Suleyman Demirel University

Project Report

Compiler from Java to Python

Prepared by:
Boranov Rustam
Tazhimbetov Elnur
Shokubassov Amir
Pan Danil
Abylkairov Marat

Kaskelen 2019

Abstract

This report describes the method of creating a compiler. A program that translates a programming language from one language to another. The report consists of 3 parts: introduction, body and conclusion. The introduction says more about the purpose of the project and what is a compiler. The body says the basic methods for creating a compiler. The conclusion says our findings and analysis. The report is intended for people who know formal languages and automata and who want to write their own compiler.

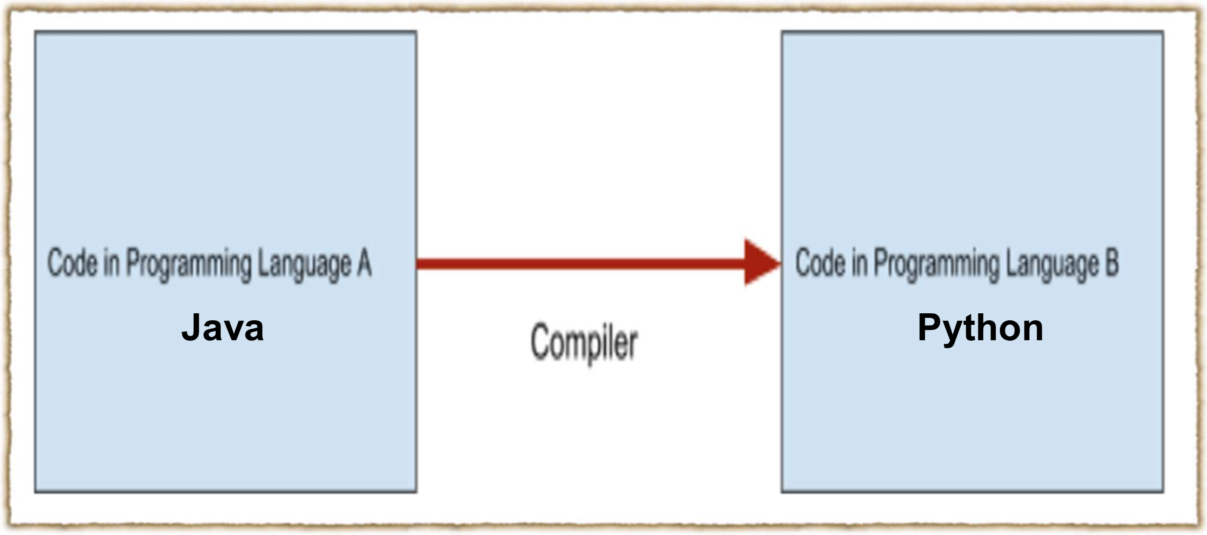
Key Words: compiler, token, parsing, abstract syntax tree, generation code, Java, Python, frontend, backend

Content

Introduction

A **compiler** is a computer program that translated computer code written in one programming language (the source language) into another programming language (the target language).In our case source is Java language while target is Python language.

The aim of our project was to write a small compiler that would take such simple operations as **assignment**, **if** condition and loop (in our case **while**) from Java and translate them into equivalent code in Python.


Code in Programming Language A
Java
Compiler
Code in Programming Language B
Python

Structure of our compiler consist of 2 main stages: **front end** and **back end**.

Front end includes lexical analysis and syntax analysis, it means tokenization and parsing of our source language (Java).

Back end includes code generation, it means creation code of our target language (Python).

To create a compiler, we had the following tasks:

1) Know the source and target languages (Java and Python)
2) Know the formal languages and automata (I.e. Context free grammars)
3) Write a input code in Java
4) Write a compiler itself using 2 main stages
5) Get an output code and check for correctness

Main body

Tokenizer

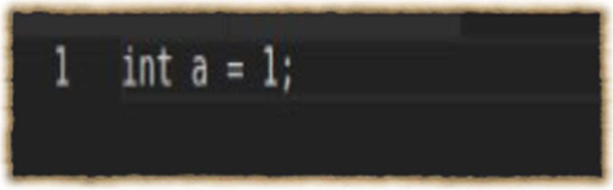
Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing.

Token - each element of our input code.

So, in this step we divide our input code into tokens that have value and type. For example we got «+» from input code Java, so we consider this token as type 'arithmetic' and value '+'.

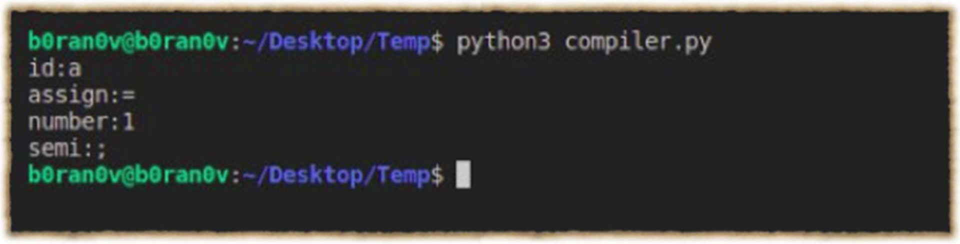
Clear demonstration of **assign** operation:

Our input in Java ->



```
int a = 1;
```

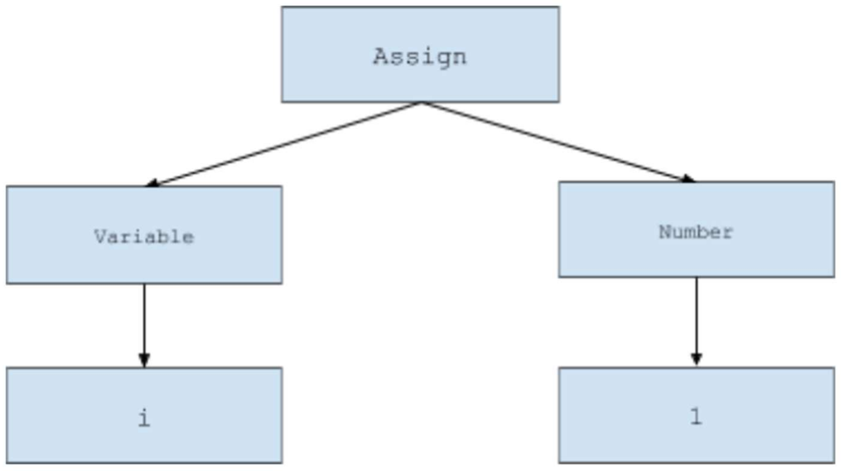
Output in Python ->



```
b0ran0v@b0ran0v:~/Desktop/Temp$ python3 compiler.py
id:a
assign:=
number:1
semi:;
b0ran0v@b0ran0v:~/Desktop/Temp$ 
```

As you can see each element of input code, was analyzed and distributed.

Here is **variable** has same meaning as **id**



In our code we have such tokens as:

- Parentheses
- Brackets
- Conditional operators
- Arithmetic Operators
- String
- Number
- Semicolons

Most of them are identified by checking the symbols, some of them are identified by **regular expressions**. For instance, numbers are identified by regular expression ‘R = [0-9]+’ , which means that if there is one or more digit, then it is a number token.

Shortly saying, it is the basic part of compiler, but the important one. The tokens are collected and transformed into bigger structure called **AST**, where group of tokens form statement. Usually, simple statements are made up from 3 to 10 tokens. But sometimes there are complex statements, which can have hundreds of tokens.

Parsing

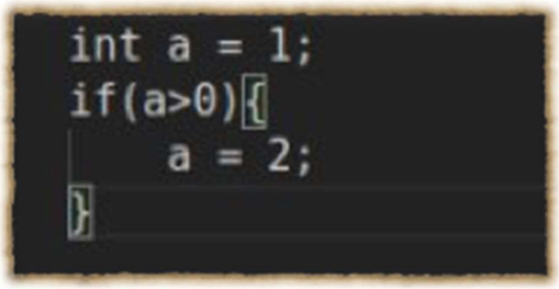
Parsing is the process of analyzing a string of symbols, either in natural languages, computer languages or data structures, conforming to the rules of a formal grammar.

After tokenization of our input code, we must do parsing step. For our compiler it means creation of **abstract syntax tree**.

AST is a tree representation of the abstract syntactic structure of source code written in a programming language.

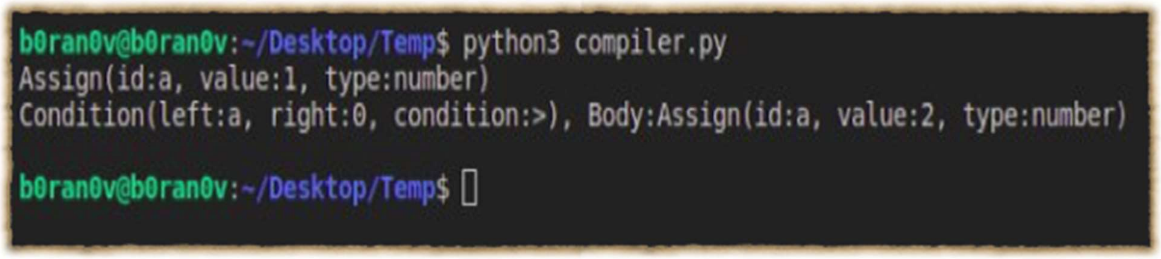
Clear demonstration of **assign** and **if** condition operations:

Input ->



```
int a = 1;
if(a>0){
    a = 2;
}
```

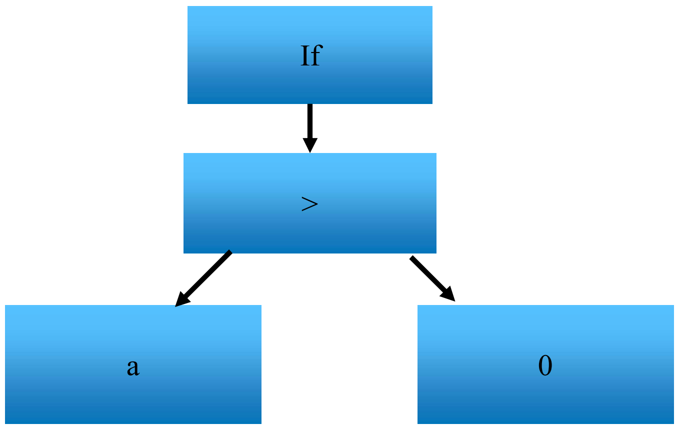
Output ->

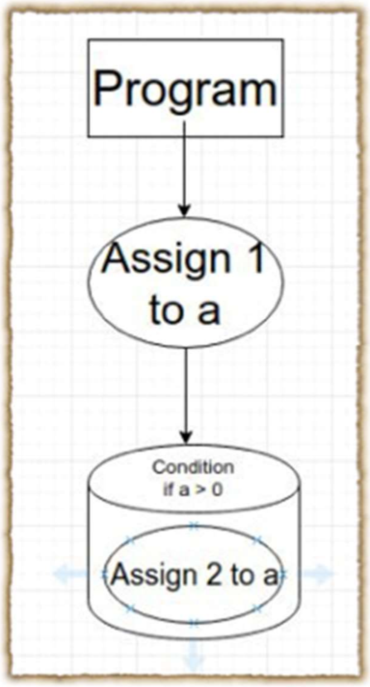


```
b0ran0v@b0ran0v:~/Desktop/Temp$ python3 compiler.py
Assign(id:a, value:1, type:number)
Condition(left:a, right:0, condition:>), Body:Assign(id:a, value:2, type:number)

b0ran0v@b0ran0v:~/Desktop/Temp$ 
```

So, the if condition step also can be represented like this:





In our code, the parser performs the main job. It collects tokens into bigger structure called **statement**, as it was said at the previous part. Then statements are collected into AST.

You may ask how we determine at which token statement must end? At this point, we can answer that we build up our parser looking to Java language syntax. Every statement at java ends with the semicolon. So, when our parser meets the semicolon token, it automatically collects all the previous read tokens and forms one statement. At some cases, when condition and loop tokens, such as if, else, while occurs, it looks for the right brackets, because the conditions and loops are finished with this token.

Also, there might be cases when conditions and loops can contain other statements in their own body. At these situations, here comes the recursion. It helps us to obtain the inner statements from the main statement. And when we build AST, these statements don't occur at the main list of statements, but if we open the body, we can see that inner statements are also attend and when we convert our AST, they are also can be converted.

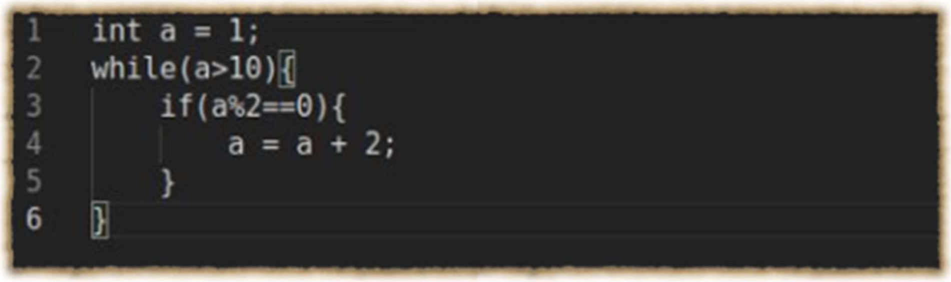
Code Generation

Code generation is the process by which a compiler's **code generator** converts some intermediate representation of source code into a form that can be readily executed by a machine.

It's last step in our compiler, our **AST** will be input for our **code generator.**

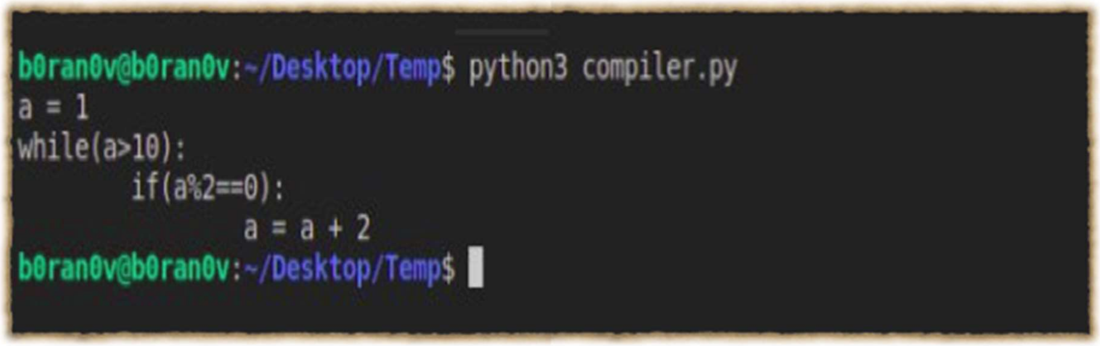
Clear demonstration of full working process:

Input ->



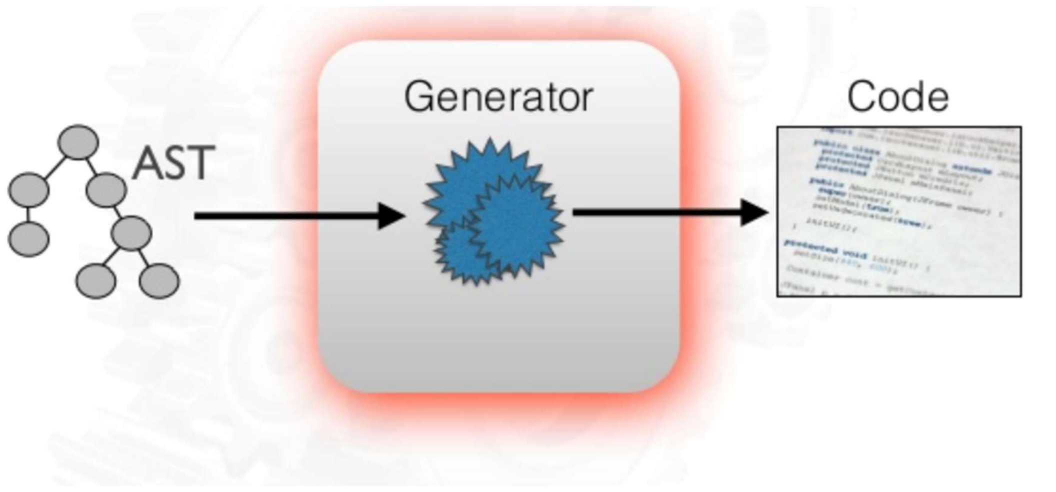
```
int a = 1;
while(a>10){
    if(a%2==0){
        a = a + 2;
    }
}
```

Output ->

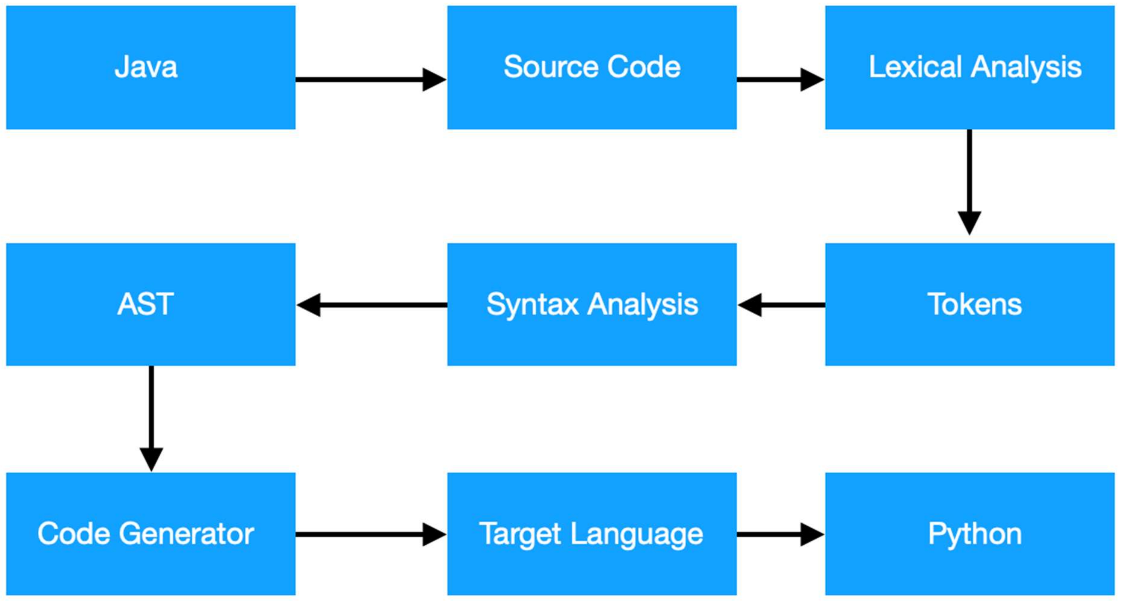


```
b0ran0v@b0ran0v:~/Desktop/Temp$ python3 compiler.py
a = 1
while(a>10):
        if(a%2==0):
                a = a + 2
b0ran0v@b0ran0v:~/Desktop/Temp$ 
```

Our output can be easily run and work on real Python IDE.



Full demonstration in diagram



Conclusion

So, what we have now? After we have written the compiler, we started to understand how simple computers work and how to convert one language into another. By this practice we came to point that writing compiler is neither easy nor hard.

Why it is not hard? Because if you follow the steppes written above, you can construct simple compiler, which you can use for some cases when you, for example, create your own programming language and you want to test or debug it.

And why it is not easy? Because constructing compiler requires big patience, much time and nerves. You must test every step every time when error occurs. Also there are some facts like following:

- More symbols – more possible tokens
- More tokens – more possible statements
- More possible statements – complex syntax tree

At present days some programmers think that coding is enough to be qualified specialist but understanding how the computer work is the must-have unit for every computer engineer. To construct fully working compiler we need to know the basics of finite and infinite automata, learn how to deal with regular expressions.

It must be not hard to select source and target language. You only have to know the syntax of the languages. As we did, you can first try to translate only statement from main part of source language script. Then, it can become more complex in order to make your compile better.

To sum up, we want to say that this project helped us fully realize our potential at building helpful software, also we gained more experience at working in a team project.

References

1. **About Compiler**
https://en.wikipedia.org/wiki/Compiler

2. **About AST**
https://en.wikipedia.org/wiki/Abstract_syntax_tree

3. **About Code generator**
https://en.wikipedia.org/wiki/Code_generation_(compiler)

4. **About how to write a program**
https://www.youtube.com/watch?v=Tar4WgAfMr4
https://www.youtube.com/watch?v=eF9qWbuQLuw
https://www.youtube.com/watch?v=LDDRn2f9fUk

5.**About design**
https://www.tutorialspoint.com/compiler_design/
https://www.geeksforgeeks.org/compiler-design-tutorials/