Implementing a Python to Scheme Compiler

Daniel P.M. Silva

April 26, 2004

Abstract

This paper describes a Python-to-Scheme compiler. The compiler translates Python code into its Scheme equivalent and provides a runtime system to model the Python environment. The generated Scheme code may be evaluated or used by DrScheme tools, giving Python programmers access to the entire DrScheme suite while writing in their favorite language, and giving Scheme programmers access to Python libraries.

1 Introduction

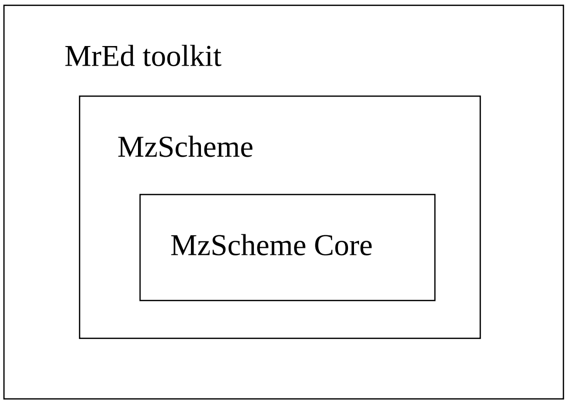
Programming languages are tools wielded by software developers. This paper describes a compiler that translates one such language, Python, into another, MzScheme. The translator allows Python developers to use PLT's software development tools and provides Scheme developers access to the rich Python runtime system.

Section 2 will present the necessary background for the project, followed in section 3 by the BNF grammar used by the compiler. Finally, the fourth section will describe the current implementation of the compiler.

2 Background

The Python programming language [1] was designed by Guido van Rossum in the early 1990s as a descendant of the ABC programming language, which was a teaching language created by van Rossum in the early 1980s. It includes a sizeable standard library, powerful primitive

Figure 1: PLT Scheme partial language hierarchy



data types, and a self-documenting system based on the language's emphasis on readability of source text. It is interpreted by a few different programs: C-Python [2], currently the most widely used interpreter for the Python programming language, is implemented in the C language. Another Python interpreter, Jython [3], is written in Java. The compiler this paper describes serves as yet another interpreter; it is written in MzScheme.

MzScheme [4] is an interpreter for the MzScheme programming language [5], which is a dialect of the Scheme language [6]. MzScheme compiles syntactically valid MzScheme language programs into the MzScheme Core language, a subset of the MzScheme language, before compiling the core language into an internal bytecode representation for evaluation.

MrEd [7] is a graphical user interface (GUI) toolkit that builds on the MzScheme interpreter and works uniformally across several platforms, namely Windows, Mac OS X, and the X Window System.

Originally meant for Scheme, DrScheme [8] 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, which accept the MzScheme Core language as their input. Because the internal DrScheme data structures representing MzScheme code also store source file location, any reference by a development tool or the MzScheme interpreter to the Core code can be mapped back to a reference to the original program text.

DrScheme is no longer a development environment only 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 2). When using any language from within the IDE, the program developer may use all of DrScheme's development tools, such as Syntax Check, which checks a program's syntax and highlights its bindings (Figure 3), or MrFlow, which analyses a program's possible flow of values. Also, any new tool added to the DrScheme IDE will automatically work with all languages that DrScheme now supports (Figure 4).

To support a new language, however, DrScheme needs software to translate programs written in the new language into MzScheme. In the case of adding Python support to DrScheme, this is the task of the Python-to-Scheme compiler. The compiler is packaged as a DrScheme language tool, thus introducing Python as a language in DrScheme's graphical list of choices (Figure 2). This paper describes the components that comprise the compiler. Section 3 presents the Python grammar used in this implementation. Section 4 describes the implementation itself.

3 Grammar

This section presents the grammar for Python programs currently supported by the Python-to-Scheme compiler.

The starting non-terminal is usually $\langle$*file_input*$\rangle$ (3.1.1), but $\langle$*eval_input*$\rangle$ (3.1.2) is used instead by the `eval` function.

If a non-terminal is named $\langle$*nonterm_list_plus*$\rangle$, it is assumed that it defines the regular expression *nonterm+*, while $\langle$*nonterm_list*$\rangle$ usually defines the regular expression *nonterm**, though it might be *(nonterm ,)** as well.

The empty string is represented by ε.

3.1 Program

3.1.1 file_input

This non-terminal is the result of the parser in the following situations:

- when parsing a complete Python program (from a file or from a string);

Figure 2: DrScheme language selection menu

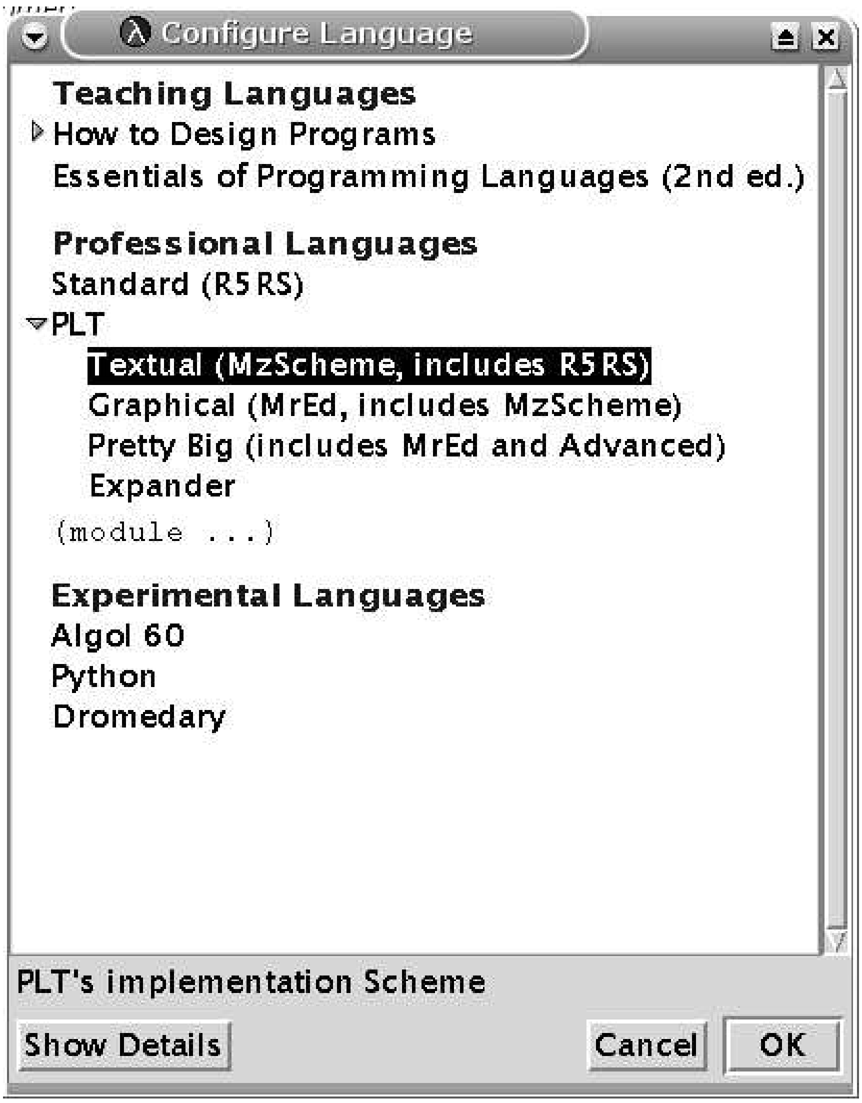


Figure 3: Syntax Check and Scheme

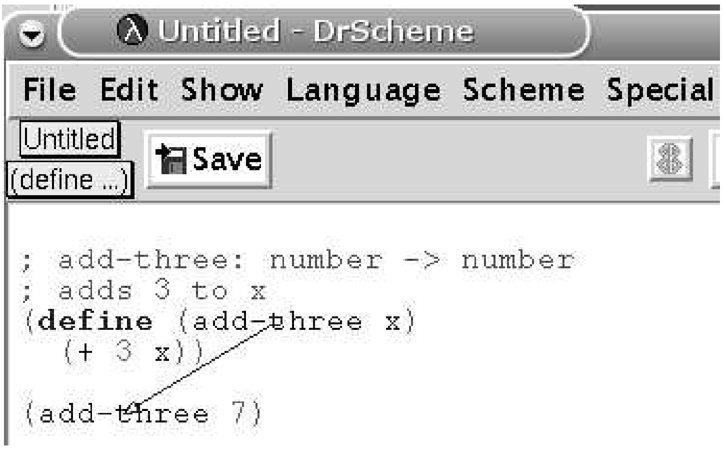
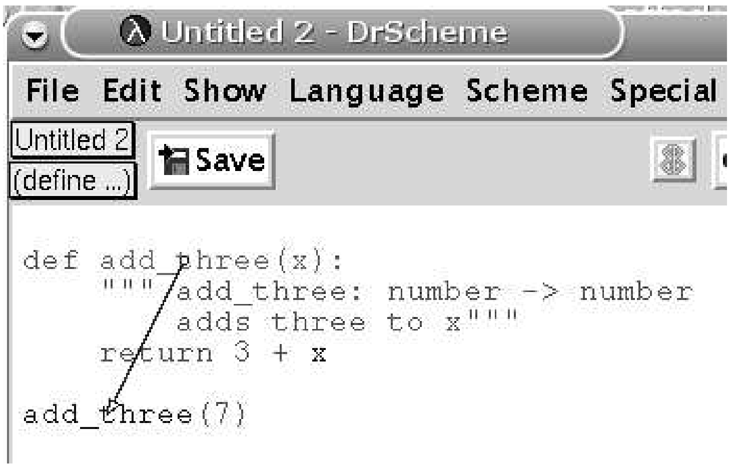


Figure 4: Syntax Check and Python



- when parsing a module;
- when parsing a string passed to the exec statement.

$\langle$*file_input*$\rangle$ ::= ε
| $\langle$*file_input*$\rangle$ NEWLINE
| $\langle$*file_input*$\rangle$ $\langle$*stmt 3.2.1*$\rangle$

3.1.2 eval_input

Eval Input is used by the Python `eval` and `input` functions. Only expressions (3.3.1), possibly followed by newlines, are allowed.

$\langle$*eval_input*$\rangle$ ::= $\langle$*tuple_or_test 3.3.1*$\rangle$
| $\langle$*eval_input*$\rangle$ NEWLINE

3.2 Statements

3.2.1 stmt

$\langle$*stmt*$\rangle$ ::= $\langle$*simple_stmt 3.2.2*$\rangle$
| $\langle$*compound_stmt 3.2.25*$\rangle$

Simple statements (3.2.2) span a single line (no new indentation levels). Compound statements (3.2.25) may span multiple lines.

3.2.2 simple_stmt

$\langle$*simple_stmt*$\rangle$ ::= $\langle$*small_stmt 3.2.3*$\rangle$ NEWLINE
| $\langle$*small_stmt 3.2.3*$\rangle$ ';' NEWLINE
| $\langle$*small_stmt 3.2.3*$\rangle$ ';' $\langle$*simple_stmt*$\rangle$

3.2.3 small_stmt

$\langle$*small_stmt*$\rangle$ ::= $\langle$*expr_stmt 3.2.4*$\rangle$
| $\langle$*print_stmt 3.2.7*$\rangle$
| $\langle$*del_stmt 3.2.8*$\rangle$
| $\langle$*pass_stmt 3.2.9*$\rangle$
| $\langle$*flow_stmt 3.2.10*$\rangle$
| $\langle$*import_stmt 3.2.16*$\rangle$
| $\langle$*global_stmt 3.2.22*$\rangle$

| ⟨*exec_stmt 3.2.23*⟩
| ⟨*assert_stmt 3.2.24*⟩

3.2.4 expr_stmt

⟨*expr_stmt*⟩ ::= ⟨*test 3.3.3*⟩ ⟨*augassign 3.2.6*⟩ ⟨*tuple_or_test 3.3.1*⟩
| ⟨*testlist_list_plus 3.2.5*⟩

An expression statement consists of either a mutative operation or a ⟨*testlist_list_plus*⟩ (3.2.5).

3.2.5 testlist_list_plus

⟨*testlist_list_plus*⟩ ::= ⟨*tuple_or_test 3.3.1*⟩
| ⟨*tuple_or_test 3.3.1*⟩ '`=`' ⟨*testlist_list_plus*⟩

A ⟨*testlist_list_plus*⟩ is an assignment or an expression (3.3.1) which will be displayed in the output of the interpreter.

3.2.6 augassign

⟨*augassign*⟩ ::= '`+=`'
| '`-=`'
| '`*=`'
| '`/=`'
| '`%=`'
| '`&=`'
| '`|=`'
| '`^=`'
| '`<<=`'
| '`>>=`'
| '`**=`'
| '`//=`'

3.2.7 print_stmt

⟨*print_stmt*⟩ ::= '`print`' ⟨*test_list 3.3.4*⟩
| '`print`' '`>>`' ⟨*test_list 3.3.4*⟩

From the Python Reference [1], section 6.6:

> `print` has an extended form, sometimes referred to as "print chevron." In this form, the first expression after the `>>`

> must evaluate to a “file-like” object, specifically an object that has a `write` method, or `None`.

3.2.8 del_stmt

See the Python Reference [1], section 6.5.

⟨*del_stmt*⟩ ::= ‘`del`’ ⟨*target_tuple_or_expr 3.3.34*⟩

3.2.9 pass_stmt

⟨*pass_stmt*⟩ ::= ‘`pass`’

3.2.10 flow_stmt

A flow statement directs or modifies program flow.

```
⟨flow_stmt⟩ ::= ⟨break_stmt 3.2.11⟩
              | ⟨continue_stmt 3.2.12⟩
              | ⟨return_stmt 3.2.13⟩
              | ⟨raise_stmt 3.2.14⟩
              | ⟨yield_stmt 3.2.15⟩
```

3.2.11 break_stmt

⟨*break_stmt*⟩ ::= ‘`break`’

From the Python reference [1], section 6.10:

> `break` may only occur syntactically nested in a for or while loop, but not nested in a function or class definition within that loop.

3.2.12 continue_stmt

⟨*continue_stmt*⟩ ::= ‘`continue`’

From the Python reference [1], section 6.11:

> `continue` may only occur syntactically nested in a for or while loop, but not nested in a function or class definition or try statement within that loop.

3.2.13 return_stmt

⟨*return_stmt*⟩ ::= ‘`return`’ ⟨*tuple_or_test 3.3.1*⟩
| ‘`return`’

From the Python reference [1], section 6.7:

> In a generator function (see 3.2.15), the `return` statement is not allowed to include an expression list (3.3.1). In that context, a bare return indicates that the generator is done and will cause `StopIteration` to be raised.

3.2.14 raise_stmt

⟨*raise_stmt*⟩ ::= ‘`raise`’
| ‘`raise`’ ⟨*test 3.3.3*⟩
| ‘`raise`’ ⟨*test 3.3.3*⟩ ‘`,`’ ⟨*test 3.3.3*⟩
| ‘`raise`’ ⟨*test 3.3.3*⟩ ‘`,`’ ⟨*test 3.3.3*⟩ ‘`,`’ ⟨*test 3.3.3*⟩

From the Python reference [1], section 6.9:

> `raise` may have up to three arguments, the first being the type of the exception, the second being its value, and the third being a traceback.

3.2.15 yield_stmt (NOT YET IMPLEMENTED)

⟨*yield_stmt*⟩ ::= ‘`yield`’ ⟨*tuple_or_test 3.3.1*⟩

From the Python reference [1], section 6.8:

> The `yield` statement is only used when defining a generator function, and is only used in the body of the generator function. Using a yield statement in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

3.2.16 import_stmt

⟨*import_stmt*⟩ ::= ⟨*import_stmt1 3.2.17*⟩
| ‘`from`’ ⟨*dotted_name 3.2.21*⟩ ‘`import`’ ‘`*`’
| ‘`from`’ ⟨*dotted_name 3.2.21*⟩ ‘`import`’ ⟨*import_stmt2 3.2.18*⟩

See the Python reference [1], section 6.12.

3.2.17 import_stmt1

⟨*import_stmt1*⟩ ::= '`import`' ⟨*dotted_as_name 3.2.20*⟩
| ⟨*import_stmt1*⟩ '`,`' ⟨*dotted_as_name 3.2.20*⟩

3.2.18 import_stmt2

⟨*import_stmt2*⟩ ::= ⟨*import_as_name 3.2.19*⟩
| ⟨*import_as_name 3.2.19*⟩ '`,`' ⟨*import_stmt2*⟩

3.2.19 import_as_name

⟨*import_as_name*⟩ ::= ⟨*ident 3.3.38*⟩ NAME ⟨*ident 3.3.38*⟩
| ⟨*ident 3.3.38*⟩

3.2.20 dotted_as_name

⟨*dotted_as_name*⟩ ::= ⟨*dotted_name 3.2.21*⟩ NAME ⟨*ident 3.3.38*⟩
| ⟨*dotted_name 3.2.21*⟩

3.2.21 dotted_name

⟨*dotted_name*⟩ ::= ⟨*ident 3.3.38*⟩
| ⟨*ident 3.3.38*⟩ '`.`' ⟨*dotted_name*⟩

3.2.22 global_stmt

⟨*global_stmt*⟩ ::= '`global`' ⟨*ident 3.3.38*⟩
| ⟨*global_stmt 3.2.22*⟩ '`,`' ⟨*ident 3.3.38*⟩

From the Python reference [1], section 6.13:

> The global statement means that the listed identifiers are to be interpreted as globals. Names listed in a global statement must not be used in the same code block textually preceding that global statement (not yet implemented). Names listed in a global statement must not be defined as formal parameters or in a for loop control target, class definition, function definition, or import statement (not yet implemented).
>
> Programmer's note: the global is a directive to the parser. It applies only to code parsed at the same time as the global statement. In particular, a global statement contained in

an exec statement does not affect the code block containing the exec statement, and code contained in an exec statement is unaffected by global statements in the code containing the exec statement. The same applies to the eval(), execfile() and compile() functions.

For example, in:

```
exec "global x"
x = 2
```

the generated code will define a new `x`, not modify an existing one.

3.2.23 exec_stmt (NOT YET IMPLEMENTED)

$\langle$*exec_stmt*$\rangle$ ::= '`exec`' $\langle$*expr 3.3.12*$\rangle$
| '`exec`' $\langle$*expr 3.3.12*$\rangle$ '`in`' $\langle$*test 3.3.3*$\rangle$
| '`exec`' $\langle$*expr 3.3.12*$\rangle$ '`in`' $\langle$*test 3.3.3*$\rangle$ '`,`' $\langle$*test 3.3.3*$\rangle$

See the Python reference [1], section 6.14.

3.2.24 assert_stmt

$\langle$*assert_stmt*$\rangle$ ::= '`assert`' $\langle$*test 3.3.3*$\rangle$
| '`assert`' $\langle$*test 3.3.3*$\rangle$ '`,`' $\langle$*test 3.3.3*$\rangle$

From the Python reference [1], section 6.2:

The simple form, "`assert expression`", is equivalent to

```
if __debug__:
   if not expression: raise AssertionError
```

The extended form, "`assert expression1, expression2`", is equivalent to

```
if __debug__:
   if not expression1: raise AssertionError, expression2
```

3.2.25 compound_stmt

$\langle$*compound_stmt*$\rangle$ ::= $\langle$*if_stmt 3.2.26*$\rangle$
| $\langle$*while_stmt 3.2.30*$\rangle$
| $\langle$*for_stmt 3.2.31*$\rangle$
| $\langle$*try_stmt 3.2.32*$\rangle$

| ⟨*funcdef 3.2.35*⟩
| ⟨*classdef 3.2.40*⟩

Compound statements can span multiple lines, so they may introduce a new indentation level.

3.2.26 if_stmt

⟨*if_stmt*⟩ ::= '`if`' ⟨*test 3.3.3*⟩ '`:`' ⟨*suite 3.2.28*⟩ ⟨*elif_list 3.2.27*⟩
| '`if`' ⟨*test 3.3.3*⟩ '`:`' ⟨*suite 3.2.28*⟩ ⟨*elif_list 3.2.27*⟩ '`else`' '`:`' ⟨*suite 3.2.28*⟩

3.2.27 elif_list

⟨*elif_list*⟩ ::= ε
| ⟨*elif_list*⟩ '`elif`' ⟨*test 3.3.3*⟩ '`:`' ⟨*suite 3.2.28*⟩

3.2.28 suite

⟨*suite*⟩ ::= ⟨*simple_stmt 3.2.2*⟩
| NEWLINE INDENT ⟨*stmt_list_plus 3.2.29*⟩ DEDENT

The INDENT token indicates a new indentation level. Similarly, the DEDENT token indicates a return to the previous indentation level.

3.2.29 stmt_list_plus

⟨*stmt_list_plus*⟩ ::= ⟨*stmt 3.2.1*⟩
| ⟨*stmt_list_plus*⟩ ⟨*stmt 3.2.1*⟩

3.2.30 while_stmt

⟨*while_stmt*⟩ ::= '`while`' ⟨*test 3.3.3*⟩ '`:`' ⟨*suite 3.2.28*⟩
| '`while`' ⟨*test 3.3.3*⟩ '`:`' ⟨*suite 3.2.28*⟩ '`else`' ⟨*suite 3.2.28*⟩

3.2.31 for_stmt

⟨*for_stmt*⟩ ::= '`for`' ⟨*target_tuple_or_expr 3.3.34*⟩ '`in`' ⟨*tuple_or_test 3.3.1*⟩ '`:`' ⟨*suite 3.2.28*⟩
| '`for`' ⟨*target_tuple_or_expr 3.3.34*⟩ '`in`' ⟨*tuple_or_test 3.3.1*⟩ '`:`' ⟨*suite 3.2.28*⟩ '`else`' '`:`' ⟨*suite 3.2.28*⟩

3.2.32 try_stmt

$\langle$*try_stmt*$\rangle$ ::= `try` `:` $\langle$*suite 3.2.28*$\rangle$ $\langle$*except_clause_list_plus 3.2.33*$\rangle$
| `try` `:` $\langle$*suite 3.2.28*$\rangle$ $\langle$*except_clause_list_plus 3.2.33*$\rangle$ `else` `:` $\langle$*suite 3.2.28*$\rangle$
| `try` `:` $\langle$*suite 3.2.28*$\rangle$ `finally` `:` $\langle$*suite 3.2.28*$\rangle$

3.2.33 except_clause_list_plus

$\langle$*except_clause_list_plus*$\rangle$ ::= $\langle$*except_clause 3.2.34*$\rangle$ `:` $\langle$*suite 3.2.28*$\rangle$
| $\langle$*except_clause_list_plus 3.2.33*$\rangle$ $\langle$*except_clause 3.2.34*$\rangle$ `:` $\langle$*suite 3.2.28*$\rangle$

3.2.34 except_clause

$\langle$*except_clause*$\rangle$ ::= `except`
| `except` $\langle$*test 3.3.3*$\rangle$
| `except` $\langle$*test 3.3.3*$\rangle$ `,` $\langle$*test 3.3.3*$\rangle$

3.2.35 funcdef

See the Python reference [1], section 7.5.

$\langle$*funcdef*$\rangle$ ::= `def` $\langle$*ident 3.3.38*$\rangle$ $\langle$*parameters 3.2.36*$\rangle$ `:` $\langle$*suite 3.2.28*$\rangle$

3.2.36 parameters

$\langle$*parameters*$\rangle$ ::= `(` `)`
| `(` $\langle$*varargslist 3.2.37*$\rangle$ `)`

3.2.37 varargslist

$\langle$*varargslist*$\rangle$ ::= `**` $\langle$*ident 3.3.38*$\rangle$
| `*` $\langle$*ident 3.3.38*$\rangle$
| `*` $\langle$*ident 3.3.38*$\rangle$ `,` `**` $\langle$*ident 3.3.38*$\rangle$
| $\langle$*fpdef 3.2.38*$\rangle$ `,`
| $\langle$*fpdef 3.2.38*$\rangle$ `=` $\langle$*test 3.3.3*$\rangle$ `,`
| $\langle$*fpdef 3.2.38*$\rangle$
| $\langle$*fpdef 3.2.38*$\rangle$ `=` $\langle$*test 3.3.3*$\rangle$
| $\langle$*fpdef 3.2.38*$\rangle$ `,` $\langle$*varargslist 3.2.37*$\rangle$
| $\langle$*fpdef 3.2.38*$\rangle$ `=` $\langle$*test 3.3.3*$\rangle$ `,` $\langle$*varargslist 3.2.37*$\rangle$

From the Python reference [1], section 7.5:

If the form "*identifier" is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form "**identifier" is present, it is initialized to a new dictionary receiving any excess keyword arguments, defaulting to a new empty dictionary.

If a parameter has a default value, all following parameters must also have a default value—this is a syntactic restriction that is not expressed by the grammar, but is checked by the parser.

3.2.38 fpdef

$\langle$*fpdef*$\rangle$ ::= $\langle$*ident 3.3.38*$\rangle$
| '`(`' $\langle$*fplist 3.2.39*$\rangle$ '`)`'

A function parameter is either an identifier or a tuple that will be unpacked. For example, in:

```
def f(x, (y, z)):
    pass

f(1,(2,3))
```

when f is called, `x` is bound to `1`, `y` is bound to `2`, and `z` is bound to `3`.

3.2.39 fplist

$\langle$*fplist*$\rangle$ ::= $\langle$*fpdef 3.2.38*$\rangle$
| $\langle$*fpdef 3.2.38*$\rangle$ '`,`'
| $\langle$*fpdef 3.2.38*$\rangle$ '`,`' $\langle$*fplist*$\rangle$

3.2.40 classdef

$\langle$*classdef*$\rangle$ ::= '`class`' $\langle$*ident 3.3.38*$\rangle$ '`:`' $\langle$*suite 3.2.28*$\rangle$
| '`class`' $\langle$*ident 3.3.38*$\rangle$ '`(`' $\langle$*test 3.3.3*$\rangle$ '`)`' '`:`' $\langle$*suite 3.2.28*$\rangle$
| '`class`' $\langle$*ident 3.3.38*$\rangle$ '`(`' $\langle$*testlist 3.3.5*$\rangle$ '`)`' '`:`' $\langle$*suite 3.2.28*$\rangle$

In the CPython interpreter (version 2.2), "old-style" classes are defined by `class classname(superclasses)`, and "new-style" classes are defined by `class classname`. Such distinction is not made here; all classes are "new-style" classes.

3.3 Expressions

3.3.1 tuple_or_test

$\langle tuple\_or\_test \rangle$ ::= $\langle tuple\ 3.3.2 \rangle$
| $\langle test\ 3.3.3 \rangle$

3.3.2 tuple

$\langle tuple \rangle$ ::= $\langle testlist\ 3.3.5 \rangle$

A tuple has the same syntactic rules as a testlist (3.3.5), but the name "tuple" is more descriptive sometimes.

3.3.3 test

See the Python reference [1], section 5.10.

$\langle test \rangle$ ::= $\langle or\_test\ 3.3.7 \rangle$
| $\langle lambdef\ 3.3.6 \rangle$

3.3.4 test_list

$\langle test\_list \rangle$ ::= ε
| $\langle test\ 3.3.3 \rangle$
| $\langle test\ 3.3.3 \rangle$ ',' $\langle test\_list\ 3.3.4 \rangle$

This non-terminal is only used by `print` statements (3.2.7).

3.3.5 testlist

$\langle testlist \rangle$ ::= $\langle test\ 3.3.3 \rangle$ ','
| $\langle test\ 3.3.3 \rangle$ ',' $\langle test\ 3.3.3 \rangle$
| $\langle test\ 3.3.3 \rangle$ ',' $\langle testlist\ 3.3.5 \rangle$

See note for $\langle test\_list \rangle$ (3.3.4). A $\langle testlist \rangle$ is never empty. This non-terminal is used by:

- $\langle tuple \rangle$ (3.3.2);
- $\langle listmaker \rangle$ (3.3.26);
- $\langle testlist\_safe \rangle$ (3.3.36);
- and class definitions: $\langle classdef \rangle$ (3.2.40).

3.3.6 lambdef

⟨lambdef⟩ ::= 'lambda' ⟨varargslist 3.2.37⟩ ':' ⟨test 3.3.3⟩
| 'lambda' ':' ⟨test 3.3.3⟩

3.3.7 or_test

⟨or_test⟩ ::= ⟨and_test 3.3.8⟩
| ⟨or_test⟩ 'or' ⟨and_test 3.3.8⟩

3.3.8 and_test

⟨and_test⟩ ::= ⟨not_test 3.3.9⟩
| ⟨and_test⟩ 'and' ⟨not_test 3.3.9⟩

3.3.9 not_test

⟨not_test⟩ ::= 'not' ⟨not_test⟩
| ⟨comparison 3.3.10⟩

3.3.10 comparison

⟨comparison⟩ ::= ⟨expr 3.3.12⟩
| ⟨comparison⟩ ⟨comp_op 3.3.11⟩ ⟨expr 3.3.12⟩

3.3.11 comp_op

⟨comp_op⟩ ::= '<'
| '>'
| '=='
| '>='
| '<='
| '<>'
| '!='
| 'in'
| 'not' 'in'
| 'is'
| 'is' 'not'

3.3.12 expr

⟨expr⟩ ::= ⟨xor_expr 3.3.13⟩
| ⟨expr⟩ '|' ⟨xor_expr 3.3.13⟩

3.3.13 xor_expr

$\langle xor\_expr \rangle$ ::= $\langle and\_expr\ 3.3.14 \rangle$
| $\langle xor\_expr \rangle$ `^` $\langle and\_expr\ 3.3.14 \rangle$

3.3.14 and_expr

$\langle and\_expr \rangle$ ::= $\langle shift\_expr\ 3.3.15 \rangle$
| $\langle and\_expr \rangle$ `&` $\langle shift\_expr\ 3.3.15 \rangle$

3.3.15 shift_expr

$\langle shift\_expr \rangle$ ::= $\langle arith\_expr\ 3.3.16 \rangle$
| $\langle shift\_expr \rangle$ `<<` $\langle arith\_expr\ 3.3.16 \rangle$
| $\langle shift\_expr \rangle$ `>>` $\langle arith\_expr\ 3.3.16 \rangle$

3.3.16 arith_expr

$\langle arith\_expr \rangle$ ::= $\langle term\ 3.3.17 \rangle$
| $\langle arith\_expr \rangle$ `+` $\langle term\ 3.3.17 \rangle$
| $\langle arith\_expr \rangle$ `-` $\langle term\ 3.3.17 \rangle$

3.3.17 term

$\langle term \rangle$ ::= $\langle factor\ 3.3.18 \rangle$
| $\langle term \rangle$ `*` $\langle factor\ 3.3.18 \rangle$
| $\langle term \rangle$ `/` $\langle factor\ 3.3.18 \rangle$
| $\langle term \rangle$ `%` $\langle factor\ 3.3.18 \rangle$
| $\langle term \rangle$ `//` $\langle factor\ 3.3.18 \rangle$

3.3.18 factor

$\langle factor \rangle$ ::= `+` $\langle factor \rangle$
| `-` $\langle factor \rangle$
| `~` $\langle factor \rangle$
| $\langle power\ 3.3.19 \rangle$

3.3.19 power

$\langle power \rangle$ ::= $\langle atom\ 3.3.24 \rangle$ $\langle trailer\_list\ 3.3.20 \rangle$
| $\langle atom\ 3.3.24 \rangle$ $\langle trailer\_list\ 3.3.20 \rangle$ `**` $\langle factor\ 3.3.18 \rangle$

A trailer list is an index operation (e.g., the `[7]` in `x[7]`), a function argument list, or a class attribute reference.

3.3.20 trailer_list

⟨*trailer_list*⟩	::=	ε
	|	⟨*trailer 3.3.21*⟩ ⟨*trailer_list*⟩

3.3.21 trailer

⟨*trailer*⟩	::=	‘(’ ‘)’
	|	‘(’ ⟨*arglist 3.3.22*⟩ ‘)’
	|	‘[’ ⟨*subscriptlist 3.3.30*⟩ ‘]’
	|	‘[’ ⟨*subscript 3.3.31*⟩ ‘]’
	|	‘.’ ⟨*ident 3.3.38*⟩

3.3.22 arglist

⟨*arglist*⟩	::=	‘**’ ⟨*test 3.3.3*⟩
	|	‘*’ ⟨*test 3.3.3*⟩
	|	‘*’ ⟨*test 3.3.3*⟩ ‘,’ ‘**’ ⟨*test 3.3.3*⟩
	|	⟨*argument 3.3.23*⟩
	|	⟨*argument 3.3.23*⟩ ‘,’
	|	⟨*argument 3.3.23*⟩ ‘,’ ⟨*arglist 3.3.22*⟩

3.3.23 argument

⟨*argument*⟩	::=	⟨*test 3.3.3*⟩
	|	⟨*ident 3.3.38*⟩ ‘=’ ⟨*test 3.3.3*⟩

3.3.24 atom

⟨*atom*⟩	::=	‘(’ ⟨*tuple_or_test 3.3.1*⟩ ‘)’
	|	‘[’ ⟨*listmaker 3.3.26*⟩ ‘]’
	|	‘{’ ⟨*dictmaker 3.3.37*⟩ ‘}’
	|	‘(’ ‘)’
	|	‘[’ ‘]’
	|	‘{’ ‘}’
	|	‘‘’ ⟨*tuple_or_test 3.3.1*⟩ ‘‘’

- ( ) is the empty tuple
- `[ ]` is the empty list
- { } is the empty dictionary
- ‘... ‘ is a shortcut for `repr(...)`

3.3.25 string_list_plus

⟨*string_list_plus*⟩ ::= STRING
 | STRING ⟨*string_list_plus*⟩

3.3.26 listmaker

⟨*listmaker*⟩ ::= ⟨*test 3.3.3*⟩ ⟨*list_for 3.3.28*⟩
 | ⟨*testlist 3.3.5*⟩
 | ⟨*test 3.3.3*⟩

3.3.27 list_iter

⟨*list_iter*⟩ ::= ⟨*list_for 3.3.28*⟩
 | ⟨*list_if 3.3.29*⟩

3.3.28 list_for

⟨*list_for*⟩ ::= '`for`' ⟨*target_tuple_or_expr 3.3.34*⟩ '`in`' ⟨*testlist_safe 3.3.36*⟩
 | '`for`' ⟨*target_tuple_or_expr 3.3.34*⟩ '`in`' ⟨*testlist_safe 3.3.36*⟩ ⟨*list_iter 3.3.27*⟩

See the Python reference [1], section 5.2.4.

3.3.29 list_if

⟨*list_if*⟩ ::= '`if`' ⟨*test 3.3.3*⟩
 | '`if`' ⟨*test 3.3.3*⟩ ⟨*list_iter 3.3.27*⟩

3.3.30 subscriptlist

⟨*subscriptlist*⟩ ::= ⟨*subscript 3.3.31*⟩ '`,`'
 | ⟨*subscript 3.3.31*⟩ '`,`' ⟨*subscript 3.3.31*⟩
 | ⟨*subscript 3.3.31*⟩ '`,`' ⟨*subscriptlist 3.3.30*⟩

3.3.31 subscript

⟨*subscript*⟩ ::= '`:`'
 | '`:`' ⟨*test 3.3.3*⟩
 | ⟨*test 3.3.3*⟩ '`:`'
 | ⟨*test 3.3.3*⟩ '`:`' ⟨*test 3.3.3*⟩
 | '`:`' ⟨*sliceop 3.3.32*⟩
 | '`:`' ⟨*test 3.3.3*⟩ ⟨*sliceop 3.3.32*⟩

| ⟨*test 3.3.3*⟩ ':' ⟨*sliceop 3.3.32*⟩
| ⟨*test 3.3.3*⟩ ':' ⟨*test 3.3.3*⟩ ⟨*sliceop 3.3.32*⟩
| ⟨*test 3.3.3*⟩
| '...'

3.3.32 sliceop

⟨*sliceop*⟩ ::= ':'
| ':' ⟨*test 3.3.3*⟩

This non-terminal occurs only inside of a ⟨*subscript*⟩ (3.3.31).

3.3.33 exprlist

⟨*exprlist*⟩ ::= ⟨*expr 3.3.12*⟩ ','
| ⟨*expr 3.3.12*⟩ ',' ⟨*expr 3.3.12*⟩
| ⟨*expr 3.3.12*⟩ ',' ⟨*exprlist*⟩

3.3.34 target_tuple_or_expr

⟨*target_tuple_or_expr*⟩ ::= ⟨*target_tuple 3.3.35*⟩
| ⟨*expr 3.3.12*⟩

3.3.35 target_tuple

⟨*target_tuple*⟩ ::= ⟨*exprlist 3.3.33*⟩

3.3.36 testlist_safe

⟨*testlist_safe*⟩ ::= ⟨*test 3.3.3*⟩
| ⟨*test 3.3.3*⟩ ',' ⟨*testlist 3.3.5*⟩

Used in list comprehensions (3.3.28). A safe `testlist` cannot end with a comma.

3.3.37 dictmaker

⟨*dictmaker*⟩ ::= ⟨*test 3.3.3*⟩ ':' ⟨*test 3.3.3*⟩
| ⟨*test 3.3.3*⟩ ':' ⟨*test 3.3.3*⟩ ','
| ⟨*test 3.3.3*⟩ ':' ⟨*test 3.3.3*⟩ ',' ⟨*dictmaker*⟩

3.3.38 ident

$\langle ident \rangle$ ::= NAME

The NAME token is a valid Python identifier. A valid Python identifier is an alphanumeric sequence of characters starting with a letter. The underscore character (_) counts as a letter. The language is case-sensitive.

4 Implementation

This section describes the current implementation of the Python-to-Scheme compiler. The compiler is composed of three major components: the front-end, which uses a lexical analyzer (scanner) 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 a library that the generated code makes use of (Figure 5). This section delineates these three components. Section 4.1 describes the scanner and parser; section 4.2, the code generator; and section 4.3, the runtime system.

4.1 Lexical and Syntax Analysis

Python program text is read by the lexical analyzer—created by Scott Owens of Utah University—and transformed into tokens. Tokens are either reserved keywords, represented as symbols, or input items, such as literal values, identifiers, and indentation directives. The lexical analyzer outputs tokens, which are consumed by the parser.

The parser—also provided by Scott Owens, along with the grammar it uses—accepts a stream of tokens from the lexical analyzer and generates abstract syntax trees (ASTs) according to the grammar described in section 3.

Abstract syntax trees are data structures representing the terms present in a program. For example, the Python expression `x + 3` is a binary expression (section 3.3.16). This binary expression is made up of the identifier `x`, the number `3`, and the addition operation symbol (`+`). Figure 6 displays the UML [9] type hierarchy of the data structures necessary to represent `x + 3`. These are the MzScheme classes (from the class system provided by `mzlib/class.ss`) used by the parser to create its output of ASTs. The `object%` base class of

Figure 5: Compiler overview

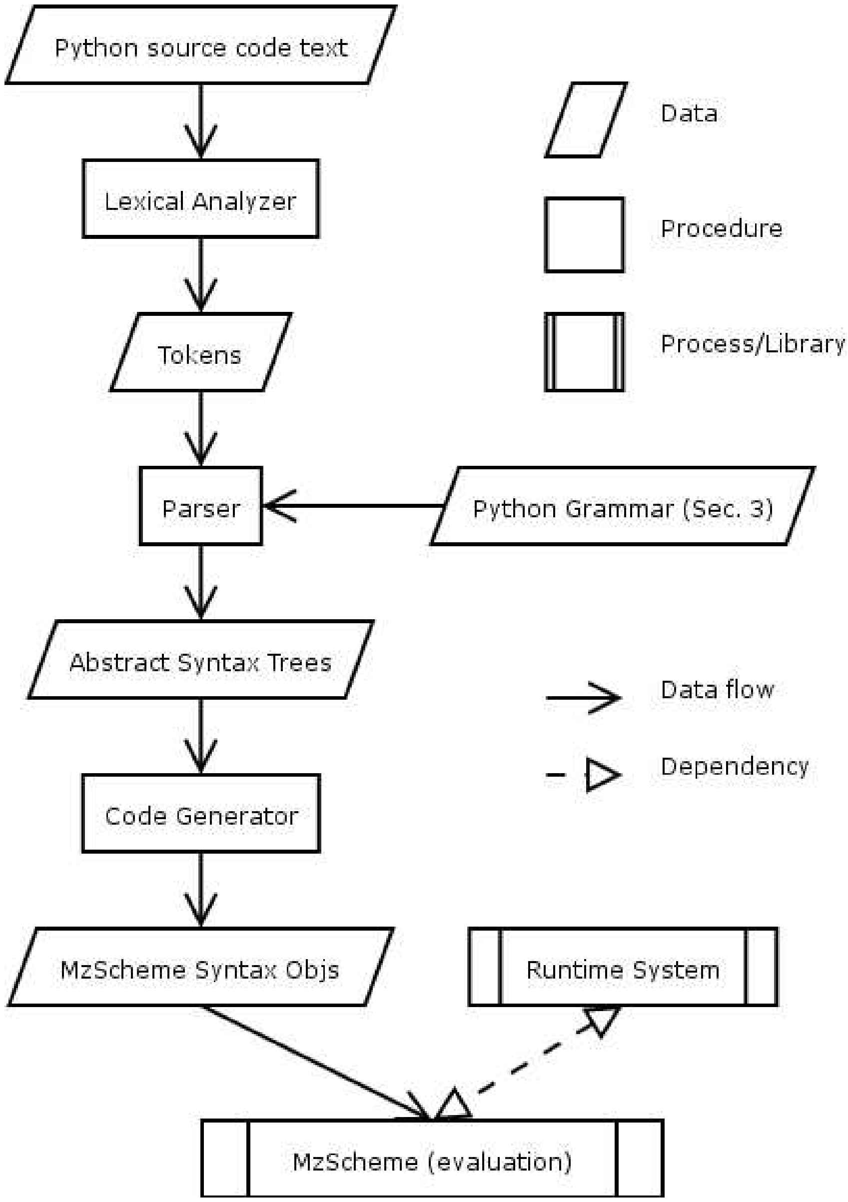
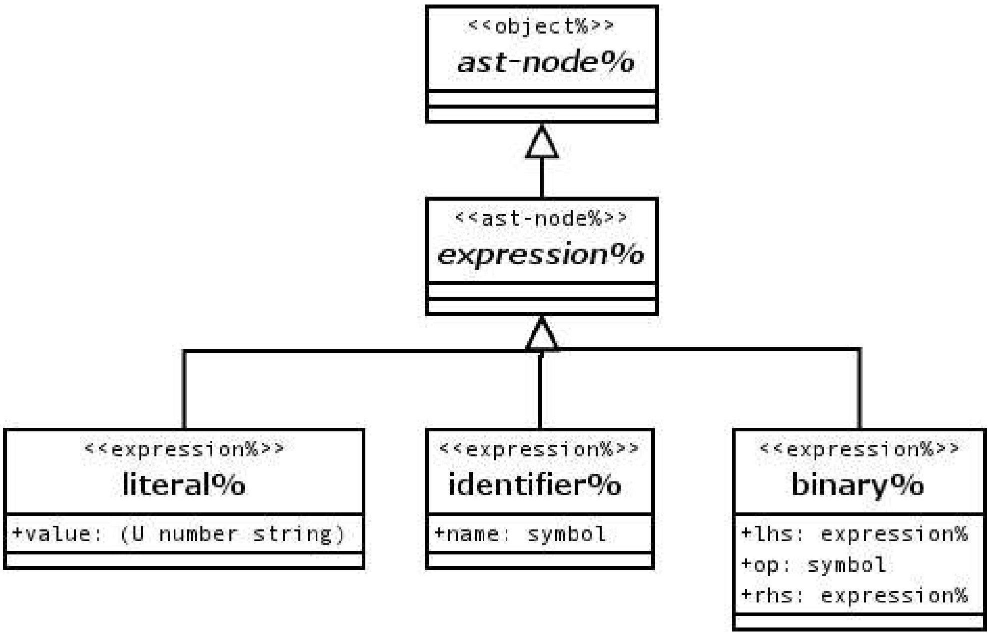


Figure 6: AST type hierarchy



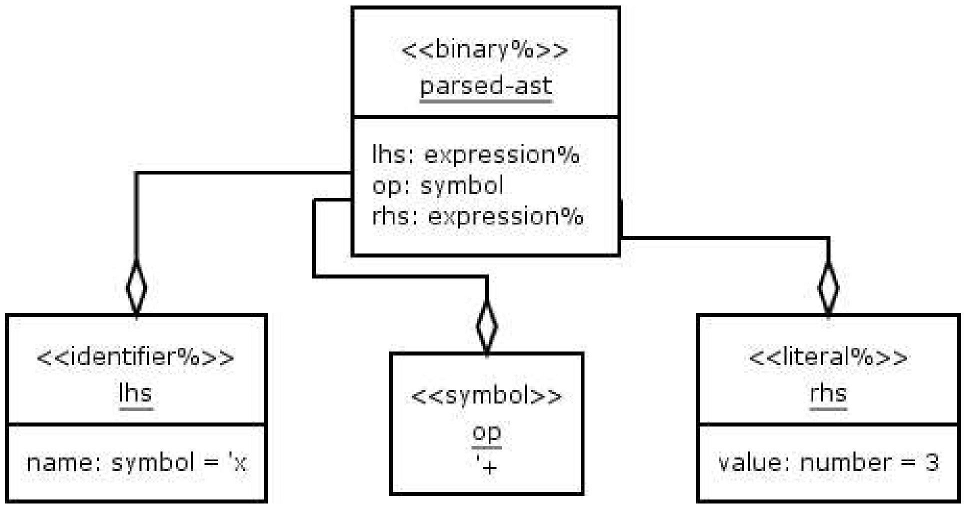
`ast-node%` is the MzScheme `object%` class provided by the PLT class system. Figure 7 displays the objects involved in the representation of the expression's syntax tree. These are instantiations of the classes in Figure 6.

The parser produces a list of abstract syntax trees, one for each top-level statement in the original program. They are then accepted by the code generator as input.

4.2 Code Generation

The code generator must produce Scheme code from a list of ASTs. It does so by converting the supplied ASTs (by the parser described in section 4.1) into Scheme code that is operationally equivalent to the original Python code. The following subsections explain the Scheme code generated for the most important parts of the Python language.

Figure 7: AST for x + 3



4.2.1 Function Definitions

A python function has a few features not present in the syntax for 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. To illustrate this, let us define the following Python function, which will consume three required arguments, a rest argument (for spilled-over positional arguments), and a dictionary argument:

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

Keeping this function in mind, consider these two calls to `f`:

```
Welcome to DrScheme, version 203.10-cvs27apr2003.
Language: Python.
> f(1, 2, 3, 4, 5, 6, test = 8)
1 2 3
(4, 5, 6)
```

```
{'test': 8}
> f(1, z = 2, y = 3, test = 4)
1 3 2
()
{'test': 4}
```

This behavior is emulated by converting the function definition into the following Scheme code, where `procedure->py-function%` takes a procedure, its name, and its argument names to produce an object representing the Python function:

```
(define 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 x y z))
          (py-print #f (list rest))
          (py-print #f (list dict))
          py-none))))
   'f (list 'x 'y 'z) null 'rest 'dict))
```

In this translation, `procedure->py-function%`, `list->py-tuple%`, and `py-print` are runtime system functions, and `py-none` is a runtime system constant representing the Python `None`. We will delay further discussion of the runtime system until section 4.3. The more interesting part of the example is the generated lambda.

In the function described by the `opt-lambda` expression, the compiler shifts the dictionary argument, `dict`, into the first position among the function's parameters (where the function call mechanism knows to put keyword arguments), and places a `let` binding to convert the Scheme list of leftover arguments—the `rest` in the `opt-lambda` parameter list—into the Python tuple of leftover arguments. After handling parameters, the compiler generates the function's `return` handler, followed by the function body itself.

The meaning of a Python `return` statement can be emulated with a Scheme escape continuation, which represents the rest of the program. A label is generated for the "return" continuation; any `return` *value* statement would be translated into `(return10846` *value*`)`. With the returning mechanism established, the function body can now be generated.

The body of a Python function being a sequence of statements, the code generator translates that into a sequence of Scheme commands (expressions evaluated for side-effect) plus the default return value, `None`.

While it is fairly simple to convert a Python function into Scheme text that looks like a typical function definition, functions tend to look more like `let` bindings when defined as class methods, as the next section displays.

4.2.2 Class Definitions

The Python built-in `type` functor returns a new type (i.e., a new class) when given a name, tuple of parents, and dictionary of member fields and methods. The compiler generates a Python class object as the result of a call to `type`; that is, a statement of the form `class C...` is treated as `C = type("C",...)`.

Consider this small Python class:

```
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 short example compiles into thirty lines of Scheme code, which we now dissect:

```
01(define C
02  (python-method-call type '__call__
03   (list
04    (symbol->py-string% 'C) (list->py-tuple% (list A B))
```

Since `type` is both a class and a callable object, what looks like the function call `type(...)` is in reality the *static method* call `type.__call__(...)`, which is what line 2 starts to invoke. The `__call__` method yields a new class object when given three arguments: a name, a tuple of parents, and a dictionary of fields and methods (or list of thunks ready to be converted into such a dictionary). Line 4 hands off the class name, C, and a tuple of base classes, A and B, as the first two parameters of the call. The third argument is a list of functions, each of which accept one argument, the created class, and returns a pair where the first item is the name of a class field or method and the second, its value. The entire rest of the program listing makes up this third argument.

```
05     (list
06      (lambda (this-class)
07        (list 'some_static_field
08              (number->py-number% 7)))
```

The need for wrapping each key-value pair around a function is shown in the result of compiling the next member field:

```
09      (lambda (this-class)
10        (list
11         'another_static_field
12         (let-values ([(some_static_field)
13                       (values (python-get-member this-class
14                                                   'some_static_field
15                                                   #f))])
16           (number->py-number% 3))))
```

Notice the `let-values` form wrapping the member field value. In Python, at class creation time, member fields (but not methods) have access to the previously created fields and methods. To emulate this, the Scheme code must contain the right bindings when creating the member field value, hence the need to always pass the class object to allow the extraction of currently bound values. As member methods do not have access to previously defined member variables, the following Scheme code for the method `m` lacks a `let-values`:

```
17      (lambda (this-class)
18        (list
19         'm
20         (procedure->py-function%
21          (opt-lambda (this x)
22            (call/ec (lambda (return1732)
23                       (return1732
```

```
                          (python-method-call
                          (python-get-attribute C
                                                      'some_static_field)
                          '__add__
                          (list x)))
                        py-none)))
            'm (list 'this 'x) null #f #f))))))))
```

Though a class definition is compiled as though it were the result of a function call being bound to a variable (the class name), it is not compiled as though it were any Python assignment statement such as `classname = type(classname, parents, members)`. It is currently treated as a simple special case of assignments (one target and no tuples), but in the future the class emitter might be removed in favor of translating class statements into assignment ASTs. We turn now to the process of generating Scheme code for those.

4.2.3 Variable Assignments

Identifiers are bound either at the top or function level. Imported modules' identifiers are bound at a different top level (see section 4.2.5).

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.

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

The first line becomes the following two Scheme lines:

```
(define rhs2320 (number->py-number% 1))
(define x rhs2320)
```

While it seems redundant to use an auxiliary variable (`rhs2320`), its need is exemplified in the translation of the second statement:

```
(define rhs2321 (number->py-number% 2))
(set! x rhs2321)
(define y rhs2321)
```

As x and y share the same value, the right-hand side must only be evaluated once. A similar strategy is followed for function variables, though as a current shortcoming of the compiler, they are all defined as `void` at the start of a function. For example, the following function uses a single variable, `x`.

```
def f():
    print "fn start"
    x = 1
```

Its body is translated into this Scheme equivalent (omitting the `call/ec` scaffolding):

```
(opt-lambda ()
  (let ([x (void)])
    (py-print #f
              (list (string->py-string% "fn start")))
    (let ([rhs1718 (number->py-number% 1)])
      (set! x rhs1718))
    py-none))
```

This does ensure that a runtime error is the result of using `x` before its definition, but it does not provide a good error message. 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 at the outer scope instead of defining a new one.

4.2.4 Function Application

A function is applied through `py-call` (Section 4.3.9). The function object expression 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)
```

4.2.5 Importing Modules

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 file and evaluates it in this new namespace. Finally, new bindings are introduced in the original namespace for the necessary values. 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. In Figure 8, squares represent both Python modules and Scheme namespaces, and the arrow represents the values to be copied from one namespace to another.

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 in this case, as shown by the following listings:

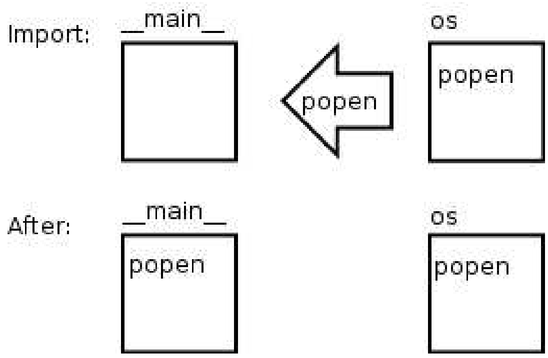
```
### module  b.py
x = 3

### module  a.py
from b import x
import b

print "b.x:", b.x
print "x:", x
x = 13
print "b.x", b.x
print "x", x

### output of running a.py
b.x: 3
x: 3
b.x 3
x 13
```

Figure 8: import popen from os



4.3 The Runtime System

The examples in the previous section have all made references to `py-print`, `procedure->py-function%`, `py-none`, and other runtime system library exports. This section explains the runtime system and the runtime libraries used throughout the generated Scheme program text.

The Python runtime system models the mechanics of the Python language. Python objects—called *python-nodes* in the runtime system for readability so as to not confuse them with the `object` class that is also seen at many places in the source code—have a type, a mutability specifier (so that you cannot, for example, change the contents of an immutable string), and a hash table (dictionary) of dynamic fields. The syntactic form *obj.attrib* refers, save a couple of exceptions, to the value associated with *attrib* in *obj*'s (or *obj*'s parent's) dictionary.

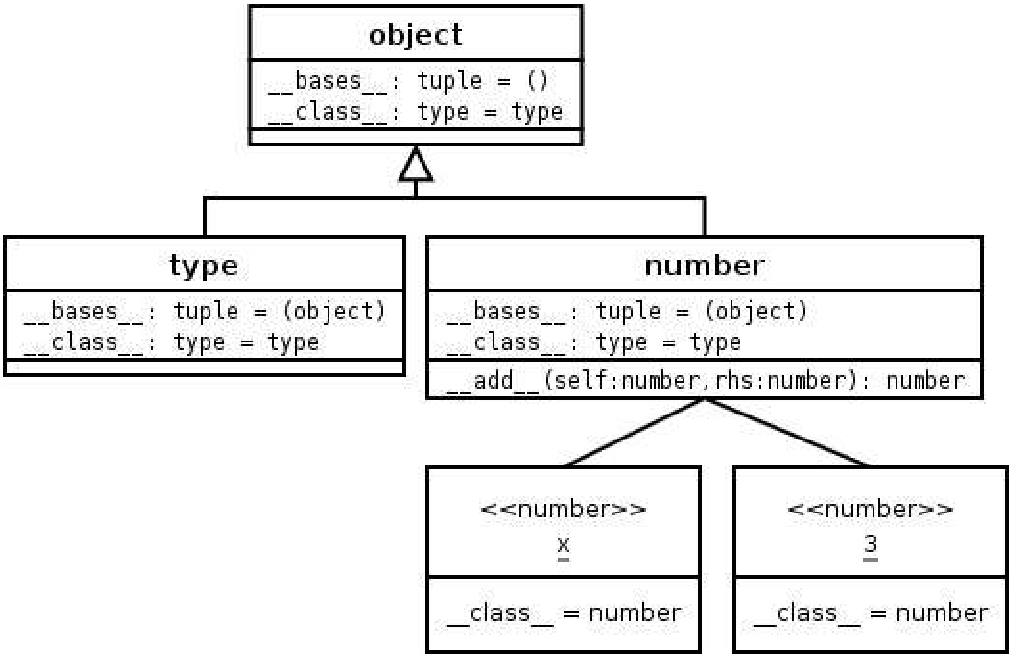
There are two special attributes not stored in the object dictionary:

1. `__dict__` points to the object's dictionary; and
2. `__class__` refers to the object's type.

Though an object's type is its `__class__` attribute, a class's type is always the built-in `type` object, and its parent classes are found in its `__bases__` attribute. Base types inherit from the built-in type `object` (Figure 9). This is the hierarchy in the Python programmer's mind when writing Python code.

Internally, all Python objects, types and instances alike, are represented by the `python-node` data structure (Figure 10). This is the

Figure 9: A simple Python class



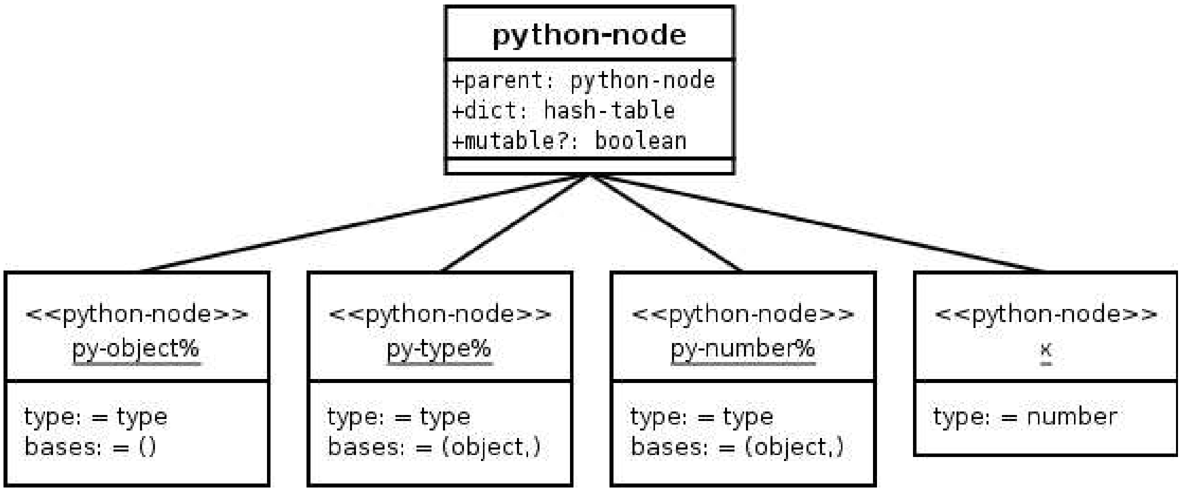
Scheme encoding of Python objects, and the hierarchy from Figure 9 is emulated by storing an object's type (in the Python sense) in a field (called `parent`) of the Scheme structure. The convention adopted by the runtime system has been to name each Python type `py-`*name*%, where *name* is the Python type (e.g., the Python `object` type is named `py-object%` in the runtime system).

With these concepts in mind, let us now explore a set of important functions from the runtime system.

4.3.1 py-print

`py-print` accepts a list of objects to print and an output device object (one that has a *write* method). If the output object is false, the current output port is used. The string stream sent to the output object consists of each item from the list of objects, each converted to its string representation, with a space character in between each one.

Figure 10: Internal representation of Python objects



4.3.2 number->py-number%

Wraps a Scheme number as a Python number object.

4.3.3 string->py-string%

This function converts a Scheme string into a Python string.

4.3.4 procedure->py-function%

This function accepts a Scheme procedure, a function name symbol, a list of positional argument names, a list of optional argument names, a rest-argument name (or false), and a dictionary argument name (or false). It returns a Python function object, which is internally a `python-node` with its parent field set to `py-function%` (another `python-node` object, this one representing the Python `function` type object).

This function is used by the runtime system to wrap internal functions (such as the built-in *repr*) as Python functions and by the compiler in the generated Scheme code for function definitions.

4.3.5 python-get-member

This function takes a Python object, an attribute name (as a symbol), and an optional flag. It extracts and returns the value associated with the given attribute name in the object's dictionary. If no such attribute is found, the object's parent classes are searched in the same manner. Once there are no more parent classes to search, the Python `AttributeError` exception is raised.

When the optional flag is set to `true`, `py-function%` objects are wrapped inside `py-method%` objects, which, as per the Python language specification, contain attributes referring to the wrapped function, owner class, and possibly owner object. This allows the Python programmer to assign `some_object.some_method` to a variable `m`, which, when applied, will invoke `some_method` with `some_object` as its first argument. The flag's default value is `true`.

4.3.6 python-get-attribute

This function looks up an object's attribute through the use of the object type's `__getattrib__` method. The default implementation of `__getattrib__` is `python-get-member` (4.3.5).

4.3.7 python-set-member!

Sets a specified attribute in the object's dictionary to the given value.

4.3.8 python-method-call

This function accepts a Python object, an attribute name, and a list of arguments. It fetches the method named by the attribute argument and applies it, passing the object argument as the first argument before the supplied argument list. It is a shorthand for calling `py-call` without explicitly fetching a method from an object.

4.3.9 py-call

This function accepts two arguments: a callable object and list of Python values. The callable object is applied with those values as its arguments, and this application's result is returned as the result of `py-call`. A callable object is either a Python `function` object, a Python object implementing a `__call__` method, or a Scheme procedure. When the input is an object implementing a `__call__` method, that method is extracted as a Python function object and passed to a recursive call to `py-call`. When the input is a Python function object, it is converted to a Scheme procedure and passed to a recursive call to `py-call`. When the input is a Scheme procedure, it is applied.

5 Future Work

In order to complete the compiler, a mechanism must be devised to dynamically load (through the `import` statement, section 3.2.16) low-level Python libraries written in C, including the core library from CPython. This matter is currently under investigation, as are the following:

- completion of an automated test suite;
- generation of better error messages;
- implementation of ⟨*yield_stmt*⟩ (Section 3.2.15);
- implementation of ⟨*exec_stmt*⟩ (Section 3.2.23); and
- handling of default function parameters (Section 4.2.1).

6 Conclusion

The Python-to-Scheme compiler translates Python programs into their Scheme equivalents. The lexer and parser read and check the syntactic correctess of such programs. The code generator turns the parser's output of syntactic elements into Scheme code. The generated Scheme code uses the runtime system, which models the Python environment. This allows Python programmers to take advantage of DrScheme and its development tools while writing in their preferred language. The benefit to Scheme programmers comes by way of Python libraries, which are increasingly numerous.

Contents

References

[1] G. van Rossum, *Python Reference Manual.* PythonLabs, February 2003.

[2] G. van Rossum, "The Python interpreter." PythonLabs, February 2003. Version 2.3a2.

[3] S. Pedroni and N. Rappin, *Jython Essentials.* O'Reilly, March 2002.

[4] M. Flatt, "The MzScheme interpreter." PLT, December 2002. Version 203.

[5] M. Flatt, *PLT MzScheme: Language Manual.* PLT, December 2002.

[6] R. Kelsey, W. Clinger, and J. R. (Editors), "Revised^5 report on the algorithmic language Scheme," *ACM SIGPLAN Notices*, vol. 33, no. 9, pp. 26–76, 1998.

[7] M. Flatt, *PLT MrEd: Graphical Toolbox Manual.* PLT, December 2002.

[8] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen, "DrScheme: A Pedagogic Programming Environment for Scheme," *Programming Languages: Implementations, Logics, and Program s*, vol. 1292, pp. 369–388, September 1997.

[9] P. Pinheiro da Silva and N. W. Paton, "UMLi: The unified modeling language for interactive applications," in *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings* (A. Evans, S. Kent, and B. Selic, eds.), vol. 1939, pp. 117–132, Springer, 2000.