NPRG045 Project Specification: OPythn

Marcel Goh

March 13, 2019

1 Target

The OPythn project aims to implement a working subset of the Python programming language by means of a bytecode compiler and interpreter, written in OCaml. Users will be able to interact with OPythn via a top-level read-eval-print loop or compile OPythn source code to bytecode. OPythn will run on Unix operating systems, including macOS and Ubuntu.

2 Language

The core of OPythn is designed to be lightweight and minimal. Basic types and operations, control structures, and elementary data structures are included, while more complex Python constructions, such as anonymous functions, list comprehensions, generators, and coroutines are omitted.

2.1 Features

OPythn inherits the following features directly from Python:

- Primitive types `int`, `float`, `str`, and `bool`
- Arithmetic and boolean operators
- Control structures `if`, `for`, and `while`
- Lists, tuples, and dictionaries
- Named functions
- Classes and objects

OPythn integers can be between -4,611,686,018,427,387,904 and -4,611,686,018,427,387,904 inclusive. Ordinary arithmetic operators for addition, multiplication, subtraction, division, modulus, and exponentiation are supported between types `int` and `float`. Additionally, OPythn supports the bitwise operations `|`, ˆ, `&`, <<, >>, and ˜ on integers. Boolean values can be compared using the operators <, <=, >, >=, ==, `!=`, `is`, and `is not`. Any object can be tested for truth value. The objects `None`, `False`, `0`, `0.0`, `''`, `[]`, and `{}` are considered false; any other object is considered true.

OPythn lists are zero-indexed resizing arrays and accessing an element takes constant time. Slice operations are supported. For example, if `a = [1,2,3,4,5]`, then `a[1:3]` would give the list `[2,3]`. However, the Python `slice` function (and associated object) is not supported.

OPythn supports higher-order functions and nested functions. However, nested classes and multiple inheritance are not supported. Exceptions are also not supported in Python. When an error occurs, the program terminates/returns to the REPL with an error message and a stack trace is printed. The following functions comprise OPythn's standard library and are included in a normal installation. Some of these functions are implemented in OPythn itself while others are implemented in OCaml.

`abs()`	`bin()`	`bool()`	`chr()`	`divmod()`	`enumerate()`
`filter()`	`float()`	`hash()`	`hex()`	`input()`	`int()`
`isinstance()`	`issubclass()`	`iter()`	`len()`	`map()`	`max()`
`min()`	`next()`	`oct()`	`open()`	`ord()`	`pow()`
`print()`	`range()`	`repr()`	`reversed()`	`round()`	`sorted()`
`str()`	`sum()`	`type()`			

2.2 Lexical Conventions

The following are OPythn keywords and cannot be used in ordinary names:

int	float	str	bool	def	return
True	False	None	and	or	not
if	elif	else	for	in	while
break	continue	class	is	del	import
in	from	as			

The # character can be used to indicate comments in source code. Any letters from # to the end of the line will be ignored. The rules that the lexer uses to recognise identifiers, numbers, and special characters are defined by the following regular expressions:

```
ENDMARKER: '\Z'
NAME: '[^\d\W]\w*'
NEWLINE: '\n'
NUMBER: '[+-]?((\d+\.\d*|\d*\.\d+)'
STRING: '(\"[^\n\"\\]\")|(\'[^\n\'\\]\')'
```

2.3 Representation

OPythn primitive types will be represented in OCaml with a datatype definition.

```
type py_prim =
  INT of int
| FLOAT of float
| STR of string
| BOOL of bool
```

We can have a separate type for OPythn compound types and then a general `py_val` type to capture both of these possibilities. OPythn lists can have elements of different types, making `py_val array` a natural representation. To represent a dictionary, we use an OCaml hashtable.

```
type py_comp =
  LIST of py_val array
| DICT of (py_prim, py_val) Hashtbl.t
and type py_val =
  PRIM of py_prim
| COMP of py_comp
```

2.4 Grammar

This is the complete OPythn grammar specification:

```
# Start symbols
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER

funcdef: 'def' NAME parameters ':' suite
classdef: 'class' NAME ['(' [arglist] ')'] ':' suite
parameters: '(' [arglist] ')'
```

```
varargslist: arg (',' arg)*
arg: NAME

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | flow_stmt | import_stmt)
expr_stmt:
compound_stmt: if_stmt | while_stmt | for_stmt | funcdef |
    classdef

# Control structures
if_stmt: 'if' test ':' suite ('elif' test ':' suite)*
    ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite
    ['else' ':' suite]

# Tests and expressions
test: or_test ['if' or_test 'else' test]
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: '<'|'>'|'=='|'>='|'<='|'!='|'in'|'not' 'in'|
    'is'|'is' 'not'
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom_expr ['**' factor]
atom_expr: atom trailer*
atom: (NAME | NUMBER | STRING+ | '...' | 'None' |
    'True' | 'False')
trailer: '(' [arglist] ')' | '[' subscript ']' | . NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: expr (',' expr)* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( ((test ':' test | '**' expr)
                  ((',' (test ':' test)* [','])) |
                 (test (',' test)* [','])) )
arglist: argument (',' argument)* [',']

argument (test | test '=' test)
```

3 Implementation

Both the OPythn bytecode compiler and interpreter will be implemented in OCaml. This section gives a rough overview of the process by which OPythn source code is handled and executed.

3.1 Input/Output

The OPythn program can be run directly in a Unix terminal via the command `opythn`. When run without arguments, this command launches a read-eval-print loop into which the user can enter commands. In interactive mode, the interpreter evaluates a valid command as soon as it is received:

```
OPythn (Interactive), version 0.0.0
:? for help, :q to quit
]=> print("Hello, world!")
Hello, world!
```

Alternatively, OPythn source code can be defined in a separate `.opy` file and provided to the interpreter as a program argument. In this, mode, OPythn immediately compiles and interprets the source code directly.

3.2 Lexical Analysis

Upon reading source code, the OPythn front-end passes the data as a string to a series of functions that lex the code into tokens. The lexer will be created with the help of `ocamllex`, a program that generates a deterministic finite automaton in OCaml. This resulting lexer matches regular expressions in the string to convert chunks of characters into the correct tokens.

3.3 Parsing

The tokens produced by the lexer is then fed into the parser, which will be implemented according to the grammar rules outlined in Section 2.3 and using `ocamlyacc`, an OCaml parser generator. The result will be an abstract syntax tree that represents the structure and semantics of the OPythn program.

3.4 Bytecode

OPythn evaluates an abstract syntax tree by generating intermediary bytecode, which is evaluated by a virtual stack machine, as in many conventional Python implementations. As an example of what this looks like, consider this simple algorithm for integer exponentiation:

```
x = int(input("Enter the multiplicand: "))
y = int(input("Enter the multiplier: "))

acc = 0
while y > 0:
    if y % 2 == 0:
        x *= 2
        y //= 2
    else:
        acc += x
        y -= 1
```

```
print("The product is: ", acc)
```

The disassembled output produced by the CPython interpreter looks like this:

```
  0 LOAD_NAME                0 (int)
  2 LOAD_NAME                1 (input)
  4 LOAD_CONST               0 ('Enter the multiplicand: ')
  6 CALL_FUNCTION            1
  8 CALL_FUNCTION            1
 10 STORE_NAME               2 (x)

 12 LOAD_NAME                0 (int)
 14 LOAD_NAME                1 (input)
 16 LOAD_CONST               1 ('Enter the multiplier: ')
 18 CALL_FUNCTION            1
 20 CALL_FUNCTION            1
 22 STORE_NAME               3 (y)

 24 LOAD_CONST               2 (0)
 26 STORE_NAME               4 (acc)

 28 SETUP_LOOP              58 (to 88)
 30 LOAD_NAME                3 (y)
 32 LOAD_CONST               2 (0)
 34 COMPARE_OP               4 (>)
 36 POP_JUMP_IF_FALSE       86

 38 LOAD_NAME                3 (y)
 40 LOAD_CONST               3 (2)
 42 BINARY_MODULO
 44 LOAD_CONST               2 (0)
 46 COMPARE_OP               2 (==)
 48 POP_JUMP_IF_FALSE       68

 50 LOAD_NAME                2 (x)
 52 LOAD_CONST               3 (2)
 54 INPLACE_MULTIPLY
 56 STORE_NAME               2 (x)

 58 LOAD_NAME                3 (y)
 60 LOAD_CONST               3 (2)
 62 INPLACE_FLOOR_DIVIDE
 64 STORE_NAME               3 (y)
 66 JUMP_ABSOLUTE           30

 68 LOAD_NAME                4 (acc)
 70 LOAD_NAME                2 (x)
 72 INPLACE_ADD
```

```
     74 STORE_NAME                 4 (acc)

     76 LOAD_NAME                  3 (y)
     78 LOAD_CONST                 4 (1)
     80 INPLACE_SUBTRACT
     82 STORE_NAME                 3 (y)
     84 JUMP_ABSOLUTE             30
     86 POP_BLOCK

     88 LOAD_NAME                  5 (print)
     90 LOAD_CONST                 5 ('The product is: ')
     92 LOAD_NAME                  4 (acc)
     94 CALL_FUNCTION              2
     96 POP_TOP
     98 LOAD_CONST                 6 (None)
    100 RETURN_VALUE
```

OPythn compiled source will look largely the same, but will not be converted into bitstrings. Instead, an `instr` datatype will be used to capture the same information as CPython's two-byte instructions. A different type tag will be used for each bytecode instruction.

```
type instr =
  LOAD_NAME of int
| LOAD_CONST of int
(* ... *)
| RETURN_VALUE
```

The advantage of this representation is that the interpreter can easily pattern-match on instruction tags. When expressed in this way, the above bytecode would be represented as the following `instr array`.

```
[| LOAD_NAME 0;
   LOAD_NAME 1;
   LOAD_CONST 0;
   CALL_FUNCTION 1;
   CALL_FUNCTION 1;
   STORE_NAME 2;
   (* ... *)
   RETURN_VALUE |]
```

3.5 Virtual Machine

Finally, the bytecode is interpreted by a stack-based virtual machine that produces the desired output. We can define a datatype specifically for stack items, so that any sort of data can be at the top of the stack. Then the stack can be represented as an `item list`, because we will only ever have to access the first few elements in the (linked) list. Many bytecode instructions access and manipulate the stack directly. For example, the instruction `POP_TOP` removes the top element of the stack.

Additionally, the interpreter has access to an array of names and and array of constants, both of which were created upon code generation. For example, the instruction `LOAD_NAME 1` accesses the name array at index 1 and pushes the associated object onto the stack; in our case above, this was the function `input`. Likewise, the instruction `LOAD_CONST 5` accesses the constant array at the index 5 and pushes its value onto the stack.

Control-flow logic is implemented using the stack using certain instructions that tell the interpreter to jump to other parts of the bytecode. For example, at line 46 of the example bytecode, the top two elements on the stack are the values `0` and `y % 2`. The `COMPARE_OP 2` instruction checks if the top two elements of the stack are equal, and then pushes a boolean value onto the stack. The next instruction, `POP_JUMP_IF_FALSE 68`, pops this boolean off the stack and jumps to line 68 if the boolean was false. Loops are implemented in a similar way. The instruction `SETUP_LOOP` n designates the next n instructions as a block, and a test has to be run every loop to determine if the block should be exited via a `JUMP` instruction.

The evaluation procedure is performed by a recursive function `eval : int -> item list -> int`, in a context where the instruction, name, and constant arrays are defined. The `eval` function has as its inputs a line number and the current stack, and each time it is called, it performs the appropriate stack manipulations before calling itself again with the next line to evaluate. When `eval` runs out of instructions to evaluate (i.e. reaches the last instruction in the array with no `JUMP` command), the program halts and an exit code is returned.

References

[1] Guido van Rossum. *The Python Language Reference*. Python Software Foundation, 2019.