NPRG045 Project Specification: OPythn

Marcel Goh

March 7, 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 runs on macOS and Linux platforms.

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`			

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
parameters: '(' [arglist] ')'
arglist: 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
```

3 Implementation

The front-end the program will be written in OCaml. This consists of getting the user input, displaying program output, as well as functionality for lexing, parsing, and compiling the Python code. The lexing and parsing will be accomplished using `ocamllex` and `ocamlyacc`, unless more flexibility is required than those libraries can provide, in which case hand-written rules will be applied.

The target bytecode will read by a virtual machine and for this step, we will consider writing certain functions in C and calling them into the OCaml code if necessary, via OCaml's interface with C. This may facilitate low-level operations on bitstrings, for instance. Designing the bytecode and virtual machine is a core part of the project. The virtual machine will be stack-based with typed data. The plan is to have one-byte instructions, which should allow a relatively wide instruction set. This means that different types of data can be handled by separate instructions. A significant advantage of writing the bytecode interpreter in OCaml (as opposed to switching to pure C for this step) will be that we can make use of OCaml's native garbage collector to manage OPythn's garbage collection.

References

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