two possibilities: the first option is to do the name mangling yourself. This is not too difficult, but quite error-prone.

The alternative is the `procedures` variable in `psyche.schemefct`. This is a dictionary mapping all Scheme procedure names to the corresponding Python function.

While it is possible to modify the `procedures` dictionary, you are advised not to. This variable is used to initialize instances of `SchemeEnvironment5`, the initial environment for most Interpreters.

Example 6.1 U *sing `number?` in Python*

```
from psyche import schemefct, interpreter

i = interpreter.Interpreter()
obj = i.eval("5")

result = schemefct.isNumber(obj)
# or
result = schemefct.procedures["number?"](obj)
```

Chapter 7

Extending Scheme with Python Functions

It is easy to extend Psyche with new Scheme procedures, written in Python. In this Chapter, Psyche will be extended with a new `dict` object that uses Python dictionaries for fast lookups.

7.1 General Process

Adding new features to Psyche is generally done as follows:

1. Define the Scheme procedures that will be added
2. Implement these procedures, using the Psyche types where necessary as described in Chapters 5 and 9.
3. Create a new `Environment`, derived from `SchemeEnvironment5` as described in Chapter 8. Add the new procedures to this environment.
4. Instantiate a new `Interpreter` that uses this environment.

7.2 Example: Adding a Dictionary to Scheme

In Scheme, dictionaries or tables are usually implemented using association lists. While this is a nice and general algorithm, in some cases real hash tables might actually be a better choice.

In this Section we shall implement a dictionary object that works on Numbers, Characters and Symbols[1].

7.2.1 Defining the Scheme Procedures

The first step is to define the Scheme procedures. Using the Scheme naming scheme, we come to the following set of operations:

(`make-dict`) Creates a new dictionary object.

(`dict-ref` *dict key*) *key* must be a key in *dict*. *key* must be a number of a symbol. `dict-ref` returns the value associated with *key*.

(`dict-set!` *dict key value*) Associates *key* with *value* in *dict*. If *key* was already associated with a value, the old association is removed.

(`dict-key?` *dict key*) Returns `#t` if *key* is a key in *dict*. Returns `#f` if *key* is not a key.

(`dict->list` *dict*) Returns a newly allocated association list with the same bindings as *dict*. The order of the associations in the list is unspecified.

These procedures are probably not sufficient, but they give a nice overview of the possibilities.

Example 7.1 Using the dictionary

```
(define d (make-dict))              ==> unspecified
(dict-key? d 4)                     ==> #f
(dict-ref  d 4)                     ==> error
(dict-set! d 4 (list 'a 'b))        ==> unspecified
(dict-set! d "x" "y")               ==> error

(dict-key? d 4)                     ==> #t
(dict-ref d 4)                      ==> (a b)
(set-car! (dict-ref d 4) 'b)        ==> unspecified

(dict-set! d #\H "hello")           ==> unspecified

(dict->list d)                      ==> ((4 b b)) (#\H . "hello"))
```

[1] Hash tables use hash functions for storing and accessing their associations; since hash functions for mutable objects are tricky, we only allow immutable objects as keys

7.2.2 Implementing the Python functions

We can now continue by implementing the Python functions. We start out by creating a new module and importing `psyche.schemefct`. The functions defined in `schemefct` will be useful later on. For the sake of an argument, let's assume the new module is called `psychedict`.

The first functions, the equivalents of `make-dict`, `dict-ref` and `dict-key?` are pretty straightforward.

Example 7.2 `make-dict`, `dict-ref` and `dict-key?`

```
def makeDict():
    return {}

def dictRef(d, key):
    if not isinstance(d, dict):
        schemefct.error("Not a dictionary", d)
    return d[key]

def isDictKey(d, key):
    if not isinstance(d, dict):
        schemefct.error("Not a dictionary", d)
    return schemefct.schemeBool(d.has_key(key))
```

Some remarks are in order. First of all, notice how we use the Psyche function `error` to raise explicit errors; on the other hand, for the `dict-ref` procedure we rely on Python's behavior of raising a `KeyError` when a key is not present.

Furthermore, the names of the Python procedures are created from the Scheme names by using the name mangling scheme from Chapter 6.

Finally, notice how we have to convert Python boolean values to Scheme boolean values using the `schemeBool` function. This is very important, since Scheme booleans have different semantics from Python booleans.

The `dict-set!` procedure is a bit more interesting. It will use the `isNumber`, `isChar` and `isSymbol` functions from `schemefct` to check the key.

Example 7.3 `dict-set!`

```
def dictSet(d, key, value):
    if not isinstance(d, dict):
        schemefct.error("Not a dictionary", d)
    if not (schemefct.isNumber(key)
            or schemefct.isChar(key)
            or schemefct.isSymbol(key)):
        schemefct.error("Invalid key", key)
    d[key] = value
```

Notice how this function has no return value; this is the preferred behavior when implementing Scheme procedures with undefined return values.

The last one, `dict->list`, is the most complicated. In this example, it uses the `schemefct.list_` and `schemefct.cons` methods; it would also have been correct to import the `Pair` type from `psyche.types` and use them directly.

Example 7.4 `dict->list`

```
def dictToList(d):
    if not isinstance(d, dict):
        schemefct.error("Not a dictionary", d)

    # assoc is a python list of pairs
    assoc = [schemefct.cons(key, value) for (key, value) in d.items()]

    # schemefct.list_ requires a list of arguments
    return schemefct.list_(*assoc)
```

7.2.3 Using the New Procedures

For using the new procedures, two steps are left: creating a new environment and creating a new interpreter with this environment.

There are several ways of creating new environments. This Section will show how it is done in Psyche.

First of all, we add one more statement to the `psychedict` module we have created in the previous chapter:

Example 7.5 Creating the map from Scheme names to Python objects

```
procedures = {"make-dict": makeDict,
              "dict-key?": isDictKey,
              "dict-set!": dictSet,
              "dict-ref": dictRef,
              "dict->list": dictToList}
```

With this statement, we map Scheme procedure names to Python function objects.

Now we go to the code where we actually want to instantiate a new interpreter using these functions. We start out by creating a new Scheme environment and we update it with our new procedures.

Example 7.6 Creating the new environment

```
from psyche import interpreter
import psychedict

# code...

env = interpreter.SchemeEnvironment5()
env.update(psychedict.procedures)
```

That's it! With these two lines of code we have registered the new dictionary procedures with the Scheme environment.

Instantiating the new interpreter then becomes trivial.

Example 7.7 Creating the new interpreter

```
# code...
# code creating the new environment env

i = interpreter.Interpreter(env)

i.eval("(define d (make-dict))")
i.eval("(dict-set! d 4 4)")

print i.eval("d")
# this will print {4: 4}
```

Part II

The Psyche API

Chapter 8

psyche.interpreter

This module provides the Scheme Interpreter, the interactive Shell and the Scheme Environments.

exception `SchemeException` This exception is raised by the Shell and the Interpreter whenever a Scheme expression calls the built-in `(error...)` procedure. Its accompanying value is a tuple containing the arguments to the original `error` call.

exception `UndefinedException` This exception is raised by the Shell and the Interpreter whenever a Scheme command tries to reference an undefined variable. The referenced name can be obtained by calling the `name` method.

class `Environment(`[*parent*]`)` The base class for Scheme environments. It provides dictionary access by implementing the magic `__getitem__` and `__setitem__` methods. If a key is not found, it is subsequently looked up in the parent environment. By default, the parent environment is `None`

class `Interpreter(`[*environment*]`)` Instances of this class represent a Scheme interpreter using the specified environment. If *environment* is not specified, it defaults to an instance of `SchemeEnvironment5`.

class `SchemeEnvironment5()` Instances of this class represent a Scheme Environment as specified by the `(scheme-report-environment 5)` call in [1].

class `Shell()` Instances of the `Shell` can be used for interactive access to the interpreter. When possible, it provides a `readline` interface. Each Shell contains its own interpreter.

8.1 Environment Objects

Each `Environment` instance represents a single Scheme environment; its methods can be subdivided into two groups: methods providing dictionary-like access and methods providing typical Environment functionality.

Environments can contain both procedures and variables; the difference between them is irrelevant to an Environment.

`__getitem__`(*key*)

`__setitem__`(*key*, *value*) These methods allow dictionary access with the subscripting operator `[]`.

`keys()` Returns a list of all variables and procedure names in this environment.

`update`(*dict*) Updates the entries in this Environment with the entries specified in the dictionary *dict*. This is especially useful to add multiple user-defined procedures and variables at once.

`extend()` Extends this Environment by returning a new Environment that has the current Environment as its parent. The interpreter uses this method to implement local scopes.

8.2 Interpreter Objects

Each `Interpreter` instance represents a single Scheme interpreter. Different Interpreters do not interfere with each other; procedures defined in one interpreter will not be visible in other interpreters.

`USE_TAIL_RECURSION` Boolean value indicating whether or not the Interpreter should use tail recursion. By default, it has the value `1`. If set to false, the interpreter will not use tail recursion. However, by not using tail recursion Scheme expressions will be subjected to the Python recursion limit.

On the other hand, setting this flag to false will result in slightly faster execution since Psyche will not have to analyze the parse tree to mark those expressions in tail context.

`environment()` Returns the environment used by the interpreter.

`eval`(*line*) Evaluates the string *line* and returns the object resulting from evaluating the Scheme expression specified in *line*. If the result of this expression is undefined, `eval` will return `None`.

This method will raise all kinds of exceptions if anything went wrong.

`reset()` Resets the current environment to the environment given at initialization by removing all entries that resulted from evaluating Scheme code. More specifically, if any standard procedures (such as `(map...)`) were shadowed as the result of evaluating a Scheme expression, calling `reset` will remove the shadowing definition.

8.3 Shell Objects

Each `Shell` instance represents a single interactive Shell. Different Shells will not interfere with each other.

`complete(`*text*, *n*`)` Returns the nth completion of *text*. Used for tab-completion if the `readline` interface is active. The default implementation completes keywords and elements from the interpreter's environment.

`run()` Executes the read-eval-print loop until `scheme_input` returns `(quit)`.

`scheme_input()` Prompts on `sys.stdout` and reads from `sys.stdin` until an expression has been read with at least as many closing brackets as opening brackets. This expression is then returned as a joined line.

Chapter 9

`psyche.types`

Warning: Unfinished

Part III

Appendix

Appendix A

R5RS Compliance

What follows is a list of R5RS Chapters and Sections; for each Chapter and/or Section there is a piece of text explaining the compliance of Psyche.

The text "Full" is used to mark full compliance. This means that the class `R5RSTest` in file `interpretertest.py` tests at least all the examples in that Section or Chapter.

"Complete" means that for every implementation this feature depends on, the feature is fully implemented. For example, section 3.5 requires on the recognition of all primitive expressions; since not all primitive expressions are recognized, section 3.5 cannot be fully implemented; however, for those expressions that are recognized, section 3.5 is fully implemented.

1 Introduction N/A.

2.1 Identifiers Full.

2.2 Whitespace and Comments Full.

2.3 Other notations Unquoting and the backquote are not implemented; numbers with a `#` are not implemented either.

3.1 Variables, Syntactic Keywords and Regions Full.

3.2 Disjointness of Types Complete. The type associated with `port?` is not implemented.

3.3 External Representations Full.

3.4 Storage Model Full.

3.5 Proper Tail Recursion Complete.

4.1.1 Variable References Full.

4.1.2 Literal Expressions Full.

4.1.3 Procedure Calls Full.

4.1.4 Procedures Variable argument lists are not implemented.

4.1.5 Conditionals Full.

4.1.6 Assignments Full.

4.2.1 Conditionals The alternate form `=>` is not implemented. The `case` statement is not implemented.

4.2.2 Bindings constructs Only `let` is implemented. `let*` and `letrec` are not implemented.

4.2.3 Sequencing Not implemented.

4.2.4 Iteration Not implemented.

4.2.5 Delayed Evaluation Not implemented.

4.2.6 Quasiquotation Not implemented.

4.3 Macros Not implemented.

5.1 Programs Full.

5.2 Definitions Full.

5.3 Syntax Definitions Not implemented.

6.1 Equivalence Predicates Complete.

6.2.1 Numerical Types Only reals, rationals and integers are implemented.

6.2.2 Exactness Not implemented.

6.2.3 Implementation Restrictions N/A.

6.2.4 Syntax of Numerical Constants Only integer constants without a `#` are recognized.

6.2.5 The following procedures are not implemented: `complex?`, `real?`, `rational?`, `integer?`, `exact?`, `inexact?`, `max`, `min`, `gcd`, `lcm`, `numerator`, `denominator`, `floor`, `ceiling`, `truncate`, `round`, `rationalize`, `atan`, `sqrt`, `make-rectangular`, `make-polar`, `real-part`, `imag-part`, `magnitude`, `angle`, `exact->inexact`, `inexact->exact`

6.2.6 Numerical Input and Output Not implemented.

6.3.1 Booleans Full.

6.3.2 Pairs and Lists Full.

6.3.3 Symbols Full.

6.3.4 Characters Full.

6.3.5 Strings Full.

6.3.6 Vectors Full.

6.4 Control Features The following procedures are not implemented: `apply`, `map`, `for-each`, `force`, `call-with-current-continuation`, `values`, `call-with-values`, `dynamic wind`

6.5 Eval Not implemented.

6.6.1 Ports Not implemented.

6.6.2 Input Not implemented.

6.6.3 Output Not implemented.

6.6.4 System Interface Not implemented.

7 Formal Syntax and Semantics N/A.

A.1 Extra Features

Scheme recognizes two environments: the Scheme5 environment, which contains all procedures from R5RS, and the Interaction environment, which can contain extra procedures. This section describes the extra procedures in Psyche. They are enabled by default.

(`error` *obj* ...)
Raises an error with the specified objects as its arguments. Identical to the error procedure as used in SICP[2].

Appendix B

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

B.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

B.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

B.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

B.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher

of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

B.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

B.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

B.7 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

B.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

B.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

B.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Bibliography

[1] R. Kelsey, W. Clinger, J. Rees (eds.), Revised^5 Report on the Algorithmic Language Scheme, *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, September, 1998
`http://www.schemers.org/Documents/Standards/R5RS/`

[2] H. Abelson, G. J. Sussman, J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd edition, Massachusetts Institute of Technology, 1996
`http://www-mitpress.mit.edu/sicp/`

[3] G. Ewing, *Plex, a lexical analysis module for Python*, version 1.2
`http://www.cosc.canterbury.ac.nz/~greg/python/Plex/`

[4] D. Yoo, *pyscheme – Scheme in Python*
`http://www-hkn.eecs.berkeley.edu/~dyoo/python/pyscheme/`

[5] J. Aycock, *SPARK, Scanning, Parsing, and Rewriting Kit*, version 0.7
`http://pages.cpsc.ucalgary.ca/~aycock/spark/`