# 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 is the need to have a language processor to transform the source language into machine code. There are three types of language processors, compiler, interpreter and assembler. 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. 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. The assembler focus on processing the Assembly language into relocatable machine code [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 Subsection 2.3.

## 2.1 Motivation

There are many different implementations of Python, e.g. *Jython* to integrate with Java modules, *IronPython* for better communication with .NET components. The latest version of Python, 3.8.5, has *CPython* 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 compiled and interpreted, 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 2 major components, the store and the interpreter as described at the Figure 1. The component LEXER is described at the Section 3 and the PARSER is detailed at the Section 4. Other components of this language processor will be implemented and described thorough the semester.

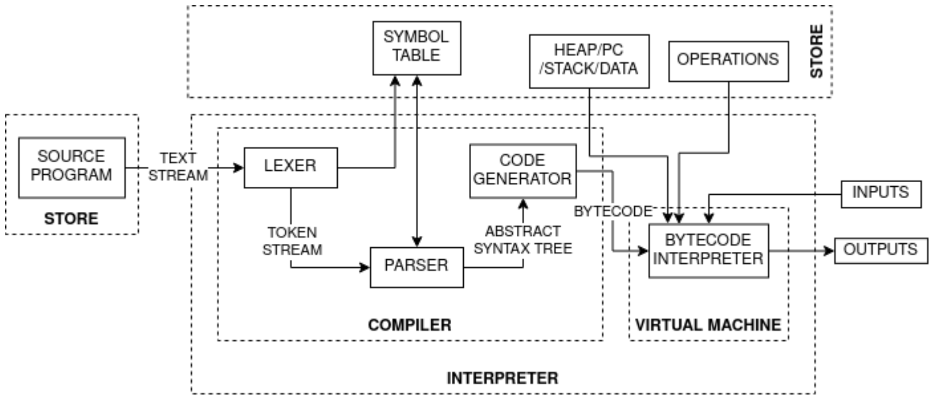

**Fig. 1.** *CPPython* architecture overview.

## 2.3 Syntax

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

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 at 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.

```
# literals
nzerodigit   := '1'  | ... | '8' | '9';
digit        := '0' | nzerodigit;
letter       := 'a' | ... | 'z' | 'A' | ... | 'Z';
whitespace   := ' ' | '\t';
newline      := '\n';
operator     := '+' | '-' | '/' | '*';
boolean_op   := '==' | '>='| '<=' | '!=' | '>' | '<' | '|' | '
                               &' | '~';
delimiter    := '=' | '(' | ')' | ',' | '.' | ':' | ';' | '['
                               | ']';
other_s      := '%' | '?' | '_' | '\' | '^' | '@' | '`' | '{'
                               | '}' | '!';
symbols       := whitespace | newline | operator | boolean_op
                               | delimiter | other_s
boolean      := 'True' | 'False';
null         := 'None';

# types
name_dq      := "" | char_dq name_dq;
name_sq      := '' | char_sq name_sq;
char_dq      := char | "'";
char_sq      := char | '"';
char         := letter | digit | symbols;

list         := '[' [sublist] ']' [sliceop];
sublist      := types (',' types)* [','];
sliceop      := [integer] ':' [integer];

string       := '"' name_dq '"' | "'" name_sq "'";
integer      := nzerodigit digit* | '0';
float        := [digit+] "." digit+ | digit+ ".";
number       := integer | float;
index_types  := string | list
types        := index_types | number | boolean | null;
atom_expr    := index_types trailer*
trailer      := '[' subscriptlist ']' | '.' NAME

# general
var              := (letter | '_') (letter | digit | '_')*;
stmt             := newline | simple_stmt | compound_stmt
                               newline;
simple_stmt      := (expr_stmt | flow_stmt) [';'] newline
expr_stmt        := var '=' (var | types)
```

```
compound_stmt    := if_stmt | while_stmt | for_stmt | funcdef
comments         := '#' char* newline
suite            := simple_stmt | newline INDENT stmt+ DEDENT

# arithmetic's expression
exprlist: expr (',' expr)* [',']
expr: arith_expr ('|' arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'/') factor)*
factor: ('+'|'-') factor | atom_expr

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

# 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)*
comp_op     := '<'|'>'|'=='|'>='|'<='|'<>'|'!='

# flow
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' (list | string) ':' suite ['
                              else' ':' suite]

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

**Syntax modifications** The non literal *symbol* was reduced and the non literal *other_s* was added to separate the literals allowed in general data types and inside of strings. Addition of other non literals: *whitespace*, *newline*, *boolean_op*, *delimiter*, *boolean* and *null*.

## 2.4 Semantic

**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 modified 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 at 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 at the end of it.

### 2.5 Tech Stack

The language processor is implemented using C language and the tools FLEX, for lexical analyses, and BISON for the parser.

## 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 valid tokens, creates the symbol table and initialize it with information regarding objects name. The module overview can be found at the Figure 2.

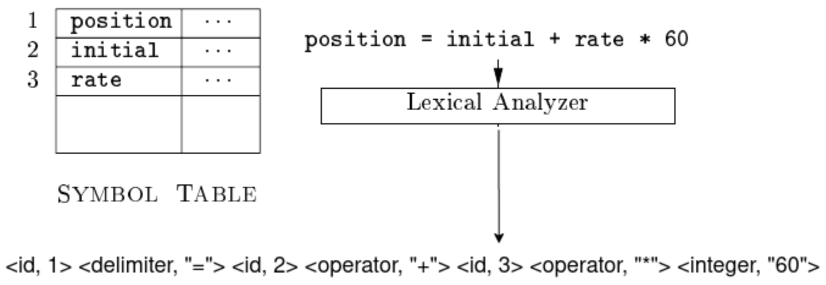
1 position ...
2 initial ...
3 rate ...
Symbol Table
position = initial + rate * 60
Lexical Analyzer
<id, 1> <delimiter, "="> <id, 2> <operator, "+"> <id, 3> <operator, "*"> <integer, "60">

**Fig. 2.** *CPPython* lexer input, output and symbol table.

The input needs to match the *CPPython* regular expressions, defined at the Subsection 3.2, that follows the formal grammar defined at the Subsection 2.3, e.g. the input "var = 1 + 1" will generates the following output $< id, 1 >< delimiter, => < integer,' 1' >< operator,' +' >< integer,' 1' >$.

There are two types of outputs $< token, index >$ and $< token, lexeme >$. The first one is to show the variable token and the index position of this token in the symbol table. The second case is for the other types of tokens, e.g. delimiters, operators. In this case the *lexeme* will be printed between single quotes.

## 3.2 Regular expressions

```
DIGIT                    [0-9]
NDIGIT                   [1-9]
LETTER                   [a-zA-Z]
OPERATOR                 ("-"|"+"|"*"|"/")
BOOLEAN_OP               ("=="|"<="|">="|"!="|"<"|">"|"~"|"|"|"&"|
                                   "and"|"or"|"not")
DELIMITER                ("="|"("|")"|"["|"]"|";"|","|"."|":")
BOOLEAN                  ("True"|"False")
NULL                     ("None")
NEWLINE                  (\n)
WHITESPACE               ([ \t]+)
VAR                      ({LETTER}|"_")({LETTER}|{DIGIT}|"_")*
STRING                   (\".*\"|\'.*\')
INTEGER                  ({NDIGIT}{DIGIT}*|"0")
FLOAT                    ([{DIGIT}+]"."{DIGIT}+|{DIGIT}+".")
```

```
NUMBER              ({INTEGER}|{FLOAT})
("input"|"print"
 |"if"|"elif"|"else"
 |"return"|"def"
 |"for"|"while")    return KEYWORD;
{INTEGER}           return INTEGER;
{FLOAT}             return FLOAT;
{OPERATOR}          return OPERATOR;
{DELIMITER}         return DELIMITER;
{BOOLEAN}           return BOOLEAN;
{BOOLEAN_OP}        return BOOLEAN_OP;
#.*{NEWLINE}        return COMMENT;
{STRING}            return STRING;
{NEWLINE}           return NEWLINE;
{WHITESPACE}        return WHITESPACE;
{VAR}               return ID;
.                   return ERROR;
```

## 3.3 Error Handler

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

```
columns |1|2|3|4|5|6|7|8|9|10|
line 1. |d|e|f| |f|u|n|(|)|: |
line 2. | |v|a|r| |=| |1| |  |
line 3. | |v|a|r| |+|=| |1|  |
line 4. | |v|a|r|_|2| |=| |@ |
```

The operator += below in the line 2 will not be implemented in *CPPython* according to the Subsection 2.3, 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:

```
line 1.  <keyword, 'def'> <id, 1> <delimiter, '('> <delimiter
                                    , ')'> <delimiter, ':'>
line 2.  <id, 2> <delimiter, '='> <integer, '1'>
line 3.  <id, 3> <operator, '+'> <delimiter, '='> <integer, '
                                    1'>
line 4.  <id, 4> <delimiter, '='>
LexerError: line 4, column 10, token '@' is not recognized
```

## 3.4 Symbol Table

The library *uthash* [O1'D] is used to create the symbol table structure. Each symbol has a structure *word* that contains a key of type integer, a char, called name, with limit of 50 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 $delete\_all()$.

### 3.5 Tech Stack

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 patterns. Flex uses the follow code structure:

```
 %{
   declarations
 %}
   definitions
%%
   rules (regex)
%%
   subroutines
```

This structure above is defined in the file **cppython.lex**. The documentation about the system requirements and how to compile and run this file can be found at the **README.md** and the bash script called **build.sh**.

## 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, and generate the abstract syntax tree. The overview of its architecture can be found at the Figure 3.

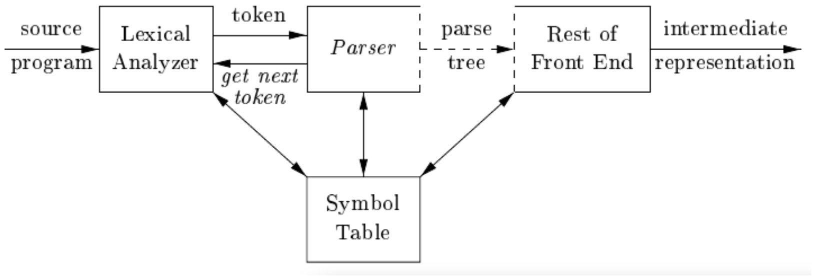

**Fig. 3.** *CPPython* parser inputs and outputs [ALSU06].

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.

# 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.

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

[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).

[O'D] Arthur O'Dwyer, *uthash: a hash table for c structures*. `https://troydhanson.github.io/uthash/`, last accessed on 26/09/20.

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