NPRG045 Project Report: OPythn

Marcel Goh

June 29, 2019

1 Introduction

The OPythn project implements 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 interpret OPythn source code directly from a file. This report will consist of a specification of the interpreter and the OPythn language, followed by a brief description of the implementation.

2 Getting started

2.1 Installation

OPythn should work on Unix/Linux operating systems. OPythn's source code requires that an installation of OCaml be present on the system in order to compile. In addition, the following packages are needed:

- `extlib`
- `menhir`
- `ppx_deriving`

The user may find it convenient to use OPAM, OCaml's package manager. All required packages can be installed via OPAM by running `opam install extlib menhir ppx_deriving`. Once this is set up, running `make` from the main OPythn directory compiles the program. To delete files generated by Make, run `make clean`.

2.2 Usage

Once compiled, OPythn can be started using commands of the format `./main [options] [filename]` from the main `opythn` directory. A list of valid command-line options can be displayed by running `./main --help`. Calling the program with no options or file arguments starts the REPL (read-eval-print loop) with default options.

```
+-------------------------------------------+
|            OPYTHN INTERACTIVE MODE        |
|   Author: Marcel Goh (Release: 24.VI.2019)|
|          Type "Ctrl-C" for options.       |
+-------------------------------------------+
]=>
```

In the REPL, the user can enter OPythn statements line-by-line for the interpreter to evaluate. The REPL saves the environment between inputs. Entering `Ctrl-C` interrupts the REPL and opens a menu from which the user can set options, clear the current environment, or exit the program.

Another way to run OPythn source code is to enter an entire program into a plain-text editor and save it as a file with the extension `.opy`. Then running `./main <filename>` from the command-line interprets the code and produces the specified program behaviour. Simple examples of OPythn programs are included in the `examples` folder.

3 The OPythn programming language

This section describes the OPythn language. To avoid repeating too much of the Python specification, it is assumed that the reader has basic familiarity with Python.

3.1 Lexical conventions

The OPythn interpreter reads source code as ASCII characters, which are fed into a lexer. The following is a brief outline of the input that the interpreter expects.

3.1.1 Line and block structure

The end of a line is represented by the `NEWLINE` token. In general, simple statements cannot be split over multiple lines. However, an expression in parentheses, square brackets, or curly braces can be written over multiple lines without ending the statement. The `#` character can be used to indicate comments in source code. Any letters from `#` to the end of the line will be ignored by the lexer. A line that contains only whitespaces or comments is ignored by the lexer except during interactive evaluation, when a blank line is used to indicate the end of a multi-line statement.

3.1.2 Keywords and identifiers

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

```
True    False    None     and      or     not
 if     elif     else     for      in    while
break continue  class     is      del    pass
 in      def    global  nonlocal lambda
```

Any string that is not a keyword and contains uppercase and lowercase letters, underscores, and (except for the first character) numbers is a valid identifier. Case is significant. Some names are used by OPythn's standard library functions, which are further described in Section 3.2.7.

3.1.3 Literals

Strings are enclosed in matching single quotes or double quotes. Triple-quoted strings are not supported, nor are formatted string literals. The backslash character `\` is used to escape certain characters such as tabs and newlines.

There are two types of numeric literals in OPythn: integers and floating-point numbers. Leading zeroes in a non-zero decimal integer are not allowed. Underscores are also not allowed in numeric literals, unlike in Python 3. Floating-point literals are supported, but must be supplied in point form and not in scientific notation.

```
integer: '-'? ['1'-'9'] ['0'-'9']* | '-'? '0'*
pointfloat: '-'? ['0'-'9']* '.' ['0'-'9']* | ['0'-'9']+ '.'
```

3.1.4 Operators and delimiters

All symbolic operators and delimiters are inherited from standard Python except the `@` and `@=` symbols. (They are used for matrix multiplication and decorators, both of which are not features of OPythn.)

3.2 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 comprehensions, generators, and coroutines are omitted.

3.2.1 Simple and compound types

Every value in OPythn is of a certain type. There are four primitive types (`int`, `float`, `str`, and `bool`) and all of them are immutable. Integers are represented with 63 bits and can range between $-4,611,686,018,427,387,904$ and $4,611,686,018,427,387,903$ inclusive. Strings contain 8-bit characters.

OPythn has three compound types: `list`, `dict`, and `tuple`. Of these, only tuples are immutable. Lists are zero-indexed and accessing an element via its index takes constant time. Dictionaries support values of different types. For example, `d = {2: 'bonjour', False: 3.3}` is a valid OPythn dictionary.

3.2.2 Numeric and boolean operations

Ordinary arithmetic operators for addition, multiplication, subtraction, division, modulus, and exponentiation are supported between types `int` and `float`. When performing integer division, OPythn takes the floor of the quotient, so `-13 // 5` gives `-3`, and the modulus operator always gives a positive result, so `-13 % 5` returns `2`.

OPythn supports the bitwise operations `|`, `^`, `&`, `<<`, `>>`, and `~` on integers. Numeric values can be compared using the operators `<`, `<=`, `>`, and `>=` (chained comparisons are not allowed), and any values can be tested for equality using `==`, `!=`, `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.

3.2.3 String operations

The following useful methods are defined on strings, along with all of the methods on sequence types described in the following subsection.

```
find()  isalpha()  isdigit()  islower()  isupper()
```

3.2.4 Operations on sequence types

Strings, lists, and tuples are considered sequence types in OPythn and support a number of useful operations.

- The element at a given index can be accessed using the postfix `[ ]` operator. The expression inside must be an integer. Negative indices are allowed. For example, if `x` is the list `[1,2,3,4]`, then `x[-2]` will return `3`. Indexing into a list is the normal way to mutate a value. For example, running the expression `x[1] = 'grape'` changes `x` into the list `[1,'grape',3,4]`. This cannot be done with tuples because they are immutable.
- The number of elements in the sequence (characters, in the case of strings) can be determined using the built-in function `len()`.

- To determine if a sequence `l` contains the element `e`, one may use the expression `e in l` which returns the corresponding boolean. When used with a `for` loop, this syntax allows the user to iterate through all the elements in a list. To see if an element does not belong to a list, one may use `not in`, but of course this cannot be used to create a `for` loop. The `enumerate` object does not exist in OPythn.
- Sequence objects can be sliced using the operator `:`. For example, if `a = [1,True,3,'hi',5]`, then `a[1:3]` would give the list `[True, 3]`. This can be used with the `del` keyword to delete a group of adjacent elements in a sequence. Slicing in step increments is not supported and a `slice` object cannot be explicitly created.
- Sequences of the same type can be concatenated using the `+` operator.
- A sequence can be explicitly converted into a list by calling `list()` on it. (Calling `list()` on a list returns the same list.)

There are no list methods in OPythn. Because concatenation is performed in amortized constant time, the user may find it convenient to append to or extend a list by using the syntax `list += [new1, new2, ...]`. In addition, OPythn supports the special method `range()`, which behaves as in Python when called with one, two, or three arguments.

3.2.5 Operations on dictionaries

The following operations are defined on dictionaries.

- Accessing an element can be done using the syntax `dict[key]`. New entries can be added by running `dict[new_key] = new_value`, but duplicate keys are not allowed, so if an entry already exists with the same key, it will be replaced with the new key-value pair.
- An entry can be deleted using the syntax `del dict[key]` and the entire dictionary can be deleted by calling `del` on the whole dictionary.
- The user can determine if a key is present in a dictionary using `in` or `not in`.

Note that two dictionaries are equal if they contain the same keys and values. The following methods are defined on dictionary objects.

```
clear()  items()  keys()  values()
```

3.2.6 Branching structures

OPythn inherits the control structures `if`, `for`, and `while` from Python. Like in Python, consecutive lines at the same indentation level belong to the same block. Conditional expressions of the form `<expr> if <condition> else <expr>` are also valid.

3.2.7 Functions

OPythn supports higher-order functions and nested functions. In OPythn, functions cannot modify variables outside their scope, unless one of the keywords `global` or `nonlocal` is used. OPythn supports anonymous functions using the keyword `lambda`. OPythn functions take positional arguments only, and specifying default values is not allowed.

The following functions, implemented "under the hood" in OCaml, comprise OPythn's standard library and are included in a normal installation.

```
abs()       bin()         bool()      chr()   float()   hex()
input()  isinstance()  issubclass()  int()    len()    oct()
ord()       print()       range()     round()  str()    type()
```

3.2.8 Classes and objects

OPythn is object-oriented and allows the user to define classes. Multiple inheritance are not supported. When creating a class, the following special methods can be added to indicate how the class should behave: `__init__()`, `__eq__()`. The `__ne__()` method is not supported, as `e1 != e2` is always equivalent to `not (e1 == e2)` in OPythn.

OPythn does not allow private attributes and methods. In general, an object's attributes and methods can be accessed and mutated by functions outside the class' definition. All Python values are objects, but the user is not allowed to subclass built-in types.

3.2.9 Error handling

OPythn does not include support for exceptions. When an error occurs, the program prints an error message and either terminates or returns to the REPL.

4 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. The modules mentioned in this section can be found in the `src` folder of the OPythn repository.

4.1 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 finite state machine in OCaml. This resulting lexer matches regular expressions in the string to convert chunks of characters into the correct tokens.

The lexer will produce tokens of the following OCaml datatype:

```
type token =
  NEWLINE | INDENT | DEDENT | EOF
| ID of string  (* identifier *)
| INT of int
| FLOAT of float
| STR of string
| IF | WHILE (* other keywords... *)
| (* operators and delimiters *)
```

In OPythn, the amount of leading whitespace at the beginning of a line indicates the indentation level of the line. A tab character counts for exactly four spaces during this computation. Indentation levels of consecutive lines are used to generate `INDENT` and `DEDENT` tokens. This is done by means of a stack using the following algorithm, as described by the standard Python reference [1].

Algorithm 1 Insertion of `INDENT` and `DEDENT` tokens (as part of general lexing procedure)

```
 1: procedure LEX(OPythn source code)
 2:     token_list ← create a new empty list
 3:     stack ← create a new empty stack
 4:     stack.push(0)
 5:     while there are lines to be read do
 6:         line ← the current line
 7:         num_sp ← the number of leading spaces in line
 8:         if curr > stack.peek() then stack.push(curr) token_list.add(INDENT)
 9:         else if curr < stack.peek() then
10:             count ← 0
11:             while curr < stack.peek() do
12:                 stack.pop()
13:                 count ← count + 1
14:             end while
15:             for i ← 0, count do
16:                 token_list.add(DEDENT)
17:             end for
18:         end if
19:         tokenise the rest of the line and add tokens to token_list
20:     end while
21:     return token_list
22: end procedure
```

4.2 Parsing

The tokens produced by the lexer is then fed into the parser, which will be implemented according to the grammar rules outlined in the appendix and using `menhir`, an OCaml parser generator. The result will be an abstract syntax tree that represents the structure and semantics of the OPythn program. This tree is represented in the datatypes `Ast.op`, `Ast.expr`, and `Ast.stmt`, which represent operations, expressions, and statements respectively.

4.3 Bytecode

The abstract syntax tree is passed to the bytecode compiler, which produces instructions for a virtual stack machine, represented as the `Instr.t` datatype. Some instructions, such as `NOP` and `BINARY_ADD`, take no arguments, while others take one or several arguments. To resolve the scope of user-defined names, the compiler makes two passes through its input. During the first pass, every identifier is assigned a tag indicating the level at which it is defined. This allows the second pass to produce the correct `STORE` and `LOAD` instructions for each name.

4.4 Virtual machine

The bytecode interpreter receives as input a `DynArray`[1] of `Instr.t`. At the beginning of execution, the program counter is initialised to 0 and the stack is empty. Various hashtables hold the variable bindings of different scopes. When interpretation begins, a "global" hashtable is initialised to empty

[1] from the `extlib` library

and a "built-in" hashtable contains all of the standard OPythn methods and functions. A separate list of hashtables is maintained for function closures. The head element of this list contains all local variable bindings and the subsequent hashtables contain the bindings of the enclosing scopes.

During each iteration of the execution loop, the instruction at the index of the program counter is read and executed. Most instructions modify the stack in some way; a notable exception is the family of jump instructions, which modify the program counter instead. If the program counter is not explicitly set, it is incremented before the next iteration. When the program counter exceeds the bounds of the instruction array, the program halts.

The stack holds typed OPythn values of the following mutually recursive datatypes, defined in the `Py_val` module. Tuples can be represented as simple arrays; however, since an OPythn list may be mutated, it is more naturally implemented as a `DynArray`. To represent a dictionary, we use OCaml's `Hashtbl` module. Finally, the "sequence" is OPythn's way of dealing with iteration and lazy lists and it is relies on OCaml's native `Seq` module.

```
type cls = {
  name : string;
  super : cls option;
  attrs : (string, t) Hashtbl.t;
}
and obj = {
  cls : cls;
  fields : (string, t) Hashtbl.t;
}
and t =
  Int of int
| Float of float
| Bool of bool
| Str of string
| Fun of string * (t list -> t)
| Obj of obj
| Class of cls
| Type of string
| List of t DynArray.t
| Tuple of t array
| Dict of (t, t) Hashtbl.t
| Seq of t Seq.t
| None
```

4.5 A simple example

The reader may find it enlightening to study how OPythn interprets the following snippet of code, which multiplies two integers:

```
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
```

The lexer reads the input and splits it into the following tokens.

```
    START_FILE
    (ID "x") ASSIG (ID "int") LPAREN (ID "input") LPAREN
        (STR "Enter the multiplicand: ") RPAREN RPAREN NEWLINE
    (ID "y") ASSIG (ID "int") LPAREN (ID "input") LPAREN
        (STR "Enter the multiplier: ") RPAREN RPAREN NEWLINE
    (ID "acc") ASSIG (INT 0) NEWLINE
    WHILE (ID "y") GT (INT 0) COLON NEWLINE
    INDENT IF (ID "y") MOD (INT 2) EQ (INT 0) COLON NEWLINE
    INDENT (ID "x") TIMES_A (INT 2) NEWLINE
    (ID "y") INT_DIV_A (INT 2) NEWLINE
    DEDENT ELSE COLON NEWLINE
    INDENT (ID "acc") PLUS_A (ID "x") NEWLINE
    (ID "y") MINUS_A (INT 1) NEWLINE
    DEDENT DEDENT (ID "print") LPAREN
        (STR "The product is:") COMMA (ID "acc") RPAREN NEWLINE
    EOF
```

Next, the parser builds an abstract syntax tree that represents the structure of the program.

```
    [(Assign ((Var "x"),
        (Call ((Var "int"),
           [(Call ((Var "input"),
            [(StrLit "Enter the multiplicand: ")]))])
        ));
      (Assign ((Var "y"),
         (Call ((Var "int"),
            [(Call ((Var "input"),
             [(StrLit "Enter the multiplier: ")]))]))
         ));
      (Assign ((Var "acc"), (IntLit 0)));
      (While ((Op (Gt, [(Var "y"); (IntLit 0)])),
         [(If ((Op (Eq, [(Op (Mod, [(Var "y"); (IntLit 2)]));
                         (IntLit 0)])),
            [(Assign ((Var "x"),
                      (Op (Times, [(Var "x"); (IntLit 2)]))));
              (Assign ((Var "y"),
                       (Op (IntDiv, [(Var "y"); (IntLit 2)]))))],
            (Some [(Assign ((Var "acc"),
                            (Op (Plus, [(Var "acc"); (Var "x")]))));
                   (Assign ((Var "y"),
                            (Op (Minus, [(Var "y"); (IntLit 1)]))))])])
```

```
            ))
          ]
       ));
    (Expr (Call ((Var "print"),
           [(StrLit "The product is:"); (Var "acc")])))]
```

The bytecode compiler walks the tree twice, first resolving the scope of each identifier (in this example, all identifiers are either built-in or global) before producing the actual instruction array.

```
    0         LOAD_GLOBAL              "int"
    1         LOAD_GLOBAL              "input"
    2         LOAD_CONST               (Str "Enter the multiplicand: ")
    3         CALL_FUNCTION            1
    4         CALL_FUNCTION            1
    5         STORE_GLOBAL             "x"
    6         LOAD_GLOBAL              "int"
    7         LOAD_GLOBAL              "input"
    8         LOAD_CONST               (Str "Enter the multiplier: ")
    9         CALL_FUNCTION            1
    10        CALL_FUNCTION            1
    11        STORE_GLOBAL             "y"
    12        LOAD_CONST               (Int 0)
    13        STORE_GLOBAL             "acc"
    14        LOAD_GLOBAL              "y"
    15        LOAD_CONST               (Int 0)
    16        COMPARE_GT
    17        POP_JUMP_IF_FALSE        42
    18        LOAD_GLOBAL              "y"
    19        LOAD_CONST               (Int 2)
    20        BINARY_MOD
    21        LOAD_CONST               (Int 0)
    22        COMPARE_EQ
    23        POP_JUMP_IF_FALSE        33
    24        LOAD_GLOBAL              "x"
    25        LOAD_CONST               (Int 2)
    26        BINARY_MULT
    27        STORE_GLOBAL             "x"
    28        LOAD_GLOBAL              "y"
    29        LOAD_CONST               (Int 2)
    30        BINARY_INT_DIV
    31        STORE_GLOBAL             "y"
    32        JUMP                     41
    33        LOAD_GLOBAL              "acc"
    34        LOAD_GLOBAL              "x"
    35        BINARY_ADD
    36        STORE_GLOBAL             "acc"
    37        LOAD_GLOBAL              "y"
    38        LOAD_CONST               (Int 1)
    39        BINARY_SUB
```

```
40        STORE_GLOBAL              "y"
41        JUMP                      14
42        LOAD_GLOBAL               "print"
43        LOAD_CONST                (Str "The product is:")
44        LOAD_GLOBAL               "acc"
45        CALL_FUNCTION             2
46        POP_TOP
47        LOAD_CONST                None
48        RETURN_VALUE
```

Finally, the bytecode is interpreted and the specified behaviour is produced in the terminal:

```
Enter the multiplicand: 23
Enter the multiplier: 98
The product is: 2254
```

5 Appendix

5.1 Grammar

This is the complete OPythn grammar specification. The structure borrows heavily from Python's grammar [1]. For brevity, the AST nodes that each rule produces have been omitted. These details may be found in the full working grammar, defined in `src/parser.mly`.

```
(* Input *)
input: START_FILE file_input | START_REPL repl_input
file_input: stmt* EOF
repl_input: NEWLINE | EOF | simple_stmt | compound_stmt NEWLINE

(* Statements *)
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmts SEMIC? NEWLINE
small_stmts: small_stmt | small_stmts SEMIC small_stmt
small_stmt:
  expr_stmt | flow_stmt | global_stmt
| nonlocal_stmt | pass_stmt | del_stmt
flow_stmt: BREAK | CONTINUE | return_stmt
pass_stmt: PASS
compound_stmt:
  if_stmt | while_stmt | for_stmt
| funcdef | classdef
condition: expr | cond_expr
else_clause: ELSE COLON suite
if_stmt: IF condition COLON suite elif_stmt* else_clause?
elif_stmt: ELIF condition COLON suite
while_stmt: WHILE condition COLON suite
for_stmt: FOR ID IN expr COLON suite
suite: deep_suite
deep_suite:
```

```
    simple_stmt NEWLINE
  | NEWLINE INDENT stmt+ DEDENT
assig_target: ID | attributeref | subscription
assignment_stmt: assig_target ASSIG assignable_expr
aug_assign: assig_target bin_op_aug expr
expr_stmt: expr | cond_expr | assignment_stmt | aug_assign
del_stmt: DEL expr
return_stmt: RETURN assignable_expr?
global_stmt: GLOBAL ID
nonlocal_stmt: NONLOCAL ID

(* Expressions *)
assignable_expr: expr | cond_expr
cond_expr: expr IF expr ELSE assignable_expr
comp_op:
    LT  | GT | EQ | LEQ | GEQ | NEQ
  | IN | NOT_IN | IS | IS_NOT
bin_op_aug:
    BW_OR_A | BW_XOR_A  | BW_AND_A | LSHIFT_A {| RSHIFT_A  | PLUS_A
  | MINUS_A | TIMES_A | FP_DIV_A | INT_DIV_A | MOD_A | EXP_A
slice: COLON expr
subscription: expr LSQUARE expr slice? RSQUARE
expr:
    atom | call | attributeref | subscription
  | expr BW_OR expr | expr BW_XOR expr | expr BW_AND expr
  | expr LSHIFT expr | expr RSHIFT expr | expr PLUS expr
  | expr MINUS expr | expr TIMES expr | expr FP_DIV expr
  | expr INT_DIV expr | expr MOD expr | expr EXP expr
  | MINUS expr | BW_COMP expr
  | expr OR expr | expr AND expr | NOT expr
  | expr comp_op expr | LPAREN expr RPAREN
  | LAMBDA param_id_list COLON expr
attributeref: expr DOT ID
key_datum: expr COLON expr
key_datum_list: key_datum | key_datum COMMA key_datum_list
tuple_list: expr COMMA expr | expr COMMA tuple_list
atom:
    ID | INT | FLOAT | STR | TRUE | FALSE | NONE
  | LPAREN RPAREN | LPAREN expr COMMA RPAREN
  | LPAREN tuple_list RPAREN | LSQUARE argument_list? RSQUARE
  | LCURLY key_datum_list? RCURLY
call: expr LPAREN argument_list? RPAREN
argument_list: expr | expr COMMA argument_list

(* Function and class definitions *)
funcdef: DEF ID LPAREN param_id_list? RPAREN COLON suite
classdef: CLASS ID class_params? COLON suite
param_id_list: ID | ID COMMA param_id_list
```

```
    class_params: LPAREN RPAREN | LPAREN ID RPAREN
```

References

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