CPPython: A reduced Python interpreter written in C

Dayanne Fernandes da Cunha

Department of Computer Science, University of Brasilia, Distrito Federal, Brasil
130107191@aluno.unb.br
https://github.com/Dayof

Abstract. This project aims to create a reduced version of *CPython*.

Keywords: Python · Language Processor · Interpreter · *CPython*.

1 Introduction

The evolution of programming languages goes from machine code (1st generation), a binary format of the instructions that will be executed by the *CPU*, to constraint programming (5th generation), focusing on logical problems, such as mathematical theorems proof.

After the 2nd generation of programming languages there was the need to have a language processor to transform the source language into machine code. There are three types of language processors: compiler, assembler and interpreter. A compiler is a program that receives a source program, in one language, and translate it into a program in another language with equivalent semantics. The assembler focus on processing the Assembly language into relocatable machine code. Likewise the compiler, the interpreter also receives a source program, but instead of only generating a target program, the processor executes the source operations alongside inputs producing outputs [ALSU06].

This project aims to create an interpreter for the language *Python* using *C* language. The lexer and parser components are implemented using the tools *Flex* and *Bison*.

2 Proposal

In this project a reduced version of *CPython* will be implemented. This version is limited and will enable only simple arithmetic's operations, read and write commands, boolean conditionals, functions and flow structures. Further information about what syntax will be maintained in this interpreter comparing to *CPython*, please read the Section 4.

2.1 Motivation

There are many different implementations of *Python*, e.g. *Jython* [Hug97] to integrate with *Java* modules, *IronPython* [H+04] for better communication with *.NET* components. The latest version of *Python*, 3.8.5, has *CPython* [VR+07] as standard language processor. *CPython*'s a *Python* implementation in *C* that can be used in many different topics and fields, e.g. web development, data science, scientific computing, machine learning.

Different from others 3rd generation languages, *Python* is an interpreted language, although it is first translated into *bytecode*. It executes the operations from the source program along with the inputs producing outputs. *Python* is stable, well maintained, easy to learn, and have a good and extensive documentation. In addition, *Python* is versatile, it has three different programming paradigms, imperative, functional and object-oriented. *CPPython* is inspired by *CPython* but will keep only the imperative and functional paradigms.

2.2 Architecture

The book [GM10] will be used as a guidance for the implementation of the abstract machine and the memory management. There are two major components, the store and the interpreter as described in Figure 1. The component LEXER is described in Section 3 and the PARSER is detailed in Section 4. Other components of this language processor will be implemented and described throughout the semester.

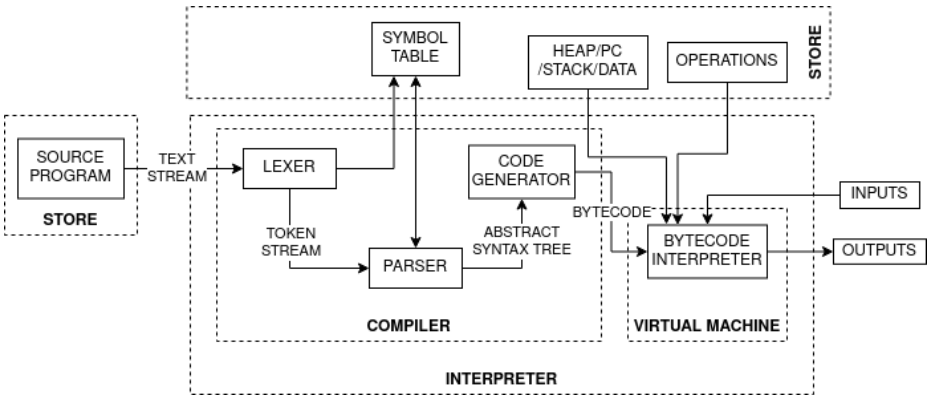

Fig. 1. *CPPython* architecture overview.

The architecture in the Figure 1 shows a diagram that represents both the compiler and the *CPPython* virtual machine, and how they are connected. It was inspired by the architecture suggested in the book [GM10], presented in Chapter 1, Abstract Machines.

2.3 Grammar

The lexicon and syntax grammar are defined using *EBNF* (Extended Backus-Naur Form), this notation enables the description of terminals, non terminals and control forms [ISO96]. The lexicon grammar is defined in Section 3 and the syntax in Section 4.

3 Lexical Analyzer

There are two major steps before generating the target program, the analysis and the synthesis phase. This section, about the lexical analysis, the parser described at Section 4, and the module that generates the intermediate code representation belongs to the first part of the process.

3.1 Architecture

The lexical analyzer in *CPPython* is called LEXER, as we could see in the Figure 1. This module receives a character stream, analysis it trying to find lexemes related to patterns and construct tokens to send to the parser module. After it, a new item in the symbol table is initialized with information regarding the object's name and send as reference to the PARSER. The LEXER module overview can be found in Figure 2.

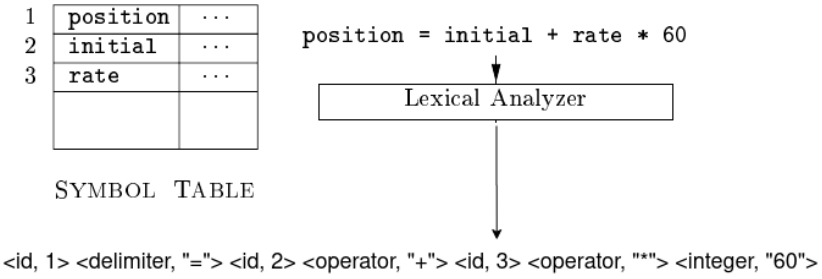

Fig. 2. *CPPython* lexer input, output and symbol table. This image was taken from [ALSU06], it has a small addition with the pair <token, lexeme> example.

The input needs to match the *CPPython* formal grammar defined in Subsection 3.2, e.g. the input "var = 1 + 1" will generates the following output <id, 1> <delimiter, => <integer, '1'> <operator, '+'> <integer, '1'>. The pair <**token, lexeme**> represents the information send to the parser.

3.2 Grammar

```
nzerodigit   := '1'  | ... | '8' | '9'
digit        := '0' | nzerodigit
letter       := 'a' | ... | 'z' | 'A' | ... | 'Z'
whitespace   := ' ' | '\t'
newline      := '\n'
add          := '+'
sub          := '-'
mult         := '*'
div          := '/'
assign       := '='
boolean_op   := '==' | '>='| '<=' | '!=' | '>' | '<' | '|' | '
                              &' | '~';
comp_op      := '<'|'>'|'=='|'>='|'<='|'<>'|'!='
delimiter    := '(' | ')' | ',' | '.' | ':' | ';' | '[' | ']';
other_s      := '%' | '?' | '_' | '\' | '^' | '@' | '`' | '{'
                              | '}' | '!';
symbols       := whitespace | newline | operator | boolean_op
                              | delimiter | other_s
boolean      := 'True' | 'False'
null         := 'None'
name_dq      := "" | char_dq name_dq
name_sq      := '' | char_sq name_sq
char_dq      := char | "'"
char_sq      := char | '"'
char         := letter | digit | symbols
string       := '"' name_dq '"' | "'" name_sq "'"
integer      := nzerodigit digit* | '0'
float        := [digit]+"."digit+
var          := (letter | '_') (letter | digit | '_')*
```

3.3 Error Handler

Tokens that are not recognized from any regular expression will be showed in the interpreter output as a *LexerError*, showing the line and column indexes that this character/pattern was not correctly identified, e.g.:

```
columns |1|2|3|4|5|6|7|8|9|10|
line 1. |v|a|r|_|2| |=| |@|
```

The operator += below in the line 2 will not be implemented in *CPPython* according to the Subsection 4, so the parser will not understand += as an operator, but the lexer understand += as an operator and a delimiter, "+" and "=" separately. Although += is recognized by the lexer, the character @ is not recognized, so the output for this case will be:

```
LexError: token '@' is not recognized in line 1, column 9.
```

3.4 Symbol Table

The library *uthash* [Han] is used to create the symbol table structure. Each symbol has a structure *word* that contains a key of type integer, a type reference as integer, a char, called name, with limit of 79 characters and an instance of an internal object from *uthash* UT_hash_handle, called hh.

There are four main functions to help add, delete and find words in the symbol table, void add_word(int key, char *name), struct word *find_word(int word_key), void delete_word(struct word *s) and void delete_all_st().

During the lexical component the symbol table will contain only the variables positions and names, e.g. the instruction "`var = 1`" will be initialized in the position 0 with name "var", but only during the parser component the type will be set.

3.5 Code Structure

GCC, version 9.3.0 and FLEX [Pax] 2.6.4 were used to build the lexical analyzer. Some of the internal Flex functions were used to read a character stream from a file and define the tokens and patterns. This Flex definition of *CPPython* is defined in the file **lexer/cppython.lex**.

3.6 Tests

There is an instruction to compile and run *CPPython* in the *README.md*. After executing the usage steps, there are some files to test the lexical grammar cases, e.g.:

Valid Lex The file "cppython/src/tests/lexer/valid_assigns.ppy" test valid patterns allowed by the language. For example, the following structure should be understood as valid by *CPPython*.

```
some_variable = True
```

Invalid Lex The file "cppython/src/tests/lexer/invalid_assign_type.ppy" test invalid characters pattern not allowed by the language. For example, the following structure should be understood as invalid by *CPPython*.

```
some_variable_1 = @
some_variable_1 = ^
```

4 Syntax Analyzer

4.1 Architecture

The syntax analyzer in *CPPython* is called PARSER. It is a module that receives 2 inputs, the tokens recognized and the symbol table initialized by the lexer. As output, the parser will update the symbol table with more relevant information about the identifies, e.g., type, value casting, and generate the abstract syntax tree. The overview of its architecture can be found in Figure 3.

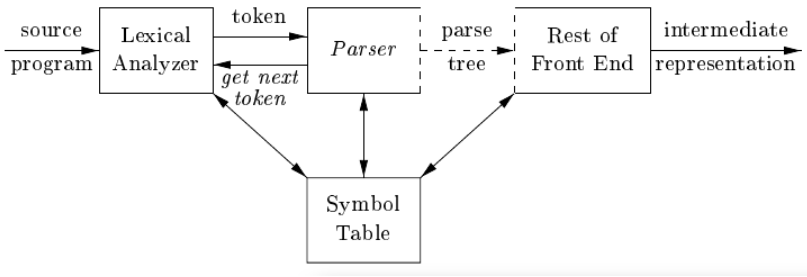

Fig. 3. *CPPython* parser inputs and outputs [ALSU06].

The process of parsing an instruction goes from trying to find a possible syntax match of the tokens received from the lexer, starting from the initial symbol in the grammar, until it finds the correct patterns, otherwise an error is emitted. There are three types of parser that can be used to match the string patterns, universal, top-down and bottom-up. This proposal will use the **bottom-up** approach **ielr** [DM10].

4.2 Grammar

There are two terminals below identified by *INDENT* and *DEDENT* that were not defined in this section, because it's not a common token/terminal. It will be defined in the parser module since there is a further logic to process them. This grammar syntax is a modified and reduced version of *CPython*, not all basic types will be found here, e.g. dict.

```
# general
input       := "" | input line
line        := NEWLINE | stmt NEWLINE | error NEWLINE
stmt        := simple_stmt | compound_stmt NEWLINE
simple_stmt := var ASSIGN expr
expr        := arith_expr | BOOLEAN
comp_stmt   := if_stmt | while_stmt | for_stmt | funcdef
var         := ID
```

```
comments     := '#' char* NEWLINE
suite        := simple_stmt | NEWLINE INDENT stmt+ DEDENT;

# arithmetic's expression
arith_expr   := arith_expr ADD term | arith_expr SUB term |
                              term;
term         := term MULT factor | term DIV factor | factor;
factor       := INTEGER | FLOAT;

# list
list         := '[' [sublist] ']'
sublist      := types (',' types)* [',']
types        := INTEGER | FLOAT | STR | VAR

# read and write
input_stmt   := 'input(' STR ')'
print_stmt   := 'print(' STR ')'

# conditional
if_stmt      := 'if' test ':' suite ('elif' test ':' suite)* [
                               'else' ':' suite]
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)*

# flow
while_stmt   := 'while' test ':' suite ['else' ':' suite]
for_stmt     := 'for' VAR 'in' (list | STR) ':' suite ['else'
                               ':' suite]

# function
funcdef       := 'def' var parameters ':' suite
parameters    := '(' [args] ')'
args          := (var (',' var )*)
flow_stmt     := 'return' [types]
```

4.3 Error Handler

When a syntax error is found, the system emits the token found and expected, also the line and column that this syntax error appeared in the source file. Example:

```
SyntaxError: syntax error, unexpected SUB, expecting INTEGER
                                or BOOLEAN or FLOAT in line 4,
                                 column 6.
```

4.4 Abstract Syntax Tree

The AST is generated by the PARSER using the grammar rules defined in the file *parser/cppython.y*, example of AST:

```
--- INSTRUCTION:
    g = 4 + 4 * 2 * 5 - 3 / 0

--- AST:
OP: =
  OP: -
    OP: /
      INT: 0
      INT: 3
    OP: +
      OP: *
        INT: 5
        OP: *
          INT: 2
          INT: 4
      INT: 4
  VAR: g
```

Each instruction generates a new AST, so there are a list of AST that are showed in the end of the *CPPython* lexer and parser processing.

4.5 Symbol Table

During the parser component the symbol table is updated with the variables type, recursively trying to find the leafs types. Example of symbol table:

```
## Symbol Table ##
Size: 6

KEY: 0, NAME: x, TYPE: FLOAT
KEY: 1, NAME: z, TYPE: FLOAT
KEY: 2, NAME: w, TYPE: FLOAT
KEY: 3, NAME: t, TYPE: BOOL
KEY: 4, NAME: o, TYPE: INT
KEY: 5, NAME: p, TYPE: BOOL
```

4.6 Code Structure

GCC, version 9.3.0 and Bison [Cor] 3.5.1 were used to build the parser analyzer. The *Bison* definition of *CPPython* is defined in the file **parser/cppython.y**.

4.7 Tests

There is an instruction to compile and run *CPPython* in the *README.md*. After executing the usage steps, there are some files to test the syntax grammar cases, e.g.:

Valid Syntax The files "cppython/src/tests/parser/valid_arithmetic_int_float.ppy", "cppython/src/tests/parser/valid_arithmetic_int.ppy" and "cppython/src/tests/parser/valid_assigns.ppy" test arithmetic operations rules and different type assignments allowed by the language. For example, the following structure should read as valid by *CPPython*.

```
x = 1 + 1.0 - 2 / 3 * 8
```

Invalid Syntax The files "cppython/src/tests/parser/invalid_arithmetic.ppy" and "cppython/src/tests/parser/invalid_assigns.ppy" test arithmetic operations rules and different type assignments not covered by the language. For example, the following structure should be understood as invalid by *CPPython*.

```
x = {"a" : 1}
p = NULL
```

5 Semantic Analyzer

The semantic rules are checked after a grammar parsing is trigger by Bison. For example, when we receive an input that triggers the grammar rule "simple_stmt := var ASSIGN expr", var type will be var.type = expr.type . This **type** attribute is synthesized from **expr** bottom-up parse tree.

5.1 Semantic rules

Dynamic type Everything in *CPPython* is an object. That means that even basic data types such as integer or float will contain a "bucket" with its memory address, type, name and value. This makes *CPPython* dynamically typed, a variable name can point to any objects of any type without the need of declaring a variable before the definition, e.g.:

```
var = 1
var = 'fish'
var = [42]
```

Mutable and immutable objects This proposal implements only one mutable object, the list. Integer, float, string, boolean and null are immutable objects, which means that every time you modify an object it will create another object with a new value. For example, we can see a immutable situation below, the first instruction creates an object with the value 1, type integer and x points to this object. The second instruction tells y to point to x, and in the last instruction 1 will be add to x, and a new object with value 2 and type integer will be created, x will then point to this new object and y continue pointing to the old object with value 1.

```
x = 1
y = x
x = x + 1
```

Implicit type conversion An implicit conversion between integer and float will be implemented. The first instruction below will result into an object of type integer, all the other three will result into float type. If there is at least one object of float type in the middle of the expression, the new object from it will be of the float type.

```
x = 1 + 1
y = 1.0 + 1.0
z = 1 + 1.0
w = 1.0 + 1
```

Newline in the end of file Since there are some structures, like the comment, that is only recognize as a token if there is a newline in the end of the structure, this requires that every *CPPython* instruction's file contains a newline in the end of it.

Variable scope *CPPython* uses a very strict static scope rule. Any assignment in a logic block, that is recognized when there is any uniform level of indentation, is considered local. *CPython* uses static scope with some extra keywords to allow the block to use variables outside the local scope, such as *global* and *nonlocal*, although *CPPython* doesn't implement them.

5.2 Error Handler

When a semantic issue is found, the system is not killed, instead it continues with a warning message, it says the line/column of the problem found and a message telling the user what exactly is the issue. Example:

```
SemanticWarning: implicit add with wrong type, BOOL and INT
                                in line 4, column 6.
```

5.3 Code Structure

The files that contains the semantic rules definition and declarations are "cppython/src/semantic/main.c" and "cppython/src/semantic/main.h". These files contains all the structure and code to check for correct type during arithmetic operations, variable scope, correct end of file, type conversions, mutable objects, and dynamic typing. To compile them it was used GCC, version 9.3.0.

5.4 Tests

There is an instruction to compile and run *CPPython* in the *README.md*. After executing the usage steps, there are some files to test the semantic cases, e.g.:

Valid Semantic The example "cppython/src/tests/semantic/valid_arithmetic.ppy" test arithmetic operations with correct types, for example, only integers and floats should make operations such as add, sub, div and mult:

```
x = 1 + 1.0 + 5
```

Invalid Semantic The example "cppython/src/tests/semantic/invalid_arithmetic.ppy" test arithmetic operations with incorrect types, for example, a boolean is never allowed to be add with an integer nor float:

```
x = True + 1.0
```

In addition to last example, the file "cppython/src/tests/semantic/invalid_eof.ppy" test if a semantic warning is emitted in case there is no newline in the end of file.

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, techniques, and tools (2nd edition)*, Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[Cor] Robert Corbett, *Bison 3.7.1.* `https://www.gnu.org/software/bison/manual/bison.html`, last accessed on 31/10/20.

[DM10] Joel E Denny and Brian A Malloy, *The ielr (1) algorithm for generating minimal lr (1) parser tables for non-lr (1) grammars with conflict resolution*, Science of Computer Programming **75** (2010), no. 11, 943–979.

[GM10] Maurizio Gabbrielli and Simone Martini, *Programming languages: Principles and paradigms*, Springer London, 2010.

[Han] Troy D. Hanson, *uthash: a hash table for c structures.* `https://troydhanson.github.io/uthash/`, last accessed on 26/09/20.

[H+04] Jim Hugunin et al., *Ironpython: A fast python implementation for .net and mono*, Pycon 2004 international python conference, 2004.

[Hug97] Jim Hugunin, *Python and java: The best of both worlds*, Proceedings of the 6th international python conference, 1997, pp. 2–18.

[ISO96] ISO Central Secretary, *Information technology — syntactic metalanguage — extended bnf (1st edition)*, Technical Report ISO/IEC TR 14977:1996(E), International Organization for Standardization, Geneva, CH, 1996 (en).

[Pax] Vern Paxson, *Lexical analysis with flex, for flex 2.6.2.* `https://westes.github.io/flex/manual/`, last accessed on 08/09/20.

[VR+07] Guido Van Rossum et al., *Python programming language.*, Usenix annual technical conference, 2007, pp. 36.