Python - Java Transpiler

Zanzonelli Dario Amedeo
Servedio Giovanni

Formal Languages and Compilers

Master's Degree in Computer Science Engineering

Politecnico di Bari

A.Y. 2020/2021

Index

Transpiler

A “transpiler” (**trans**late com**piler**), usually referred to as source-to-source compiler, is a specific type of translator, which (generally) works with high abstraction level programming languages, both in input and output.
In fact, it translates a source code to another programming language, having about the same level of abstraction. This is what distinguish a transpiler from a “simple” compiler.

Transpilers have caught on since the release of Javascript and the diffusion of Web applications. The possibility of porting entire software suits to websites, without knowledge of Javascript and web languages, made them common among programmers and encouraged their developement.

Main examples of transpilers could be: Emscripten, TypeScript, CoffeeScript.

To stay in the same scope of this work, it is possible to mention “Jython”, a Python to Java bytecode translator. Although no longer supported, Jython worked, in previous versions, also as a translator from Python source code to Java source code.

Python

Python was born in 1991 and was released as a high level, general-purpose language.

It is a simple, flexible and dynamic language, which supports object oriented, structured programming and many characteristics of functional programming.

Its uses range from distributed applications to system testing, from scripting to numerical computation.

Although variables are not typed, Python is strongly typed during runtime.

This contributes to make Python very readable and similar to pseudo-code.

Java

Java, released in 1995, is a is general-purpose, class-based, object-oriented language, intended to let developers “write once, run anywhere”. Applications are typically compiled to bytecode, able to run on any machine that has a Java Virtual Machine independently from machine architecture.

Its quirks are simplicity as an object-oriented language, robustness and security. It has been thought to be as architecture neutral as possible, threaded and dynamic.

Its extreme portability is archived by compiling java code into an intermediate language called Java bytecode analogous to machine code. Bytecode instructions are intended to be executed by a Virtual Machine (JRE) specifically written for the host hardware.

Its syntax is largely influenced by C++ and C: it varies from these two since it was built exclusively as an object-oriented language.

All its qualities make this language one of the most flexible all around, and most of its criticism is directed to its (relative) higher needs of memory and computational power.

Lexical Analyzer

The primary role of the lexical analyzer consists in an abstraction process, which tries to transform characters in tokens. In doing so, it distinguishes, also, keywords, delimiters, operators and identifiers.

It is important to notice that keywords do not carry any information other than their names. In fact, they represent a token as-is in the parser.

Identifiers are strings belonging to Python syntax. They are associated to Integer values or String values.

The lexical analyzer is built with FLEX, and coded in the file “scanner.l”.

FLEX produces a C program without main, whose entry point is the function *yylex()*.

The *YYSTYPE* constant is set to be “*char const* *”.

The *yyin* file is set through argument, when running the transpiler from command line.

Scanner macros are shown below.

```
tab             [\t]
tabS            [ ]{4}
ws              [ ]
letter          [a-zA-Z_]
digit           [0-9]
number          (-)?{digit}+
id              {letter}({letter}|{digit})*
qt              ["]
sqt             [']
inr             (in{ws}+range)
string          {qt}.*{qt}|{sqt}.*{sqt}
```

Identifiers (id) follow the Python syntax. They can contain letters, numbers and underscores, but cannot start with a number.

Numbers (int) can be negative (starting with “-“) and can contain one or more digits.

Strings can contain any character delimited by quotes (“…”) or single quotes (‘..’).

Scanner rules and tokens are shown below.

```
"("         {return LP;}                    "if"        {return IF;}
")"         {return RP;}                    "else"      {return ELSE;}
"["         {return LS;}                    "input"     {return INPUT;}
"]"         {return RS;}                    "print"     {return PRINT;}
":"         {return COLON;}                 "def"       {return DEF;}
{tab}       {return TAB;}                   ".append"   {return APPEND;}
{tabS}      {return TAB;}                   ".clear"    {return CLEAR;}
{inr}       {return INR;}                   ".count"    {return COUNT;}
"for"       {return FOR;}                   ".extend"   {return EXTEND;}
"<"         {return LESS;}                  ".index"    {return INDEX;}
">"         {return MORE;}                  ".insert"   {return INSERT;}
"=="        {return EQUALS;}                ".pop"      {return POP;}
"!="        {return DIFFER;}                ".remove"   {return REMOVE;}
"+"         {return PLUS;}                  ".copy"     {return COPY;}
"*"         {return MULT;}                  ".reverse"  {return REVERSE;}
"/"         {return DIV;}                   ".sort"     {return SORT;}
"\n"        {return END;}                   "%"         {return PERCENT;}
"="         {return ASSIGN;}                "return"    {return RETURN;}
{id}        {yylval = strdup(yytext); return ID;}
{number}    {yylval = strdup(yytext); return NUM;}
"-"         {return MINUS;}
{string}    {yylval=strdup(yytext); return STRING;}
```

Parser

The parser is an abstract machine whose role is to group input according to grammar rules.

Such machine is generated through the usage of Bison and its bottom-up parsing technique, known as LALR(1).

The parser is coded in the "*parser.y*" file. It is set to work together with FLEX. The parser, also respects C syntax and code rules.

Tokens

Parser tokens are shown below.

```
%token END TAB
%token ID
%token ASSIGN
%token STRING NUM
%token LESS MORE EQUALS DIFFER AND
%token PLUS MINUS MULT DIV
%token IF ELSE
%token INR FOR
%token RP LP LS RS COLON
%token INPUT PRINT
%token APPEND CLEAR COUNT EXTEND INSERT INDEX POP REMOVE
%token DEF RETURN
%token PERCENT
```

Grammar

The used grammar is a CFG and is shown below.

```
definition:
|definition DEF ID LP decoration RP COLON END
|input

decoration:
|ID decoration

input:
| input line
```

```
line: END
| tab
| tab IOoperation
| tab declaration
| tab list
| tab selection
| tab ELSE COLON
| tab iteration
| tab RETURN type

tab:
| TAB tab
declaration: ID ASSIGN STRING
| ID ASSIGN NUM
| ID ASSIGN math_exp
| ID ASSIGN function_call
| ID ASSIGN ID

selection: IF LP condition RP COLON
| IF condition COLON

iteration: FOR type INR LP type RP COLON
IOoperation: ID ASSIGN INPUT LP RP
| ID ASSIGN INPUT LP type RP
| PRINT LP txt RP
```

```
list: ID ASSIGN LS RS
| ID APPEND LP type RP
| ID CLEAR LP RP
| ID COUNT LP RP
| ID EXTEND LP type RP
| ID INDEX LP type RP
| ID INSERT LP type type RP
| ID POP LP RP
| ID POP LP type RP
| ID REMOVE LP type RP
| ID COPY LP RP
| ID REVERSE LP RP
| ID SORT LP RP

function_call: ID LP RP
| ID LP passing_vallues

passing_vallues:  type
|type passing_vallues

txt:type
|type txt

condition: type
| type logic_op type
| type PERCENT type EQUALS type
| condition condition

type: ID
| NUM
| STRING
| list
| function_call
```

```
math_exp: type math_op type
|   LP math_exp RP

logic_op: EQUALS
| DIFFER
| LESS
| MORE
| AND

math_op: PLUS
| MINUS
| MULT
| DIV
```

Functions

The functions and the methods used in the grammar rules are shown below.

```
Function: char* decl(char* ,char*, char*,int );
description:
     Adds non-existing elements to the symbol table. Modifies Existing
     elements in symbol table if redeclared in different scope. Outputs
     java string of an element declaration.
input:     name: Name of the variable to declare.
      type:     type of the variable to declare.
      value:    value of the variable to declare.
     currentScope:   current scope number.
output:
     String containing translated declaration of element in input.

Function: char* build_if(char*);
description:
     Builds if string in java having in input condition.
input:
     cond: String, condition of if.
output:    String containing translated if in java.
```

Function: char* build_for(char*,char*);

description:

Builds java for string given iterable variable and condition variable condition is always LESS THAN

input:

iter: String, iterator name.

cond: String, condition element.

output:

String containing translated for in java.

Function: char* build_mExp(char* l,char* c,char* r)

description:

Builds strings of mathematical expressions.

input:

l: Left Operand

c: Condition

r: Right Operand

output:

String of mathematical expression.

Function: char* build_IOop(char*,char*);

description:

Builds strings for combined Input\Output operations

input:

id: element in which to put java input function.

strout: string containing output function with text in java.

output:

String of combined output, input in java.

Function: char* build_IOop(char*,char*);

description:

Builds strings for output operations.

input:

outs: String of element to print.

output:

String of combined output in java.

Function: char* myStrCat (char* ,char*);

description:

costume string concatenation.

input:

l: Left element.

r: Right element.

output:

String of concatenated element.

Function: char* buildFCall(char* , char*);

description:

Builds a string containing function call with respective decorators.

input:

name: string of Function name.

values: string of function decoration.

output:

String of function call in java.

Function: char* getFName(char*);

description:

gets the function name from a python function call.

input:

fun: string containing python function call.

output:

String, name of the function.

```
Function: char*  build_odd_cond(char* , char* ,char* );
description:
     builds if condition with % element.
input:
     element: string containing Element to divide.
     divisor: string containing divisor value.
     equals: string containing equals condition element.
output:
     String of complete (X %% Y == Z ).

Function: void addTipo( char* ,char*,int );
description:
     Adds or modifies element to symbol table.
input:
     var:  String of element name.
     tipo: String of element type.
     currentScope:   int value of current scope id.
output:
     none.

Function: int pClosure( int,int);
description:
     get the number of parenthesis needed.
input:
     otab: int value of number of tabs in precedent line.
     tab:  int value of number of tabs in current line.
output:
     int, number of parenthesis to close.
```

Function: char* buildClosures (int,int);

description:

builds a string of n closed parenthesis.

input:

otab: int value of number of tabs in precedent line.

tab: int value of number of tabs in current line.

output: string of closed parenthesis.

Function: char* buildDecoration();

description:

This function is used to build the decoration of a function using a linkedList.

input:

none.

output:

string of java function decorators with corresponding types.

Function: char* buildFDecleration(char*,char*,char*);

description:

This Function builds the string of the java function declaration input:

input:

type: String of function type.

name: String of function name.

decoration: String of function decoration.

output:

string of java function declaration.

Function: void printOnF (char*);

description:

this function prints on file opens and closes buffer. on input requires file name.

input:

name: String of file name.

output:

none.

Function: void loadPopToBuffer();

description:

This function loads into the buffer a java method to emulate the python pop function.

input:

none.

output:

none.

Function: void loadCopyToBuffer ();

description:

This function loads into the buffer a java method to emulate the python copy function.

input:

none.

output:

none.

Global variables

The used global variables are shown below.

`int tab = 0;`	Tab counter.
`int otab = 0;`	Tab cvalue of precedent line.
`int end = 0;`	Flag for empty line.
`char* currentFType;`	Type of current function.
`int currentScope=0;`	Current scope used for declaration disambiguation in distinct functions.
`int in_line = 0;`	Inline flag.
`int returnFlag = 0;`	Return flag.

Flags

The global flags used are shown below.

```
int loseCodeFlag =0;                    Reports the presence of loose
                                        code.

int mainExistanceFlag = 0;              Reports the presence of a main
                                        function.

int emptyStart =1;                      Reports the presence of white
                                        spaces in the beginning of the
                                        code.

int listFlag = 0;                       Reports the presence of at least
                                        one list in the code.
```

Auxiliary headers

To keep the code clean and (relatively) simple, four auxiliary headers were created.

Symbol table

The symbol table is the main auxiliary structure used by this work. It contains all references to all identifiers used in the translated code. Each time an identifier is detected from the lexical analyzer, its presence in the symbol table is checked. If not found, the lexeme is added, otherwise the reference list of the same lexeme is updated. A lexeme is defined as follows.

```
struct symbol {

    char *name;                         Lexeme itself.

    char *tipo;                         Type of the correspondig
                                        variable.

    int scope;                          Scope in which le variable is
                                        located.

    struct ref *reflist;                List of all references.

};


struct ref {                            Reference list.

    struct ref *next;                   Next Reference.

    int lineno;                         Line number of current reference.

};
```

Symbol table functions are shown below.

`struct symbol *lookup(char*, char*,int);`	Checks for existence of element otherwise adds element on symTable and returns element.
`void addref(int, char*, char*,int);`	Adds reference to the reference list, checks availability of memory. Uses lookup function.
`static unsigned symhash(char *sym);`	Translates sym to hash.
`static char* getType(char*);`	Returns the type of the associated element having on input element name.
`static int exist(char*);`	Checks existance of element having on input element name.
`static int modtype(char*,char*,int);`	Modifies the type of an element if it exists on the list.
`static int getScope(char*);`	Returns the scope of an element if it exists.
`static int  setScope(char*,int);`	Modifies the scope of an element if it exists.

The hash function is:

```
while (c= *sym++) hash = hash*9 ^ c;
```

List_op.h

This header implements methods to construct Java lists syntax. Its main use is code readability.

String_buffer.h

This header contains methods to build the structure of the translated code through a data structure (linked list). For every function to be translated, it is initially empty. It is filled with instruction statements and finally declarations, containing variable types.

When the translation is finished, the function is printed.

At this point, the data structure is cleared.

Linklist.h

This header contains a linked list data structure.

Its aim is to store functions variables identifiers, in order to decode their types.

Once this task is complete and the function is translated, its variables identifiers are returned.

List Implementation Details

This work implements a translation of the python list in Java. *ArrayList* was chosen as direct translation because it allows the access to list elements through indexing. However, some methods natively available in Python do not have a direct translation in Java and had to be manually added.

These methods are in the form method(parameter), where parameter is the list, instead of list.method(). The flag *listFlag*, set to true if a list is present in the source code, determines if the methods definitions must be printed in the bottom of the files.

The methods *sort()* and *reverse()* need the Java Collection library to be imported. This library is always included.

The cited methods are:

list.pop(index) → *pop(list, index)*

list.reverse → *reverse(list)*

list.sort → *sort(list)*

Python Scripting Paradigm

Python allows the usage of object-oriented paradigm and scripting paradigm. This work can translate python code to java, comprehensively of both paradigms.

The only constraint present consists of the scripting paradigm code mandatorily being in the beginning of the .py file.

The transpiler inserts the code in a function returning void `(void loose_code()).`

If a main function exists, it will call the *loose_code()* function as first instruction, and then continue with its own instructions.

If a main function does not exist, it will be automatically created and its only instruction will be a call to the *loose_code()* function.

Scope Management

This work contemplates the redeclaration of variables.

Every element involved in a definition, call, assignment is added to the symbol table, taking into account its type and its scope.

When a variable is re-declared (in another function, for example), the transpiler looks for its identifier in the symbol table. If found, the transpilers checks the integer identifying the scope. If a mismatch is found (to be precise, if the found integer is minor than the current), the transpiler treats the identifier as a variable declaration.

Type Check Difference

Java and Python have difference behaviors during type checking. Python checks types only during runtime. This implies that no type is specified during the coding phase and that the transpiler has to deduce it.

The bottom-up approach of the parser allows to deduce the type, together with the assumption that parameters and return value have to be used in the function body.

The deduction process takes place in the symbol table.