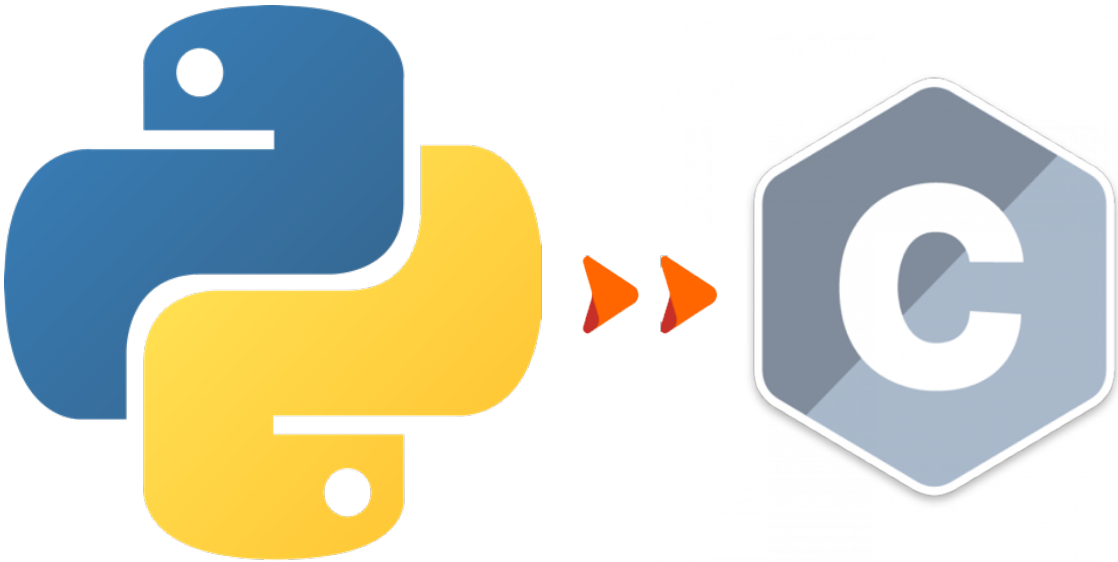
Python to C compiler



Piergiorgio Ladisa
Mat. [573036]

April 12, 2018

Contents

List of Figures

Chapter 1

Introduction to the Python to C compiler

1.1 Python

Python was created during the early '90s by Guido van Rossum as a successor of the language known as ABC and distribuited under a GPL-compatible licence and so all the releases of Python can be considered Open Source. Since those years, the language has undergone continuous developments. Python is a powerful high-level programming language, born to combine the advantages of scripting with those of structured programming. Often happens to have the need to add more functions to a large script: this could weigh it down considerably. It may also happens to want to add features that are only accessible via C, as the system-call. Therefore, Python starts to be an interpreted language and this speeds up the development of the program as it avoids the compiling and linking phases; nevertheless it offers many more facilities and support for big programs despite of shell languages. Python also allows to split the program in reusable modules and to write them in a more compact way. In this way, the program results to be easier to read for these reasons:

- high-level data types allow to express complex operations in few instructions;
- instructions are not grouped by parenthesis but through indentation;
- it is not necessary to declare variables.

Finally, one of Python's greatest potential is to be *extensible*: it is possible to add to the interpreter new functions or built-in modules to execute critical operations in a fast manner or to create links to libraries.

1.2 C

C is a general-purpose programming language, developed by Dennis M. Ritchie and Brian W. Kernighan. The main important ideas at the basis of C stem from the languages BCPL and then the language B. The language C provides a variety of data types. The fundamental types are characters, integers, floating point and then some derived data types created with pointers, arrays, structures and unions. The expressions are formed by operators and operands; C also provides the fundamental control-flow constructions required for *well-structured programs* as statement grouping, decision making (if-else), looping with tests (while, for, do-while) and selecting of a set of possible cases (switch). C also provides the feature to define functions that can be called when they are needed. Functions of a C program may exist in separate source files that are compiled separately and a preprocessing step performs macro substitution on program text, inclusion of other source files and conditional compilation. Therefore, C is a relatively "low level" language, that it means that C deals with the same sort of objects that most computers do, namely characters, numbers but also addresses: in this way, programmer can also go down to memory level and manage low-level details.

1.3 Python to C compiler

The compiler realized as a project is a *source-to-source compiler* that translates a Python 2.7 program into a C program. The steps to perform the translation are the following:

1. the **lexical analyzer** reads the character stream from the source program written in Python and tokenizes it. In this step I utilized the tool **Flex**, with which I defined the regular expressions for the patterns that identify *identifiers, strings, reserved words of Python language, integers, floating points, strings and symbols of arithmetical and conditional expressions*, and for each one a token is generated. For what concerns identifiers, they are stored in the symbol table. At the same time, values of integers, floating points and strings are stored into the global variable **yylval**. In this way, the scanner and the parser can share not only symbols, but also their values;

2. a **syntactic analyzer** that receives as input the string of tokens from the scanner, verifies if the string of tokens can be accepted from the grammar defined and, if everything is correct, generates as its output the **abstract**

syntax tree. The tool utilized in this step is the parser generator known as **Bison**;

3. finally, a **semantic analyzer** evaluates the tree to generate the target program in C or to check if semantical errors occur. In this step I didn't use any tools, but it was all written by hand as external routines called by Bison during the parsing process.

The following figure shows the block diagram of the system.

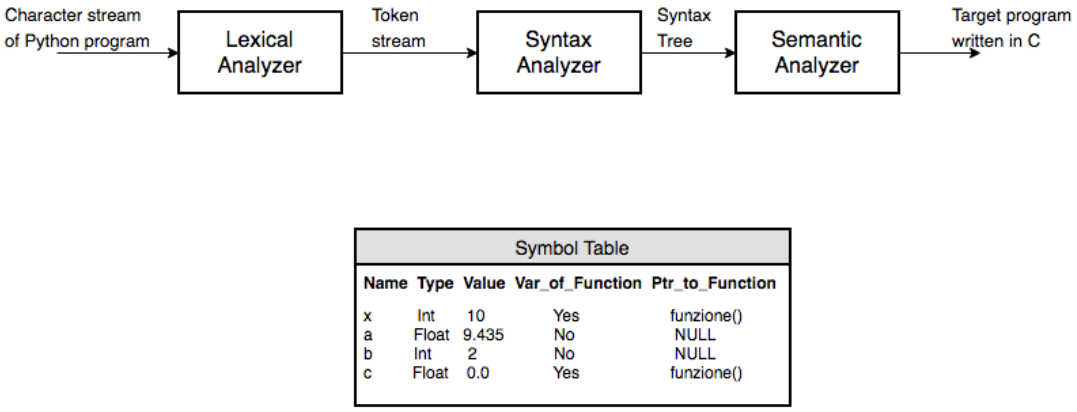


Figure 1.1: Block diagram of Python to C compiler

Chapter 2

Description of the project

2.1 Restriction on the source language

Since the design of a compiler for an entire language is a long and laborious process, the following restriction on Python language was decided for my project:

- list of statements;
- single line comments;
- three types of selection statements, that is *if*, *if-else* and *if-elif-else*;
- one loop statement, that is *while* construct;
- two built-in functions, one for the input, that is *raw_ input()*, and one for the output, that is *print*.
- definition of **one** user function without return option[1]

In addition to what has been said, it has been implemented an Abstract Syntax Tree and a Symbol Table through an hash table.

2.2 Symbol table

One of the essential function of a compiler is to store the variable names used in the source program and collect all the useful information that regards them. To reach this goal it is used a **symbol table**, that is a data structure containing a record for each variable name, with appropriate fields for attributes of that name.

First of all, in my project, a **symbol** is an istance of a struct in which are recorded these informations:

[1]This choice will be motivated later.

- name of the identifier;
- type of the identifier, that could be one of these: not-defined, void, integer, floating-point and a string;
- value of the identifier;
- a flag that specifies if that identifier belongs to a general scope or to that of a function: if this flag is setted to 1, it belongs to the user-function, otherwise to the main-scope;
- pointer to the AST of the function to which the variable belongs.

The symbol table is defined as an array of fixed size, setted to $NHASH = 9999$, and contains one routine, **lookup(char *name)**, which takes as input the name of the symbol and returns the address of that name or creating a new entry if there isn't one already. It is important to observe that the lookup technique is known as *hashing with linear probing*: it uses an hash function to transform the string in input into an entry number in the table, then checks the entry and, if it is already taken by a different symbol, scans linearly until it finds a free entry. The hash function multiplies for each character the previous hash by 9 and then *xor* the character, doing all the arithmetic as unsigned, which ignores overflows.

2.3 Scanner using Flex

As I previously said, the first step of the translation process is the *lexical analysis*. The latter is done by an element called **scanner**: this divides the stream of character into meaningful blocks called **token**. So, to generate a scanner, has been used the tool known as *Flex*. A Flex program has the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

In the **declarations part** have been included all the needed libraries such as the standard libraries for the C preprocessor, the header generated by Bison for the parser, and two important header files:

1. one containing the declarations of the symbol table (declared as *extern* to make it accessible from extern modules) and the ASTs

2. the other one containing the declaration of a stack and the init, pop, push and pick routines to manage and solve the **indentation problem**

The **translation rules** have the form of:

```
Pattern      {Action}
```

Each *Pattern* is a **regular expression** while the *Action* is a C code section in which it has to be indicated what action must be taken when the pattern is encountered from the scanner. In my project, the regular expressions match the following tokens:

- identifiers, that is variables or name for functions;
- integer literals;
- floating point literals;
- string literals;
- single character operators, that is +,-,*,/,=,:,(,) and comma;
- comparison operators, that is >,<,!=,==,>=,<=
- keywords of Python; in particular all the reserved words have been declared because these words cannot be used as identifiers; so for correctness, the compiler must warn the programmer if he's trying to use them as a name for a variable or a function. Obviously, for the constructs or statements implemented in the project, the actions for that specific reserved words returns a token, rather than a warning message;
- **indentation**, **dedentation** and **end of statement**;
- comments, that will be ignored.

The **auxiliary functions** part is void.

So, the Flex-generated scanner reads through its input, matches it against all of the regular expressions and does the appropriate action on each match. In conclusion, the tool translates all of the regular expressions into an efficient internal form, that is a **Deterministic Finite Automata**.

2.3.1 The "indentation problem"

A Python program uses the indentation and the dedentation to delimit a block of code and, in a list of statements, the *newline* character indicates the end of a statement. This features of Python language gives to it the advantage to be easier to read. But the main problem, from a lexical and a syntactical analysis point of view, is that now we are not managing a "context-free grammar" but, in such a way, a "context-sensitive grammar". To better understand the problem let us consider C. In this language, for example, the block of statements is delimited by { and }, while all the newlines and tab characters are ignored. In this case, we can generate a token everytime we encounter the characters { and } and consider every statement contained between these two as a block of code.

In Python, a program is divided into a certain number of *logical lines*. The end of a logical line is represented by a token that matches the new-line character, called *END_OF_STATEMENT*. The blank line, a logical line that contains only spaces, tabs, formfeeds and comment, is ignored.

So, how to scan the indentation, the end of a statement into a block of code, and the end of a block, that is a dedentation? The main idea is based on the use of a stack and as follows. The first value pushed into the stack is 0. While a new line is encountered in a list of statements, the token generated is an END_OF_STATEMENT. If it begins a new block of code that is, for example, in this case:

```
if (x<0):
    print "Hello world!"
```

the current value of indentation is pushed into the stack, defining the new **indentation level**. From this moment on, for every statement is checked if it is at:

- higher indentation level, then the new value is pushed into the stack, an INDENT token is generated and a new code-block begins;
- lower indentation level, then the current value is popped from the stack, a DEDENT token is generated and the current code-block ends;
- same indentation level, then an END_OF_STATEMENT token is generated and we are still in the same code-block.

2.4 Abstract syntax tree

In an **abstract syntax tree** each interior nodes represent programming constructs.

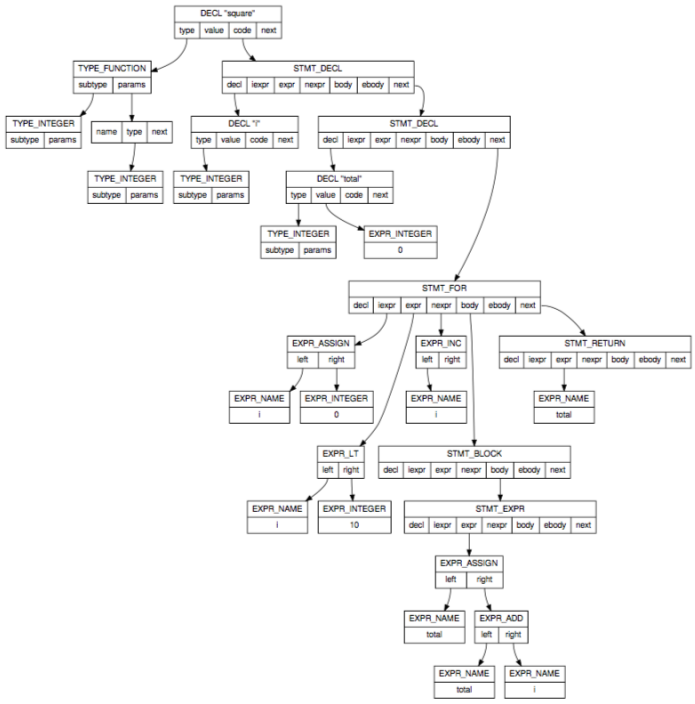


Figure 2.1: Example of an AST of a program

In particular, in my project the nodes of the ASTs can be of three type:

1. **declarations** for functions. The single node of this kind contains these information: symbol associated to the identifier of the function, pointer to the AST of an expression associated (that is the creation of leaf for the identifier), pointer to the AST of the body of the function, pointer to the next declaration;

2. **statement**. Each statement-node must have a kind, that can be a declaration, a call to a function, an expression, an if-else statement or if-elif-else, a while statement, a print, an input function, an end of statement or a block of code; moreover, the structure contains the pointer to a declaration, three pointers to expressions (the first for initialization expression, the second for the expression to be evaluated and the third for the expression for increment), one pointer to the body of the statement, one pointer to the else body and finally a pointer to the next statement;

3. **expression**. One expression can be a sum, a subtraction, a multiplication, one of the six conditional expressions, an assignement or grouping inside brackets and then a variable, an integer, a floating point or a string. So, an expression-node must specify which one of these is, a pointer to the left leaf, a pointer to the right leaf and finally a symbol if it is a variable, an integer value if it is an integer, a floating point value if it is a floating point and a string if it is a string.

The routines associated to the ASTs are the ones to create them, the ones to evaluate them and finally to free the memory allocated to create them.

2.5 Parser using Bison

While Flex recognizes the tokens inside a characters stream, **Bison** takes a grammar specified by the programmer and writes a parser that recognizes valid sentences in that grammar, building the associated *parsing tree*. Note that a *parsing tree* differs from an *abstract syntax tree* because the first can contain "helper" nodes that does not represent necessarily a programming construct, compared to the second one. A Bison program has the same structure as a Flex one:

```
declarations
%%
grammar rules
%%
auxiliary functions
```

While the first and the third part have the same meaning as Flex, in the **rule section** in a Bison program the grammar rules must be defined. The grammar rules are described through the Backus-Naur and an action is associated to each of them:

```
 Grammar rule      {Action}
```

An example of grammar rules for conditional expression is the following:

```
condExpression:
     expression GT expression
     | expression LT expression
     | expression GE expression
```

```
| expression LE expression
| expression EQ expression
| expression NE expression
;
```

Here the **head** of the rule (or the **left-hand side**) is the nonterminal *condExpression* while the **body** of the rule is composed by all the possible sequences of terminals (that is the token GT, LT, GE, LE and so on) and nonterminals (that is expression).

For each grammar rule, the corresponding action in my project creates the specific node of the AST, except for the start symbol *program*, that exit from the program if the parsing process finished successfully, and for the next rule of the start symbol, that evaluates and then frees the AST.

2.5.1 The auxiliary functions section of the parser

In this part of the Bison program of my project, I have defined all the routines declared in the helper header for management of the symbol table, that is the hash function and the lookup function, and the function for the creation of the various nodes of the AST. Beyond this, in this section we can see two important auxiliary functions:

1. the **type checker routine**, that checks if the left-hand and the right-hand types of an expression are equal. This is important for the **semantic analysis** and correctness of the program.

2. the **file-merger routine**. Infact, the evaluation of the AST generates an output file, named `codepart.txt` that contains the body of the code of the target program, and, if a function has been defined, also a file named `funcpart.txt`. Once the evaluation of the AST has been completed, both the general symbol table and the function symbol table are read to generate the files `declpart.txt` and `declpart_function.txt` with the declarations of the variables in C language. All these files must be combined in order to form a complete and correct `.c` file. So that is the job of this routine that is, reading each file line by line in the appropriate order, build the C translation of the Python program.

The functions for the evaluation of the tree and the function to free it are defined in a separate .c file.

2.6 The evaluation of the AST

The function for the evaluation of the AST is declared inside the helper-header file named `sym_ast_helper.h`, defined inside the file `YAcompiler.c` and called by the parser.

The main function is the one that evaluates the statements: it accepts as input one node of a statement, and based on the type of the node it performs a specific action, generating a specific output in C language.

In the case of an expression, an evaluating expression's AST function is called. This latter takes as input a node of the AST of an expression, call itself for the left-hand side of that expression, evaluates it generating the appropriate output, then recall itself to evaluate the right-hand side of the same expression. In this way the routine performs an **inorder traversal** and not a **postorder traversal**.

In the same way are defined the functions for the evaluation of the AST for a user-defined function, but in this case the output is printed into a separate file.

Finally, with the same logic, functions to free the memory dinamically allocated for the AST are defined.

2.7 Type assignment of Python vs. Type declaration of C

In Python it is not necessary to declare a variable and so its type, because during the assignment it takes the type from the right-hand part of the expression. So, in the project, this "problem" has been solved in the same way: during the assignment of a variable, inside the action of the grammar rule, the specific entry of the symbol table corresponding to that variable is updated to make equal the type of the right-hand side of the assignment and the type of that symbol.

2.8 The definition of a user function

In my project I implemented the definition of a user function, without parameters and that does not return anything. As much as it is a very strong restriction, this helped me to understand the way and also the problems in designing a compiler that let the programmer to define functions. Infact, every user-function must have a separate symbol table from the others. As we said, conceptually the symbol table is a map between the name of each identifier and the symbol structure that describes it:

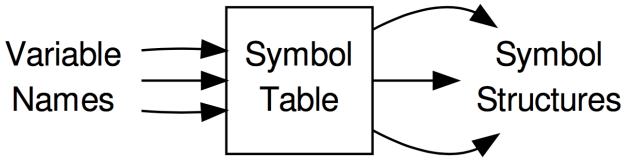


Figure 2.2: Symbol table

The problem occurs when the programmer wants to use the same variable name multiple times, as long as each definition in a distinct **scope** of the various functions. To allow this feature, in the design of a compiler, it is needed to structure the symbol table as a stack of hash tables: in this way each hash table maps the names in a given scope and so it is possible to have a symbol to exist in multiple scopes, without conflict.

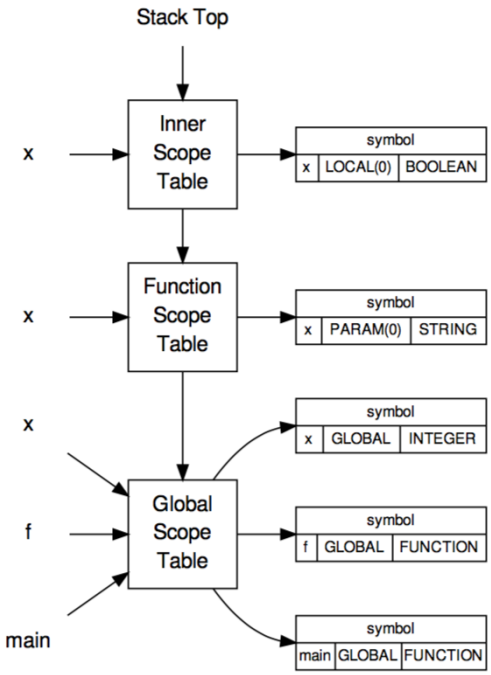


Figure 2.3: Nested symbol table

Therefore, I decided to implement a single additional symbol table to manage the two different scopes (the global and the user-function) as a starting point for a future development of my project.

Chapter 3

Test cases

Firstly, it is important to remark that the function **yyerror()** has been redefined in such a way as to print a more useful error message that indicates in which line there is the error and which kind of error occurred. While syntactic errors are reported through the standard Bison message, the errors of type have been redesigned to indicate if a symbol has no type or if the expression is invalid because its terms have different types. Also that, if a symbol has a not-declared type the compiler warns the user to declare it.

The shell-command to generate the executable of the compiler are the following:

```
bison -d YAparser.y
flex YAlexer.l
gcc lex.yy.c YAparser.tab.c stack.c YAcompiler.c -ll -o
executable.out
```

and to execute the program, assuming to have a source file called, for example, `test.txt`:

```
./exectuable.out < test.txt
```

The target program will be found always in the file `code.c`. It is important to observe that it is necessary, every new compiling, to delete the files `code.c, codepart.txt, funcpart.txt, declpart.txt, declpart_function.txt`, otherwise every output will be appended to the already existing files.

3.1 Valid source program

3.1.1 Test 1

Let us consider the following test file:

```
a=4
x=3
while(x>0):
        if(a==4):
                print "Hello"
                x=x-1
        else:
                print "World"
                x=x-1
d=raw_input("Insert something")
print d
```

the output generated by the compiler is the following:

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
\* DECLARATION PART *\
int a;
char* d;
int x;
\* CODE PART *\
a=4;
x=3;
while (x>0){
if (a==4){
printf("Hello");
x=x-1;
}else{
printf("World");
x=x-1;}
}printf("Insert something");
scanf("%s",d);
printf("%s",d);
```

```
return 0;
}
```

3.1.2 Test 2

Let us consider the following test file:

```
def funzione() :
        x=0
        if(x>=1):
                a=x*x
        else:
                a=x
print "Hello"
b=0
d=b*b
```

the output generated by the compiler is the following:

```
#include <stdio.h>
#include <stdlib.h>

void funzione(){
int a;
int x;
x=0;
if (x>=1){
a=x*x;
}else{
a=x;
}
}

int main(void){
\* DECLARATION PART *\
int b;
int d;

\* CODE PART *\
printf("ciao");
```

```
b=0;
d=b*b;
return 0;
}
```

3.1.3 Conclusions

As we can see the two correct source programs are translated into two correct target programs written in C language.

3.2 Invalid source program

3.2.1 Error-Test 1

Let us consider the following test file:

```
if(x<0)
        print "ciao"
        print d


x=5 + 3.64
```

and the error generated are these two:

```
<Line 1> x has no type.
<Line 2> syntax error, unexpected INDENT, expecting TOKEN_COLON
Parse failed.
```

3.2.2 Error-Test 2

Let us consider the following test file:

```
if (x<0):
        print x


x=a
x=5+3.64
```

and the error generated are these two:

```
<Line 1> x has no type.
<Line 6> Invalid expression: the terms of the expression have different types.
<Line 6> a must be declared.
```

References

[1]Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques, & Tools", 2nd Edition, Pearson and Wesley.

[2]John R. Levine, "flex & bison", 2009, O' Reilly.

[3]T. Niemann, "Lex and Yacc tutorial"

[4]"Python Documentation" *https://docs.python.org/2.7/reference/lexical_ analysis.html*

[5]"Flex Manual" *http://westes.github.io/flex/manual/*

[6]"Bison Manual" *https://www.gnu.org/software/bison/manual/*

[7]*https://www3.nd.edu/ dthain/courses/cse40243/fall2016/*

[8]Lectures notes of the course