NPRG045 Project Specification: OPythn

Marcel Goh

March 8, 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 and dictionaries
- Named functions
- Classes and objects

To round out OPythn's standard library, other core functions will be defined in OPythn and included in the standard OPythn installation. These mainly consist of basic operations on mathematical objects, lists, and strings.

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 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 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] ')'
vararglist: 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 will then be fed into a parser, which will be created according to the grammar rules outlined in Section 2.3 and using `ocamlyacc`. The result will be an abstract syntax tree that represents the structure and semantics of the OPythn program.

References

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