CHAPTER 6

# Objects

The main ideas for this chapter are:

**objects and classes:** objects are values that bundle together some data (attributes) and some functions (methods). Classes are values that describe how to create objects.

**attributes and methods:** Both objects and classes can contain attributes and methods. An attribute maps a name to a value and a method maps a name to a function.

**inheritance:** One class may inherit from one or more other classes, thereby gaining access to the methods in the inherited classes.

## 6.1. Syntax of $P_3$

The concrete syntax of $P_3$ is shown in Figure 1 and the abstract syntax (the Python AST classes) is shown in Figure 2.

```
expression ::= expression "." identifier
expression_list ::= expression ( "," expression )* [","]
statement ::= "class" name ["(" expression_list ")"] ":" suite
            | "if" expression ":" suite "else" ":" suite
            | "while" expression ":" suite
target ::= expression "." identifier
```

FIGURE 1. Concrete syntax for the $P_3$ subset of Python. (In addition to that of $P_2$.)

## 6.2. Semantics of $P_3$

This week we add a statement for creating classes. For example, the following statement creates a class named `C`.

```
>>> class C:
...     x = 42
```

Assignments in the body of a class create *class attributes*. The above code creates a class `C` with an attribute `x`. Class attributes may be accessed using the dot operator. For example:

```
class AssAttr(Node):
    def __init__(self, expr, attrname, flags):
        self.expr = expr
        self.attrname = attrname
        self.flags = flags          # ignore this

class Class(Node):
    def __init__(self, name, bases, doc, code):
        self.name = name
        self.bases = bases
        self.doc = doc              # ignore this
        self.code = code

class Getattr(Node):
    def __init__(self, expr, attrname):
        self.expr = expr
        self.attrname = attrname

class If(Node):
    def __init__(self, tests, else_):
        self.tests = tests
        self.else_ = else_

class While(Node):
    def __init__(self, test, body, else_):
        self.test = test
        self.body = body
        self.else_ = else_
```

FIGURE 2. The Python classes for $P_3$ ASTs.

```
>>> print C.x
42
```

The body of a class may include arbitrary statements, including statements that perform I/O. These statements are executed as the class is created.

```
>>> class C:
...     print 4 * 10 + 2
42
```

If a class attribute is a function, then accessing the attribute produces an *unbound method*.

```
>>> class C:
        f = lambda o, dx: o.x + dx
```

```
>>> C.f
<unbound method C.<lambda>>
```

An unbound method is like a function except that the first argument must be an instance of the class from which the method came. We'll talk more about instances and methods later.

Classes are first-class objects, and may be assigned to variables, returned from functions, etc. The following `if` expression evaluates to the class `C`, so the attribute reference evaluates to `42`.

```
>>> class C:
...     x = 42
>>> class D:
...     x = 0

>>> print (C if True else D).x
42
```

**6.2.1. Inheritance.** A class may inherit from other classes. In the following, class `C` inherits from classes `A` and `B`. When you reference an attribute in a derived class, if the attribute is not in the derived class, then the base classes are searched in depth-first, left-to-right order. In the following, `C.x` resolves to `A.x` (and not `B.x`) whereas `C.y` resolves to `B.y`.

```
>>> class A:
...     x = 4

>>> class B:
...     x = 0
...     y = 2

>>> class C(A, B):
...     z = 3

>>> print C.x * 10 + C.y
42
```

**6.2.2. Objects.** An object (or *instance*) is created by calling a class as if it were a function.

```
o = C()
```

If the class has an attribute named `__init__`, then once the object is allocated, the `__init__` function is called with the object as it's first argument. If there were arguments in the call to the class, then these arguments are also passed to the `__init__` function.

```
>>> class C:
...     def __init__(o, n):
...         print n

>>> o = C(42)
42
```

An instance may have associated *data attributes*, which are created by assigning to the attribute. Data attributes are accessed with the dot operator.

```
>>> o.x = 7
>>> print o.x
7
```

Different objects may have different values for the same attribute.

```
>>> p = C(42)
42
>>> p.x = 10
>>> print o.x, p.x
7, 10
```

Objects live on the heap and may be aliased (like lists and dictionaries).

```
>>> print o is p
False
>>> q = o
>>> print q is o
True
>>> q.x = 1
>>> print o.x
1
```

A data attribute may be a function (because functions are first class). Such a data attribute is not a method (the object is not passed as the first parameter).

```
>>> o.f = lambda n: n * n
>>> o.f(3)
9
```

When the dot operator is applied to an object but the specified attribute is not present in the object itself, the class of the object is searched followed by the base classes in depth-first, left-to-right order.

```
>>> class C:
...     y = 3
```

```
>>> o = C()
>>> print o.y
3
```

If an attribute reference resolves to a function in the class or base class of an object, then the result is a *bound method*.

```
>>> class C:
...     def move(o,dx):
...         o.x = o.x + dx
>>> o = C()
>>> o.move
<bound method C.move of <__main__.C instance at 0x11d3fd0>>
```

A bound method ties together the receiver object (`o` in the above example) with the function from the class (`move`). A bound method can be called like a function, where the receiver object is implicitly the first argument and the arguments provided at the call are the rest of the arguments.

```
>>> o.x = 40
>>> o.move(2)
>>> print o.x
42
```

Just like everything else in Python, bound methods are first class and may be stored in lists, passed as arguments to functions, etc.

```
>>> mlist = [o.move,o.move,o.move]
>>> i = 0
>>> while i != 3:
...     mlist[i](1)
...     i = i + 1
>>> print o.x
45
```

You might wonder how the Python implementation knows whether to make a normal function call or whether to perform a method call (which requires passing the receiver object as the first argument). The answer is that the implementation checks the type tag in the operator to see whether it is a function or bound method and then treats the two differently.

EXERCISE 6.1. Read:

(1) Section 9 of the Python Tutorial
(2) Python Language Reference, Section 3.2

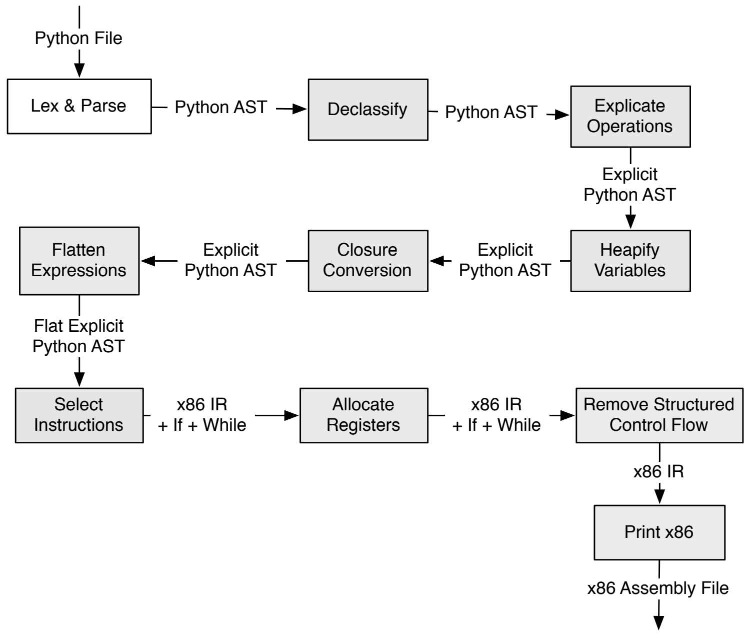


FIGURE 3. Structure of the compiler.

**6.2.3. If and While Statements.** This chapter we also add `if` and `while` statements. For the `if` statement, you don't need to support `elif` and you can assume that every `if` has an `else`. For `while` statements, you don't need to support the `else` clause.

One of the more interesting aspects of extending your compiler to handle `While` statements is that you'll need to figure out how to propagate the live-variable information through `While` statements in the register allocation phase.

## 6.3. Compiling Classes and Objects

Figure 3 shows the structure of the compiler with the addition of classes and objects. We insert a new pass at the beginning of the compiler that lowers classes and objects to more primitive operations and then we update the rest of the compiler to handle these new primitives.

In addition to the new passes and primitives, the entities introduced this week are all first-class, so the `big_pyobj` union in `runtime.h` has been extended.

**class:** The runtime representation for a class stores a list of base classes and a dictionary of attributes.

**object:** The runtime representation for an object stores its class and a dictionary of attributes.

**unbound method:** The runtime representation of an unbound method contains the underlying function and the class object on which the attribute access was applied that created the unbound method.

**bound method:** The runtime representation for a bound method includes the function and the receiver object.

The following are the new functions in `runtime.h` for working with classes, objects, bound methods, and unbound methods.

```
 /* bases should be a list of classes */
big_pyobj* create_class(pyobj bases);
big_pyobj* create_object(pyobj cl);
 /* inherits returns true if class c1 inherits from class c2 */
int inherits(pyobj c1, pyobj c2);
 /* get_class returns the class from an object or unbound method */
big_pyobj* get_class(pyobj o);
 /* get_receiver returns the receiver from inside a bound method */
big_pyobj* get_receiver(pyobj o);
 /* get_function returns the function from inside a method */
big_pyobj* get_function(pyobj o);
int has_attr(pyobj o, char* attr);
pyobj get_attr(pyobj c, char* attr);
pyobj set_attr(pyobj obj, char* attr, pyobj val);
```

**6.3.1. Compiling empty class definitions and class attributes.** Compiling full class definitions is somewhat involved, so I first recommend compiling empty class definitions. We begin with class definitions that have a trivial body.

```
>>> class C:
...     0
```

The class definition should be compiled into an assignment to a variable named `C`. The right-hand-side of the assignment should be an expression that allocates a class object with an empty hashtable for attributes and an empty list of base classes. So, in general, the transformation should be

```
class C:
  0
⟹
C = create_class()
```

where `create_class` is a new C function in `runtime.h`.

While a class with no attributes is useless in C++, in Python you can add attributes to the class after the fact. For example, we can proceed to write

```
>>> C.x = 3
>>> print C.x
3
```

An assignment such as `C.x = 3` (the `AssAttr` node) should be transformed into a call to `set_attr`. In this example, we would have `set_attr(C, "x", 3)`. Note that this requires adding support for string constants to your intermediate language.

The attribute access `C.x` (the `Getattr` node) in the `print` statement should be translated into a call to the `get_attr` function in `runtime.h`. In this case, we would have `get_attr(C, "x")`.

**6.3.2. Compiling class definitions.** A class body may contain an arbitrary sequence of statements, and some of those statements (assignments and function definitions) add attributes to the class object. Consider the following example.

```
class C:
    x = 3
    if True:
        def foo(self, y):
            w = 3
            return y + w
        z = x + 9
    else:
        def foo(self, y):
            return self.x + y
    print 'hello world!\n'
```

This class definition creates a class object with three attributes: `x`, `foo`, and `z`, and prints out `hello world!`.

The main trick to compiling the body of a class is to replace assignments and function definitions so that they refer to attributes in the class. The replacement needs to go inside compound statements such as `If` and `While`, but not inside function bodies, as those assignments correspond to local variables of the function. One can imagine transforming the above code to something like the following:

```
class C:
    pass
C.x = 3
if True:
    def __foo(self, y):
        w = 3
        return y + w
    C.foo = __foo
    C.z = C.x + 9
else:
    def __foo(self, y):
        return self.x + y
    C.foo = __foo
print 'hello world!\n'
```

Once the code is transformed as above, the rest of the compilation passes can be applied to it as usual.

In general, the translation for class definitions is as follows.

```
class $C(B_1, \ldots, B_n)$:
  $body$
$\Longrightarrow$
$tmp$ = create_class([$B_1, \ldots, B_n$])
$newbody$
$C$ = $tmp$
```

Instead of assigning the class to variable $C$, we instead assign it to a unique temporary variable and then assign it to $C$ after the $newbody$. The reason for this is that the scope of the class name $C$ does not include the body of the class.

The $body$ is translated to $newbody$ by recursively applying the following transformations. You will need to know which variables are assigned to (which variables are class attributes), so before transforming the $body$, first find all the variables assigned-to in the $body$ (but not assigned to inside functions in the $body$).

The translation for assignments is:

```
$x$ = $e$
$\Longrightarrow$
set_attr($tmp$, "$x$", $e'$)
```

where $e'$ is the recursively processed version of $e$.

The translation for variables is somewhat subtle. If the variable is one of the variables assigned somewhere in the body of this class, and if the variable is also in scope immediately outside the class, then translate the variable into a conditional expression that either does an attribute access or a variable access depending on whether the attribute is actually present in the class value.

```
$x$
$\Longrightarrow$
get_attr($tmp$, "$x$") if has_attr($tmp$, "$x$") else $x$
```

If the variable is assigned in the body of this class but is not in scope outside the class, then just translate the variable to an attribute access.

```
x
⟹
get_attr(tmp, "x")
```

If the variable is not assigned in the body of this class, then leave it as a variable.

```
x
⟹
x
```

The translation for function definitions is:

```
def f(e1,...,en):
    body
⟹
def f_tmp(e1,...,en):
    body            # the body is unchanged, class attributes are not in scope here
set_attr(tmp, "f", f_tmp)
```

**6.3.3. Compiling objects.** The first step in compiling objects is to implement object construction, which in Python is provided by invoking a class as if it were a function. For example, the following creates an instance of the C class.

```
C()
```

In the AST, this is just represented as a function call (CallFunc) node. Furthermore, in general, at the call site you won't know at compile-time that the operator is a class object. For example, the following program might create an instance of class C or it might call the function foo.

```
def foo():
    print 'hello world\n'

(C if input() else foo)()
```

This can be handled with a small change to how you compile function calls. You will need to add a conditional expression that checks whether the operator is a class object or a function. If it is a class object, you need to allocate an instance of the class. If the class defines an __init__ method, the method should be called immediately after the object is allocated. If the operator is not a class, then perform a function call.

In the following we describe the translation of function calls. The Python `IfExp` is normally written as $e_1$ `if` $e_0$ `else` $e_2$ where $e_0$ is the condition, $e_1$ is evaluated if $e_0$ is true, and $e_2$ is evaluated if $e_0$ is false. I'll instead use the following textual representation:

```
if $e_0$ then $e_1$ else $e_2$
```

In general, function calls can now be compiled like this:

```
$e_0$($e_1$,...,$e_n$)
$\Longrightarrow$
let $f$ = $e_0$ in
let $a_1$ = $e_1$ in
  ⋮
let $a_n$ = $e_n$ in
if is_class($f$) then
    let $o$ = create_object($f$) in
    if has_attr($f$, '__init__') then
        let $ini$ = get_function(get_attr($f$, '__init__')) in
        let _ = $ini$($o$, $a_1$,...,$a_n$) in
          $o$
    else $o$
else
    $f$($a_1$,...,$a_n$)          # normal function call
```

The next step is to add support for creating and accessing attributes of an object. Consider the following example.

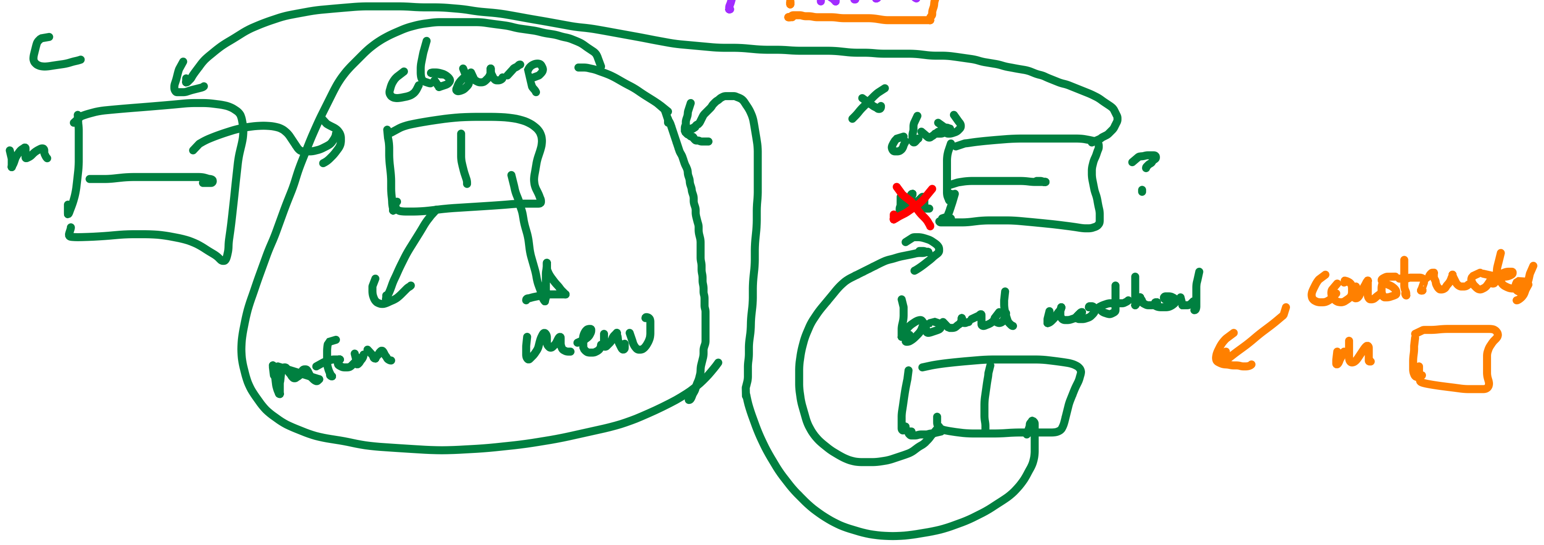
```
o = C()
o.w = 42

print o.w
print o.x      # attribute from the class C
```

An assignment to an attribute should be translated to a call to `set_attr` and accessing an attribute should be translated to a call to `get_attr`.

**6.3.4. Compiling bound and unbound method calls.** A call to a bound or unbound method also shows up as a function call node (`CallFunc`) in the AST, so we now have four things that can happen at a function call (we already had object construction and normal function calls). To handle bound and unbound methods, we just need to add more conditions to check whether the operator is a bound or unbound method. In the case of an unbound method, you should call the underlying function from inside the method. In the case of a bound method, you call the underlying function, passing

```
class C:                         class C:
    def m(...):  →                   m = lambda ...
        ≡
```

```
x = C()
~~x.m = ...~~
y = x.m
```



the receiver object (obtained from inside the bound method) as the first argument followed by the normal arguments. The suggested translation for function calls is given below.

```
$e_0$($e_1$,...,$e_n$)
$\Longrightarrow$
let $f$ = $e_0$ in
let $a_1$ = $e_1$ in
  ⋮
let $a_n$ = $e_n$ in
if is_class($f$) then
    let $o$ = create_object($f$) in
    if has_attr($f$, '__init__') then
        let $ini$ = get_function(get_attr($f$, '__init__')) in
        let _ = $ini$($o$, $a_1$,...,$a_n$) in
          $o$
    else $o$
else
    if is_bound_method($f$) then
        get_function($f$)(get_receiver($f$), $a_1$,...,$a_n$)
    else (get_closure)
        if is_unbound_method($f$) then
            get_function($f$)($a_1$,...,$a_n$)
        else
            $f$($a_1$,...,$a_n$)          # normal function call
```

methods
$\downarrow$ HW6
closures
$\downarrow$ HW5
procedures

EXERCISE 6.2. Extend your compiler to handle $P_3$. You do not need to implement operator overloading for objects or any of the special attributes or methods such as `__dict__`.

# Appendix

## 6.4. x86 Instruction Reference

Table 1 lists some x86 instructions and what they do. Address offsets are given in bytes. The instruction arguments $A, B, C$ can be immediate constants (such as $\$4$), registers (such as $\%eax$), or memory references (such as $-4(\%ebp)$). Most x86 instructions only allow at most one memory reference per instruction. Other operands must be immediates or registers.

| **Instruction** | **Operation** |
|---|---|
| `addl` $A$, $B$ | $A + B \rightarrow B$ |
| `call` $L$ | Pushes the return address and jumps to label $L$ |
| `call` *$A$ | Calls the function at the address $A$. |
| `cmpl` $A$, $B$ | compare $A$ and $B$ and set flag |
| `je` $L$ | If the flag is set to "equal", jump to label $L$ |
| `jmp` $L$ | Jump to label $L$ |
| `leave` | `ebp` $\rightarrow$ `esp`; `popl %ebp` |
| `movl` $A$, $B$ | $A \rightarrow B$ |
| `movzbl` $A$, $B$ | $A \rightarrow B$ |
| | where $A$ is a single-byte register (e.g., `al` or `cl`), $B$ is a four-byte register, and the extra bytes of $B$ are set to zero |
| `negl` $A$ | $-A \rightarrow A$ |
| `notl` $A$ | $\sim A \rightarrow A$ (bitwise complement) |
| `orl` $A$, $B$ | $A \vert B \rightarrow B$ (bitwise-or) |
| `andl` $A$, $B$ | $A \& B \rightarrow B$ (bitwise-and) |
| `popl` $A$ | $*$`esp` $\rightarrow A$; `esp` $+ 4 \rightarrow$ `esp` |
| `pushl` $A$ | `esp` $- 4 \rightarrow$ `esp`; $A \rightarrow *$`esp` |
| `ret` | Pops the return address and jumps to it |
| `sall` $A$, $B$ | $B$ << $A \rightarrow B$ (where $A$ is a constant) |
| `sarl` $A$, $B$ | $B$ >> $A \rightarrow B$ (where $A$ is a constant) |
| `sete` $A$ | If the flag is set to "equal", then $1 \rightarrow A$, else $0 \rightarrow A$. |
| | $A$ must be a single byte register (e.g., `al` or `cl`). |
| `setne` $A$ | If the flag is set to "not equal", then $1 \rightarrow A$, else $0 \rightarrow A$. |
| | $A$ must be a single byte register (e.g., `al` or `cl`). |
| `subl` $A$, $B$ | $B - A \rightarrow B$ |

TABLE 1. Some x86 instructions. We write $A \rightarrow B$ to mean that the value of $A$ is written into location $B$.

# Bibliography

[1] V. K. Balakrishnan. *Introductory Discrete Mathematics*. Dover Publications, Incorporated, 1996.

[2] D. Beazley. PLY (Python Lex-Yacc). `http://www.dabeaz.com/ply/`.

[3] D. Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979.

[4] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, 1992.

[5] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–105. ACM Press, 1982.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[7] A. H. Gebremedhin. *Parallel Graph Coloring*. PhD thesis, University of Bergen, 1999.

[8] S. Hack and G. Goos. Optimal register allocation for ssa-form programs in polynomial time. *Information Processing Letters*, 98(4):150 – 155, 2006.

[9] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, November 2006.

[10] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*, November 2006.

[11] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, November 2006.

[12] S. C. Johnson. Yacc: Yet another compiler-compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, 1979.

[13] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[14] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical report, Bell Laboratories, July 1975.

[15] H. A. Omari, K. E. nbsp;Sabri Hussein A. Omari, K. E. nbsp;Sabri Hussein A. Omari, K. E. nbsp;Sabri Hussein A. Omari, and K. E. Sabri. New graph coloring algorithms. *Journal of Mathematics and Statistics*, 2(4), 2006.

[16] J. Palsberg. Register allocation via coloring of chordal graphs. In *CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 3–3, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[17] K. H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 2002.

[18] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[19] G. van Rossum. *Python Library Reference*. Python Software Foundation, 2.5 edition, September 2006.

[20] G. van Rossum. *Python Reference Manual*. Python Software Foundation, 2.5 edition, September 2006.

[21] G. van Rossum. *Python Tutorial*. Python Software Foundation, 2.5 edition, September 2006.

[22] O. Waddell and R. K. Dybig. Fast and effective procedure inlining. In *Proceedings of the 4th International Symposium on Static Analysis*, SAS '97, pages 35–52, London, UK, 1997. Springer-Verlag.