# Python Inspired Machines PIM

Rajan Khullar
Tarek Albishara
Mohamad Karaomeroglu

***Abstract* — In our attempt to create a simple Python to C++ translator, we branched Python into a slightly different language. Our translator only supports basic features due to time restraints. Those basic features are variable assignments, printing, and lists. The translator borrows syntax from Python and makes it compliable to machine code.**

***Index Terms* — Python, C++, machine code, translator, parsing, list, dictionary, classes, generics, wrapper, interpreter, compiler, Toy Parse Generator, NYIT**

## I. Introduction

The C++ programming language is very powerful and is widely used by industry. It provides facilities for low level memory manipulation. In addition, it serves as a base for other later popular languages such as C# and Java.

Python is a widely used high level, general purpose, and interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale. [1]

## II. Background

### *A. Python*

The most basic data structure in Python is the sequence. Each element of a sequence is assigned a number its position or index. The first index is zero, the second index is one, and so forth. Python has six built-in types of sequences, but the most common ones are lists and tuples. There are certain things that can be done with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements. [2]

### *B. C++*

In the compiled languages C and C++, the array is the simplest data structure available. Other structures like lists and hash-maps can be included as libraries or the developer could create their own. These languages offer considerably more control to the programmer than interpreted language such as python or java. Perhaps the best example is the fact that all major operating systems are written in C. The operating system is responsible for managing all other processes and serves as the bridge between hardware and software. With the freedom that C gives to the programmer there is also more likelihood of error. For example, in Java and Python there is a garbage collector that takes care of deallocating objects from memory; no such thing exists for C.

## III. Benefits of PIM

### *A. Speed*

Our project of translating Python-like code to C++ stems from the idea of merging easy syntax with fast execution time. Developers can write programs much faster if the programming language is highly readable and understandable. However, these languages usually are interpreted and Python is one of those languages. This fact means that their theoretically a margin of speed up unavailable to Python program.

Compiled languages such as C and C++ offer much more control to the programmer but is not as easy to learn as Python. Using them requires a deeper understanding and more hours of work. For example, simply using a list in C requires custom implementation of a structure. In Python the list is prebuilt into the language interpreter. Therefore, the programmer has less work to do if he or she uses Python.

### *B. Security*

Businesses that sell programs are protective of their source code. The client should not be able to modify the deployed application without the implementer's consent. This feature gives a big advantage for application developing companies, since the client requires some update and modification to the current running software. If the client did not have the source code, they had to contact the business who design the application and submit the requirements for update.

When deploying Python application, the implementer has to supply the source code. With compiled applications using C or C++ the implementer only needs to supply the binary file. This approach will keep the source code safe and secure from copying or editing.

### *C. Academia*

Students and teachers can use PIM as a tool for educational purposes. Student who know Python and are interested in learning C++ can see how their Python code looks in C++. Here

is where PIM plays an important part. The programmer can type they code in Python and converted to C++ using PIM.

## IV. Related Work

The task of porting programs from one language to another is not new. A popular python extension called Cython exists already to do the job of converting Python to C. Cython is open source and falls under the Apache license. It allows complex application to be rapidly developed with the syntax of python, and allows for fast run time since the source code can be compiled to machine code. [3] The biggest limitation factors of Cython are the time it takes to compile from Python to C and the fact that there is a separate build phase. Those limitations, however are standard as any translator would have the same issues. [4]

The first phase of compiling or interpreting a program is parsing. Regular expressions can be fed into a parsing engine such as "Toy Parser Generator" (TPG) or "Yet Another Compiler Compiler" (YACC) [5] to generate the initial parse tree. The regular expressions are helpful to determine correct syntax in a program. This is comparable to Finite State Machines that are used to test if a string belongs to a language. After the parse tree is generated a semantic analyzer would be used to test if the meaning of the program is correct.

## V. Approach

Our task is to implement a translator that is able to convert a subset of Python to C++. At the start of implementation, we wanted to focus on lists, dictionaries, and object oriented-ness. The initial supported variable types would be boolean, integer, and string. The type of each variable should inferenced upon initial declaration and then kept unchanged for the lifetime of the variable.

There were several tasks to accomplish in our project. First we had to all familiarize ourselves with both Python and C++. Later in this report there is an example of a small program written in both languages. Second, we had to finalize our implementation of supported data structures in C++. For this we can opted to build our own list template class.

Perhaps the hardest phase of the project was how to parse input files and extract the logic of the program. "Toy Parser Generator is a lexical and syntactic generator for Python." Through user specified regular expressions TPG creates a parse tree which can be then analyzed by our program. We used TPG to help us generate the initial parse tree which would then be used by the rest of our program.

In our initial design approach (Figure 1) we decided that the translator program should be split up into two components. The first component would read a source python file and extract the program logic in XML format. Then the second component would take the descriptive XML program logic and build C++ files and a Make-file from that. The XML must represent all the classes, functions, variables, and statements of the input program.

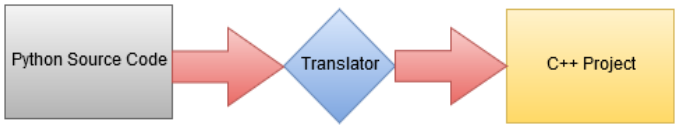


Figure 1: Initial Approach

## VI. Design

To implement our Python to C++ translator we had split our tasks two ways initially. The first problem is parsing a line of source code into some Python object. From a top level point of view, the source would be parsed into a program object. The program consists of variable declarations, and generic statements. Once all the statements are ready they can be printed easily to corresponding code. The next sections cover the entire process in more detail. See Figure 2 for the block diagram of our final design.

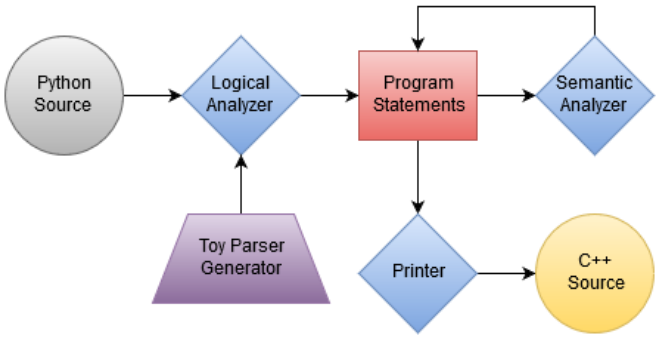


Figure 2: Final Design

### *A. Meta Program*

A Python module was created to store logic features. Instead of using XML from our initial approach we switched to direct python objects instead. The module describes what programs, variables, and statements are. Not only is structure defined in this module, but basic translation is also defined. Each class has to string functions that allows extracted logic to be printed into the C++ output.

The program consists of a list of nodes where each node can be specialized. In Python variable declaration and assignment are intertwined but in C++ the type must be explicitly declared. The two types of variable declarations we implements are simple and list declaration. A simple variable only consists of a type, name, and initial value. The list variable only has type and name. As opposed to our approach we only support two types which are integer and string.

Statement nodes are also specialized. In general a statement is an operation string, and a list of arguments. Depending on the operation the translation string function would have to act

appropriately. For example a print statement and an assignment statement would look very different from each other.

### *B. Logical Parser*

The source Python-like code needs to be parsed into a program object. Using Toy Parser Generator, we created a module that handles extracting one line of source code into one statement object. The key component here is regular expressions. First we defined separators [spaces, tabs, etc.] and tokens. For example one token is natural number which has \d+ for a regular expressions. A natural number is one more digits. Other tokens defined including variable identifiers, strings, and the print keyword.

The language to parse is a context free grammar. Using nonterminal and terminal symbols we designed each grammar rules. For example START ← PRINT | ASSIGN. This rule indicates the string to parse is either a print statement or an assign statement. Please read the grammar rules in the Appendix to see the rules we created. The convention we followed was that terminal symbols are lowercase and nonterminal are uppercase.

Some tokens we used are the '=' and '<<' strings. The first is used for assignment statements which is common in most programming languages. The latter it used to append items to a list. This feature is actually borrowed from C++ and is not implemented in Python itself. The append grammar rule indicates that an append statement is a variable identifier followed by the append token and some expression. The rule is as followed: APPEND ← var append EXPR.

### *C. Support Modules*

Besides the two modules described above there two other python files as well. A bridge module wraps some of the functionality from the Meta Module so the Logical Parser can use it easier. The Phase Module is close to the top module in the sense that it reads a source file and prints the corresponding C++ statements line by line. Some semantic analysis is also done to make sure that variables are declared before they are used in assignments.

### *D. Top Program*

The final entity of our project is a Linux bash script. The program takes two terminal arguments. The first argument provides the path of the source Python[ish] code. The second parameter the directory to create the C++ project in. This entity creates the new user specified directory and copies three files. Those files are one C++ template, the list header, and one Makefile. The main support modules is invoked and its output gets directed to the template file. Finally the script closes the new C++ file with braces and the file is ready to compile. Then the user can use the Makefile to easily compile the program and generate a target binary file.

## VII. Future Work

The translator can be further developed to convert code from Python to C instead of C++. Even though c is not an object oriented language it still supports structures. Each class in an object oriented language can be broken down into two different structures. One structure would actually hold the class data such as a box's dimensions. Another structure would point to the class functions such as constructing, printing, and calculating a box's volume. Please see the Appendix for the examples.

## VIII. Acknowledgment

We would like to thank the creator of Toy Parser Generator, Christophe Delord, for providing a way to create a new programming language with Python. We would also like to thank NYIT Engineering and Computing Sciences faculty for creating a research culture for students.

## References

[1] "Welcome to Python.org." *Python.org*. Web. 02 Mar. 2016.

[2] "Python Lists." *Www.tutorialspoint.com*. Web. 23 Mar. 2016.

[3] "Cython: C-Extensions for Python." *Cython: C-Extensions for Python*. N.p., n.d. Web. 23 Mar. 2016.

[4] "Cython: The Best of Both Worlds." *IEEE Xplore*. N.p., n.d. Web. 23 Mar. 2016.

[5] "Yacc: Yet Another Compiler-Compiler." *Yacc: Yet Another Compiler-Compiler*. N.p., n.d. Web. 06 May 2016

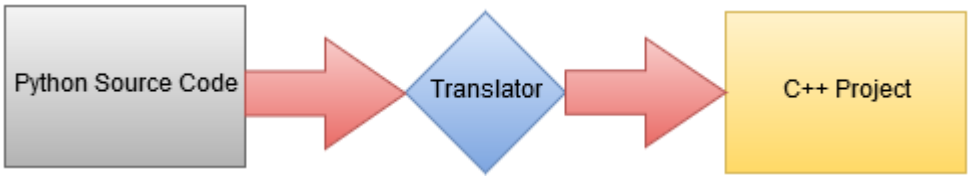
## APPENDIX 1

### GRAMMAR RULES

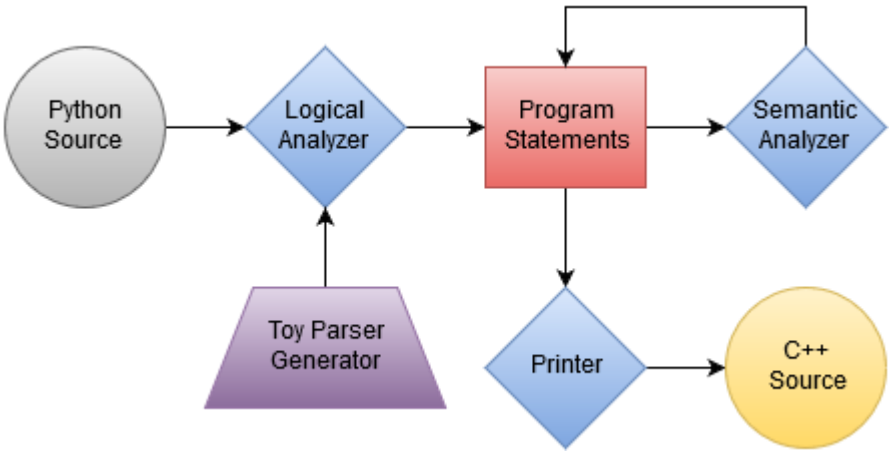
```
separator       spaces  :       '\s+'
token           natural :       '\d+'
token           string  :       '\'[a-zA-Z0-9_\s]*\''
token           output  :       'print'
token           assign  :       '='
token           append  :       '<<'
token           var     :       '[a-zA-Z_]+'
token           type    :       '\$[a-zA-Z_]+'
token           add     :       '[+-]'

START   ->      LIST | PRINT | ASSIGN | APPEND
PRINT   ->      output EXPRL
LIST    ->      var assign type '\[\]'
APPEND  ->      var append EXPR
ASSIGN  ->      var assign EXPR | var '\[' natural '\]' assign EXPR
EXPRL   ->      EXPR ( ',' EXPR )*
EXPR    ->      natural | string | var '\[' natural '\]' | var
```

# Appendix 2

## Design Approach




## Final Design



## APPENDIX 3

### BOX EXAMPLE

***box.py***

```
class box:
    def __init__(self, w, h, l):
        self.width = w
        self.height = h
        self.length = l

    def __str__(self):
        return "this is a box"

    def volume(self):
        return self.width * self.height * self.length


if __name__ == '__main__':
    b = box(1,2,3)
    print b
    print b.volume()
```

***box.cc***

```
#include <iostream>
#include <string>

using namespace std;

class box
{
    public:
      box(double w, double h, double l)
 : width(w), height(h), length(l)
        {}

        double width;
 double height;
 double length;

        double volume(){return width * height * length;}
        string output(){return "this is a box";}
};

int main()
{
    box b = box(1, 2, 3);
    cout << b.output() << endl;
    cout << b.volume() << endl;
    return 0;
}
```

APPENDIX 3 (CONT.)

BOX EXAMPLE - EXTENDED

***box.h***

```
#ifndef BOX_H
#define BOX_H

typedef struct box box;
typedef struct class_box class_box;

struct box
{
 int width, height, length;
};

struct class_box
{
    box* (*init)(int w, int h, int l);
 int  (*volume)(box *self);
    void (*print)(box *self);
};

extern class_box* class_box_load();

static box* box_init(int w, int h, int l);
static int box_volume(box *o);
static void box_print(box *o);

#endif
```

***box.c***

```c
#include <stdio.h>
#include <stdlib.h>
#include "box.h"

extern class_box* class_box_load()
{
 class_box* kls = (class_box*) malloc(sizeof(class_box));
 if(kls == NULL) exit(1);
 kls->init = &box_init;
 kls->volume = &box_volume;
 kls->print = &box_print;
 return kls;
}

static box* box_init(int w, int h, int l)
{
 box* o = (box*) malloc(sizeof(box));
 if(o == NULL) exit(1);
 o->width = w;
 o->height = h;
 o->length = l;
 return o;
}

static int box_volume(box *o)
{
 return o->width * o->height * o->length;
}

static void box_print(box *o)
{
 printf("This is a box. %d\n", o->width);
}
```

APPENDIX 3 (CONT.)

***target.c***

```c
#include <stdio.h>
#include <stdlib.h>
#include "box.h"

int main(int argc, char *argv[])
{
 class_box* box_kls = class_box_load();

 box* b1 = box_kls->init(1,2,3);
 box_kls->print(b1);

 int v = box_kls->volume(b1);
 printf("Volume: %d\n", v);

 free(b1);
 free(box_kls);
 return 0;
}
```