POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Work Project

FORMAL LANGUAGES AND COMPILERS

Php2Python

Teacher:
Prof. Ing. Floriano Scioscia

Team members:
Calò Federica
Castiglia Giovanni

Academic Year 2021/2022

Index

Introduction

The aim of this project is the development of a **transpiler** for translating code from **PHP** programming language (source) to **Python** programming language (target). The goal of our transpiler is to check the correctness of all instructions and entirely translate them in the target program if no errors occur (lexical, syntax, semantic). The project started in October 2022 and ended in January 2023 and the team members equally and simultaneously worked on the project via remote working. The transpiler includes the main phases of analysis, which includes the lexical, syntax and semantic analyzers, and synthesis for the object code generation.

We developed our transpiler using C language and the scanner generator Flex (Lexical analysis) and the parser generator Bison (Syntax analysis). We developed a file with *.l* extension (*php2python.l*) to implement the lexical analysis, in which regular expressions and associated actions were described. This is given in input to Flex that produces a scanning routine in the file *lex.yy.c* that looks for occurrences of regular expressions matching the patterns defined in the lexer file. If one is detected, the associated C code is executed.
We generated a file with *.y* extension (*php2python.y*) to implement the syntax analysis and it is used as input for Bison, that executes the code associated with productions when one of them is recognized and that then gives as output the file *php2python.tab.c*, that includes the parser routine, and the file *php2python.tab.h*, that contains the definitions of tokens.
Using Bison, we process a grammar described as LALR(1), a simplified version of the canonical grammar LR(1), where the second L means that the **input is read from Left to right**, i.e, from the start of the file (or line) to the end. The R means that the parser produces right-most derivations. **Right-derivation parsers are "bottom-up"**, i.e, items are read and combined into bigger units which are then reduced to a value.

In order to keep track of the variables defined by the user though the code wrt specific scope/context and then check possible reference errors (i.e a variable is called without being initialized/declared) we defined a symbol table to store references of variables with their specific context. To define our symbol table as a hash table, we used the **uthash** C library.

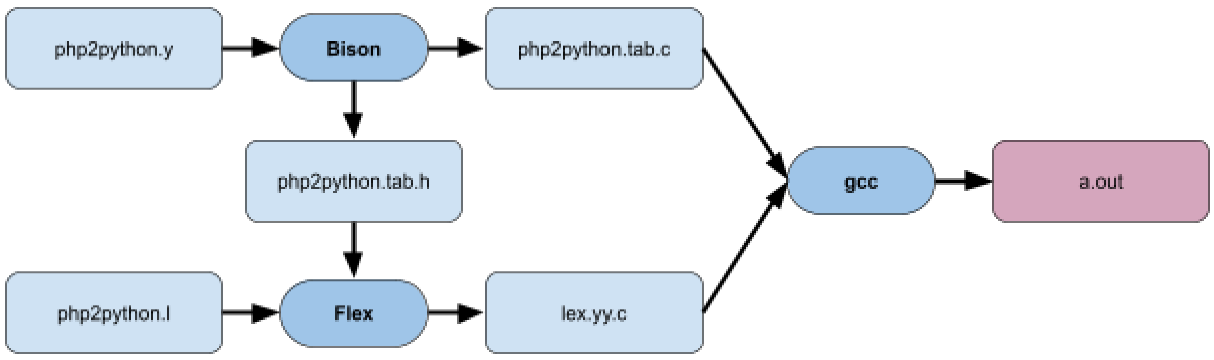
An overview structure of our project directory is reported below:

File name	Objective
php2python.l	Input file for Flex (lexical analysis)
php2python.y	Input file for Bison (syntax analysis)
symtable.h	Symbol table routines defined by utash library
valuesmemory.h	Support routines for content storage
gencode.h	Routines for the translation process
output_translated.py	Output Python file derived from the translation
abstract_tree.txt	Simple representation of the abstract syntax tree

How to build and execute our transpiler

In order to build and execute our transpiler, you need to follow this steps:

1. Create into the project folder a text file with extension .php containing PHP code (we also support the presence of HTML/JS/CSS code outside the PHP definition <?php … ?>)
2. Open the terminal and move to the project folder
3. Run the command **bison -d php2python.y**
   (**Note:** the -d "define" option allows definitions to be transferred to different header file php2python.tab.h , in order to be included in other modules, such as the lexer file)
4. Run the command **flex php2python.l**
   (**Note:** this execution produces the lex.yy.c file, useful in next command)
5. Run the command **gcc php2python.tab.c lex.yy.c -lfl**
   (**Note:** we use the standard gcc compiler. Also file tab.c is generated after the bison command. The -lfl option allows to obtain an executable file a.out)
6. Run the command **./a.out < yourPHPscript.php**
7. Open the file output_translated.py to see the translated script (if no errors occur)
8. Open the file abstract_tree.txt to see a representation of the AST (Abstract Syntax Tree)



Source language restrictions

We defined the following restrictions for the source language PHP. These define the data types, rules and constructs that are supported by our translation process.

Data types and structures	Integer, boolean, string, array of prev. defined data types (also mixed content)
Operators	Addition, subtraction, multiplication, division, modulus, equal, not equal, greater, greater equal, less, less equal, and, or, not, concatenation
Branching instructions	If, if-else, if-elseif, if-elseif-else, switch
Loop instructions	While, foreach
Functions	Function definition, function call
Input instructions	Read from keyboard (readline)

Output instructions	Print, echo
Comments	/*More lines*/, //one line
Other features	Array_pop, array_push, array_sum, sort, strlen

Note: We also took in consideration that both source and target languages (PHP and Python) are **untyped** programming languages. This allows a variable to change its content dynamically.

Lexical analysis

This phase helps the transpiler to turn the code into a set of tokens by finding matching patterns for the elementary elements of the grammar (lexemes). This phase allows to find lexical errors (i.e. a sequence of characters does not match the pattern of any token present in the grammar).

The input file for Flex, *php2python.l* contains three main sections.

First section - definitions

The first section (definition section) reports the imported libraries, defines the global variables and expresses names for starting conditions and for regular expressions that will be used in the next section. It also allows to specify options (e.g. the measurement of the line number though the %option yylineno or the stop after the reading of a single file using the %option noyywrap)
We used the following standard C libraries:

- **stdlib.h** to declare general utility functions, that can involve conversion between types (e.g., *atoi* function to convert a string to an integer) and memory allocation functions (e.g., *malloc* function for dynamic memory allocation), all of which would be used in the following sections.
- **stdio.h** to include declarations of functions and data types used for input/output operations (e.g., *fopen*, *fclose* and *remove* functions for file manipulation; *printf* and *fprintf* for manipulating input/output; *FILE* type as a structure containing pieces of information about a file needed to perform input/output operations on it).
- **stdbool.h** introduces functions for boolean data types.
- **string.h** contains functions and types used both in string and memory manipulation (e.g., *NULL* type that represents the null pointer constant that is a value that represents the address of an invalid location in memory; *strcat* function that concatenates two strings; *strcpy* function that copies one string to another one; *strlen* function that returns the length of the string; *strdup* function that allocates and duplicates a string in memory).

In particular, since our scanner is used in combination with a parser (built by Bison), we also included the *php2python.tab.h* file, that is the header file of the generated parser.

We took into consideration that a PHP script could also contain HTML code. However, a PHP significant code is always enclosed between **<?php** and **?>** lexemes, so we have been able to limit our analysis (lexical, syntax, semantic) and synthesis (code generation) steps to the rows included in a PHP code block. To do so, we defined new starting conditions:

- %x phpcode that would be activated when a PHP code block starts

- %x phpcodecomments that would be activated when a new more lines comment is found

Note: a starting condition is activated using the BEGIN keyword and this is a way of choosing which rules are currently used by the lexer and which ones are ignored

Second section - rules

The second section (rules) includes lexemes patterns and the associated action to be executed.
This section begins by setting the lexer behavior to the starting condition *phpcode* when the lexeme **<?php** is found. We additionally measure the starting line of a PHP block in order to provide additional information into the error message which could occur ("e.g. Syntax error: … at line 19 (line 3 of PHP block) "). When the lexeme **?>** is found, the action BEGIN(INITIAL), that represents the default ID of the start condition, is performed and it returns to the original state where only the rules with no start conditions are active.
So, we described the actions to do when a matching pattern is found for each lexeme (**in the PHP code**):

- in the case of *digits*, we accessed the content of the **yytext** variable (textual content of the lexeme). Then we assigned this value to the **yylval.intval** variable(we defined this data type as standard integer in the parser declarations) thanks to the **atoi()** function, and finally we return the found token NUMBER
- in the case of *strings*, the lexeme value (in yytext) is assigned to the **yylval.str** variable, though the **strdup()** function. Then the token STRING is returned
- in the case of *booleans*, we supported a process of "lowercasing" the possible boolean literals defined by users in PHP (i.e. we turn a possible and supported TrUE into true). To do so, we took in consideration the ASCII value of the characters and made a sum with the value 32. We passed the adjusted **yytext** into **yylval.str**. Then, the returned token is BOOL.
- in case of *variables*, the process is really similar to the strings and the returned token is VARIABLE
- in the case of new line or tab, no action is performed
- in the case of other supported lexemes, the corresponding token is returned (e.g. FNC when the pattern function is matched)
- in the case of *function names* or *sort options*, the **yylval.str** gets the content of **yytext** using the **strdup()** function. The returned token is NAME
- in case of a *comment* on a single line, we passed it by concatenating it to the # symbol (one line comment symbol in Python language). It will be reported to the translated output file
- in the case of a more lines comment, we launched a starting condition for PHP comments and concatenated all rows in a variable using the **strcat()** function. At the end we passed the resulting variable by concatenating it to the """ symbol (more lines comment symbol in Python language). It will be reported to the translated output file

The content not included in the PHP block (e.g. outstanding HTML code) is not considered.

Third section - user code

The third section (user code) has not been used.

Syntax analysis

This phase helps the transpiler to understand if sequences of tokens match the syntax rules of the programming language. It generates a structured representation of the program and allows to find syntax errors (i.e. undefined construct wrt grammar ones).

The input file for Bison, *php2python.y* contains three main sections.

First section - prologue

The first section (prologue) allows to:

- include standard libraries
- define constants and global variables/structures (C language)
- define user defined support functions
- define data types though the %union directive
- list all the tokens (%token directive) and define their type (%type directive)
- define associativity rules though %left and %right directives

We included standard libraries (stdio.h, regex.h, stdbool.h) but also other two header files defined to support the transliper during its process: **symbol table** (defined using the uthash library) and **gencode** (defines the translation process towards Python language) that includes **values memory** (which defines a stack data structure useful for supporting the translation process by storing data).

We kept track of each variable scope by identifying each particular scope/context by using the following encoding: **NAMEOFSCOPE - scope_number** (e.g. FUNCTIONSCOPE-0 for the first encountered function). By doing this, we are able to understand if called variables/functions are declared in their present scope or in upper level ones. In contrast, the module symtable can throw exceptions (undefined variables).

We defined the following data type: **id** for identifiers, **intval** for numbers and **str** for all other tokens.

Second section - rules

The second section (rules) consists in a formal definition of all possible constructs that could occur in a PHP program.

For instance, a PHP "program" is composed of "lines" between <?php (STARTPHP) and ?> (ENDPHP) tokens. A "Lines" element is recursively composed of "line" elements.
We decided to split non-terminal "lines" (and "line") elements in different context sensitive blocks, in order to support the context variable checking: linesinfunction, linesinifcondition, linesinelsecondition, linesinelseifcondition, linesinwhile, linesinforeach, linesinswitchcase, linesinswitchdefault.

When we find a line in a certain scope (e.g. FUNCTIONSCOPE-0), we try to add the scope in a stack data structure which contains currently open contexts.
Imagine to have a variable $a declared in the functional scope, still the current one, and then find a reference to $a: by looking at the current scope, we can be able to detect if the variable $a is declared or not in it. In contrast, we can check in upper open contexts, such as the global one.

The function used for adding new open scopes is called **add_new_open_context(name,number).**

Note: we do not support the traditional PHP global variable systems (global and $_GLOBALS directives) but we intend as “global” the highest and always open scope.

Note: example of contexts handling

```
                                                        Open contexts
$a = 3;                                                 [GLOBAL]

function B(){                                           [GLOBAL,FUNCT]

        if(true) {                                      [GLOBAL,FUNCT,IF]
                echo $a; //start by last currently open context
        }                                               [GLOBAL,FUNCT]
}                                                       [GLOBAL]
```

Each “line” non-terminal element can be:

- a simple expression (e.g print) terminated by semicolon
- a variable declaration terminated by semicolon
- a condition block (if, if else, if elseif else, …)
- a switch condition block
- a function definition or call
- a while statement
- a foreach statement
- a array_pop, array_push, array_sum and sort operations
- a comment (string containing the already translated comment)

For each of these elements, the transpiler will start by the terminal elements and then match the entire element block pattern: for example, in a variable declaration like $a = 3+$arr[2], the transpiler will first match the terminals, i.e number, number and variable, then translate/elaborate them (i.e translate the array access and check semantic, if is an array and if the index is on bound) and pass them to the higher element by the $$ variable: firstly expr, then the same will be done towards vardeclaration non terminal element.

When we find a variable declaration ($a = something) or a function definition (function a(){...}), the symbol table storage process is activated using the function **add_item(hash(a),a,VAR/FUN,SCOPE,scope_number,is_array,n_params)** which stores the reference and keeps also additional information (type of reference, scope, is an array and number of parameters if any). For more details look at the Symbol table section.

Note: we consider arrays references as variables and distinguish them by the *isarray* attribute in the symbol table entry.
Indeed, when we find a variable/function reference though the code, a process of context checking is activated. For each scope, all references to variables and functions are stored in two data structures (using **add_new_var_to_check()** and **add_new_func_to_check()** functions) and when the scope is closed (e.g end of the function) all references are checked using the **check_all_item_in_context(SCOPE,scope_number)**. The function checks in the current open scope

and in upper level open ones. This process is performed for each nested scope, when each one is closed. For more details look at the Symbol table section.

We also support few semantic checks:

- check if a variable is an array in its context or higher ones (if accessed by index, if passed to an array specific function like array_pop, etc.): This has been computed by the **check_if_var_is_array()** function, described in section Semantic analysis
- check if an array access does not cause an out of bound in its context or higher ones (index higher than the boundaries of the array): this has been performed by the function **check_n_params_match()**, described in section Semantic analysis
- check if the parameters of a function call respect the function definition in its context or higher ones: this has been performed by the function **check_n_params_match(different params config)**, described in section Semantic analysis

We also built a simple representation of the abstract syntax tree (AST) by reporting in a text file the references to the terminal and non terminal symbols in the order they are found, by using string representation (e.g. FOUND VARIABLE DECL IN GLOBAL SCOPE).

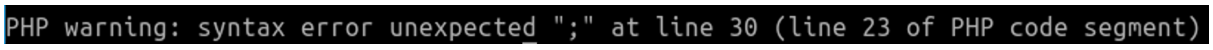
In order to support specific predictions translation, we built routines inside the gencode.h header that take the tokens and organize them in a Python valid construct. For instance, an array access, after the previously announced semantic checks, is translated using the **translate_array_access()** function. Additional details on the translation process will be reported in the Code generation section.

Third section - epilogue

In the third section, we defined the **main()** function, fundamental to launch the **yyparse()** function, which activates the parsing routine which ends with the end of the input file or with an error: in this case the yyparse function calls the routine **yyerror()**, which has been defined in this epilogue section. In the main function, we also removed any previous instances of the output files for generated Python code and generated abstract syntax tree representation.

The yyerror() function consists of a print of the detected errors (lexical, syntax or semantic). We were able to convert token symbols into human readable ones and to express the line number also with respect to the PHP code block.

An example of a possible syntax error is reported below:

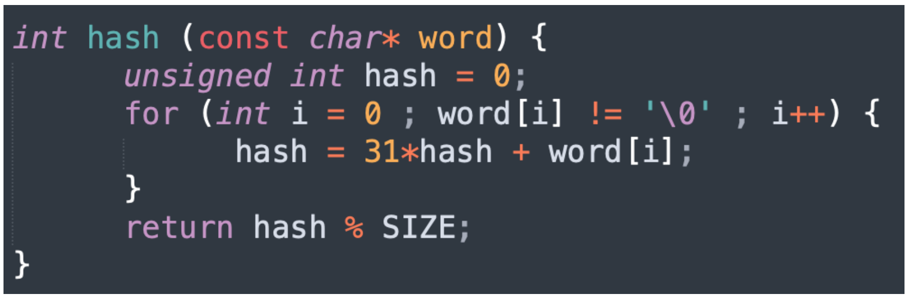


```
PHP warning: syntax error unexpected ";" at line 30 (line 23 of PHP code segment)
```

Symbol table

We decided to use the utash C library in order to handle an hash based symbol table for storing the references to variables and functions defined though the code. This helped us check possible reference errors and semantic errors).

We decided to rely on a standard hash function defined in the book called "The C Programming Language" aka "K&R" (Kernighan and Ritchie 1988). Its definition is reported below:



```
int hash (const char* word) {
      unsigned int hash = 0;
      for (int i = 0 ; word[i] != '\0' ; i++) {
            hash = 31*hash + word[i];
      }
      return hash % SIZE;
}
```

Note: this function works on each character of the passed string and builds a hashed string at each iteration, and then it uses a modulus sizing (SIZE=32768) in order to define a maximum value of hash combination equal to 32767.

We defined each symbol table entry structure (C struct) as follows:

- **id**: item key, corresponding to the hash generated from the variable/function name. In case of same name, in order to guarantee unicity, we decided to generate a new random key
- **name**: item name
- **type**: item type (variable VAR=1, function FUN=2)
- **scope**: item scope name (e.g. FUNCTIONSCOPE)
- **nscope**: item scope number associated to scope (e.g. 0 for first occurrence)
- **isarray:** flag for defining if the variable item (type=1) is also an array
- **nparams:** number of elements of the array (type=1 and isarray=1) or parameters of the function (type=2)

Note: the uthash library includes an additional field for making the struct hashable: UT_hash_handle.
Note: all symbol table entries will be unique using the **HASH_ADD_INT()** function of the utash library and referenced by their item id (hashed key).

We defined other support data structures for checking items in their contexts and for handling open contexts:

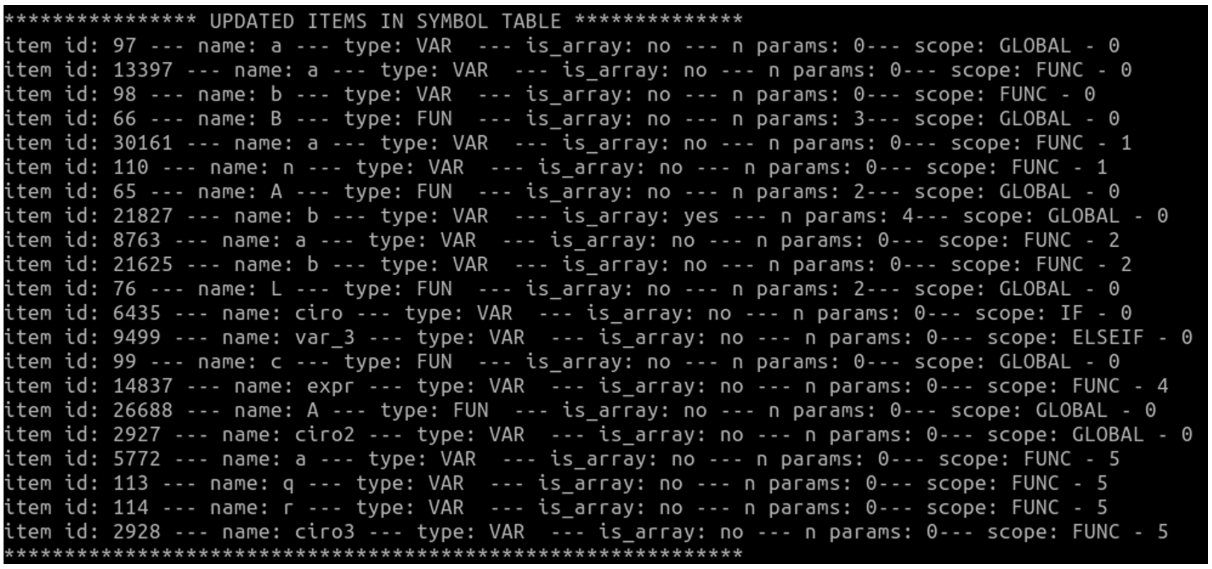
- **variables_to_check:** array containing all the variables to be checked in an open context or in upper level open ones. Filled with **add_new_var_to_check()** function.
- **functions_to_check:** array containing all the variables to be checked in an open context or in upper level open ones. Filled with **add_new_funct_to_check()** function.
- **open_contexts:** stack containing the references to all open contexts (we reserved two positions for each scope, since it is referenced by a name and a number). Filled with **add_new_open_context()** function.

Note: two important functions are used for context handling (**get_last_open_context()** for understanding the current last open scope and **close_context()** for closing a scope, e.g. end of a function)

Note: two other important functions are defined to take advantage of the standard utash procedures **HASH_FIND_INT()** and **HASH_DEL()**. **find_item()** is used to check if an item key is already registered in the symbol table and **delete_item()** and **delete_all()** are used to remove items from the symbol table.

Note: we implemented a **print_item()** function in order to display the content of the symbol table. We exploit the previously defined property of utash UT_hash_handle and its property .next.

An example is reported below:



```
**************** UPDATED ITEMS IN SYMBOL TABLE **************
item id: 97 --- name: a --- type: VAR  --- is_array: no --- n params: 0--- scope: GLOBAL - 0
item id: 13397 --- name: a --- type: VAR  --- is_array: no --- n params: 0--- scope: FUNC - 0
item id: 98 --- name: b --- type: VAR  --- is_array: no --- n params: 0--- scope: FUNC - 0
item id: 66 --- name: B --- type: FUN  --- is_array: no --- n params: 3--- scope: GLOBAL - 0
item id: 30161 --- name: a --- type: VAR  --- is_array: no --- n params: 0--- scope: FUNC - 1
item id: 110 --- name: n --- type: VAR  --- is_array: no --- n params: 0--- scope: FUNC - 1
item id: 65 --- name: A --- type: FUN  --- is_array: no --- n params: 2--- scope: GLOBAL - 0
item id: 21827 --- name: b --- type: VAR  --- is_array: yes --- n params: 4--- scope: GLOBAL - 0
item id: 8763 --- name: a --- type: VAR  --- is_array: no --- n params: 0--- scope: FUNC - 2
item id: 21625 --- name: b --- type: VAR  --- is_array: no --- n params: 0--- scope: FUNC - 2
item id: 76 --- name: L --- type: FUN  --- is_array: no --- n params: 2--- scope: GLOBAL - 0
item id: 6435 --- name: ciro --- type: VAR  --- is_array: no --- n params: 0--- scope: IF - 0
item id: 9499 --- name: var_3 --- type: VAR  --- is_array: no --- n params: 0--- scope: ELSEIF - 0
item id: 99 --- name: c --- type: FUN  --- is_array: no --- n params: 0--- scope: GLOBAL - 0
item id: 14837 --- name: expr --- type: VAR  --- is_array: no --- n params: 0--- scope: FUNC - 4
item id: 26688 --- name: A --- type: FUN  --- is_array: no --- n params: 0--- scope: GLOBAL - 0
item id: 2927 --- name: ciro2 --- type: VAR  --- is_array: no --- n params: 0--- scope: GLOBAL - 0
item id: 5772 --- name: a --- type: VAR  --- is_array: no --- n params: 0--- scope: FUNC - 5
item id: 113 --- name: q --- type: VAR  --- is_array: no --- n params: 0--- scope: FUNC - 5
item id: 114 --- name: r --- type: VAR  --- is_array: no --- n params: 0--- scope: FUNC - 5
item id: 2928 --- name: ciro3 --- type: VAR  --- is_array: no --- n params: 0--- scope: FUNC - 5
************************************************************
```

We added items in the symbol table (when a function or variable is defined) using the **add_item()** function, which params are the symbol table entry attributes defined before.

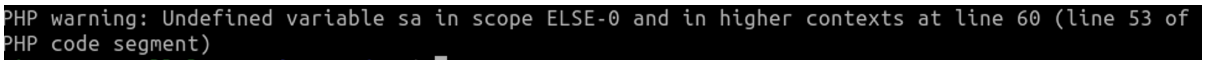
The add_item() function behavior is described as:

1. check if exists an entry in the symbol table with the item id passed as parameter (using the **HASH_FIND_INT()** function) and if not, add the new entry by binding the entry properties with the passed params.
2. In case of the same key (probably caused by homonymy) check all items id of items with the same name of the passed param with function **return_items_id_by_name()** and check if the item type matches in the same scope. If not (different types: variable and function), simply add the element with a new random generated key.
3. If the scope does not match, simply add the new element with a new generated key (since the name of the item is the same).
4. Show the updated symbol table with items described by their features

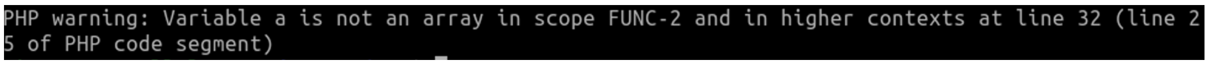
Note: the new entry adding is also supported by a **HASH_ADD_INT()** call that reserves that key.
When we found a variable/function reference, we checked if variable/function are previously defined in the current context or previously opened ones

When we find variables/functions references, we check if they have already been declared in the symbol table in the current scope or in upper open ones, using the **check_all_items_in_context(SCOPE,scope_number)** function. This scans the previously defined structures variables_to_check and function_to_check and perform on each element the function **check_item_in_context(id,name, type, SCOPE, scope_number)**: this function simply scans the symbol table and tries to find an entry which features (name, type, scope, scope_number) match the ones passed as params. If no occurrences are found in the currently open scope, we check in other open scopes of higher level (until we reach the global one) using the function **check_item_in_other_context()**. If no occurrences are found, a reference error message is thrown using the **yyerror()** function.

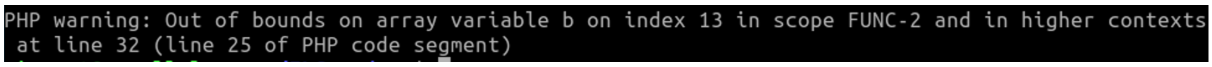
Examples of reference and semantic errors (described in Semantic analysis section) are reported below:



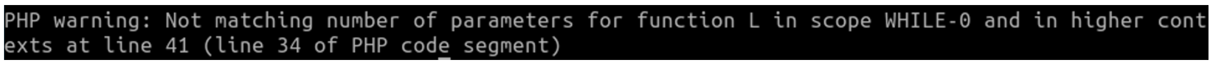
```
PHP warning: Undefined variable sa in scope ELSE-0 and in higher contexts at line 60 (line 53 of
PHP code segment)
```



```
PHP warning: Variable a is not an array in scope FUNC-2 and in higher contexts at line 32 (line 2
5 of PHP code segment)
```



```
PHP warning: Out of bounds on array variable b on index 13 in scope FUNC-2 and in higher contexts
 at line 32 (line 25 of PHP code segment)
```



```
PHP warning: Not matching number of parameters for function L in scope WHILE-0 and in higher cont
exts at line 41 (line 34 of PHP code segment)
```

Semantic analysis

This phase helps the transpiler to understand if logical errors are made though the code.

We decided to perform the following semantic checks:

1. check if a variable is an array when it is accessed by index
2. check if the index is on bound when array accessed by index
3. check if number of parameters in a function call match the function definition

The first semantic check is performed using the **check_if_var_is_array()** function, called when the transpiler matches an array access production. This function simply searches for an entry in the current scope which matches all the passed features name, scope, scope_number and also has the feature isarray=1 (the variable is an array). If no occurrences are found in the current scope, a check is performed in the other open contexts using the **check_if_var_is_array_in_other_context()** function.

The second and third checks are performed using the same function **check_n_params_match()**, called when the transpiler matches an array access production or a function call production.
The function produces these checks:

- in the case of array, it checks if exists an entry with the same name, type (1 means variable), scope, scope_number, that is an array (isarray=1) and checks if the specified index is valid (index>=0) and the number of params (that in case of array means the number of elements) is higher that the index
- in the case of functions, it checks if exists an entry with the same name, type (2 means function), scope, scope_number and checks if the specified index matches the number of params

If no occurrences are found, we check in other open contexts using the function **check_n_params_match_in_other_context()**.

Error handling

When an error occurs, our transpiler throws an exception message using the function **yyerror(error_string)**. We are able to pass a specific message that will be shown to the user. In order to make the tokens human-readable, we use the function **convert_symbols(error_message)**. We also kept track of the line number where the exception occurs (in the whole document and also with respect to the PHP code block). Some examples of error messages have been reported before.

Code generation

This phase helps the transpiler to turn the code from the source language (PHP) into the target one (Python) after a successful error checking phase. To perform this task, we build several specific functions able to organize the tokens in a correct format following Python language rules.

The bottom up parser starts by translating the terminals and then the output of the translation is passed forward using the $$ variable and all the productions are covered by the translation process. This helps moving parts of code already translated to the next organization phases.

Example: if we encounter the statement $x = 30+$a[2]; the parser will firstly detect the terminals (two numbers and a variable), then translate the number (no translation needed) and the array access (made of variable and number, translated by using the **translate_array_access()** function). Then the parser will recognize the sum operation and will translate it. After that, the parser will match the pattern of a variable declaration ($x=expr) and will translate it by organizing all the previous translated stuff.

expr: VARIABLE SQ1 NUMBER SQ2 {..... $$ = translate_array_access($1+1,itoa($3,str));}

Note: variables $x (with x a number from 1 to N) identify and contain the value of each token in the corresponding position in the production.

Before the translation phase the transpiler counts the instruction inside each context (e.g. with the function **increase_number_instr_function()**): this helps in the translation phase, since we make use of data structure for passing forward different information (such as parameters of a function definition, statements inside a function definition, etc.) that will be organized in the final construct for the specific block in Python. We also initialize the FILE write process in order to have the possibility of writing information on the output file output_translated.py using the **fprintf()** standard function.

Example 2: if we encounter this peace of code:

```
function my_func($a,$b) {
        echo $a;
      $x = $a+$b;
        }
```

The parser will detect the terminals $a and $b (VARIABLE), as function variable parameters (varparams), and will translate them and push to a stack memory (using the the **push()** function defined in the valuesmemory.h header) which will be at the end “popped out” in the organization process for the considered function. The parser will detect the variable $a inside the echo construct (VARIABLE) and then will match the pattern of the echo and will translate it. The result of the translation will be pushed to the stack memory. Then, the parser will detect the terminals $a and $b (VARIABLE), will translate them, then will match the + operation and translate it, then will match the variable declaration and will translate it and finally will push the resulting string to the stack memory. At the end, when the parser will match the pattern of the “function” production, the translating process for the considered function will be terminated using the procedure **organize_function_def()**: this function will first extract the statements lastly stored in the stack (looking at the measured number of instructions in the current function block), then will extract the parameters. The extractions are made using the **pop()** function defined in the valuesmemory.h header.
The function will lastly organize the final string that will be written to the output file:

```
def my_func(a, b):
        print(a)
        x = a + b
```

All other terminals and constructs are translated using simpler functions or using functions similar to the one presented in the Example 2, which aims at extracting the already translated statements and parameters and organizing them in the final construct.

Limits of our transpiler

In this section we honestly express the limits of our PHP to Python transpiler:

- Our transpiler is not able to properly translate a program containing a nidification of the same construct (e.g. a function inside a function, or an if condition inside an if condition)
- Our transplier does not support multiple elseif in a condition construct
- Our transpiler does not support the standard global variables system offered by PHP language, but looks at a standard scenario (moreover, for global variable we mean a variable in the first open context, called global)
- We were not able to solve shift reduce conflicts after building the parser using Bison

Support code to our transpiler

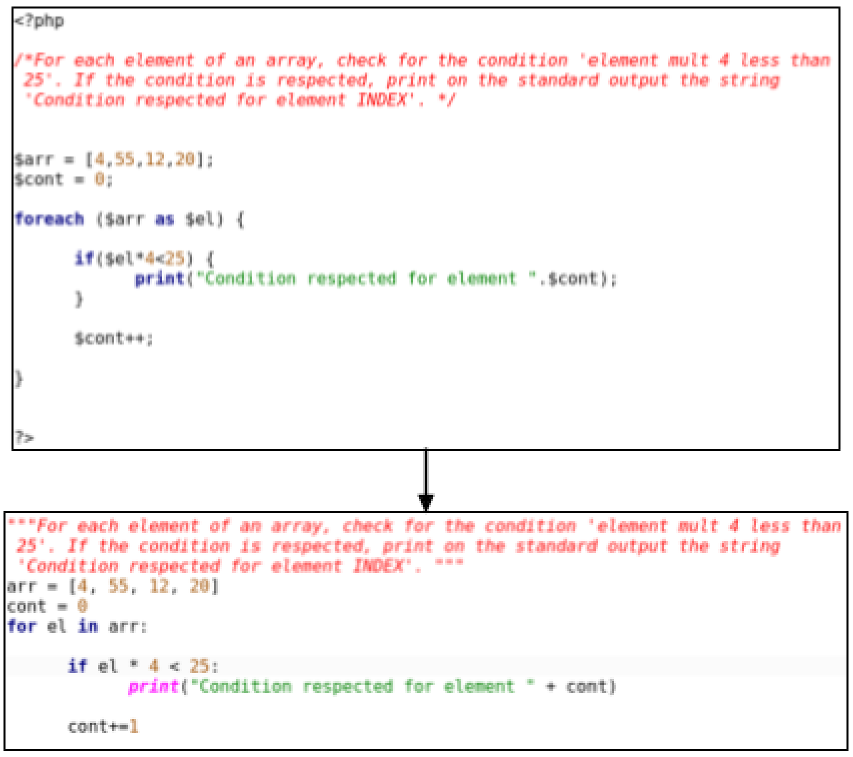
In this section we briefly recall the support code we defined through the transpiler:

- in the php2python.l script we store the number of measured lines outside the PHP block in order to enrich a possible error message with the location of the error also with respect to the PHP block
- to handle the switch cases translation, we make use of a array where we store the translated content
- to handle the operations between numbers we make use of a stack where we store the values
- the main support code is defined in valuesmemory.h, where we define a stack for keeping all translated statements and parameters in order to organize them properly in specific functions (one per type of construct). We can access the stack using the **push()** and **pop()** function as defined before.
- we write on output files the translated code and the AST.

Use case and tests

In this section we report a few test cases conducted after finishing the transpiler development. Obviously our code works in all other possible conditions specified by the restrictions.

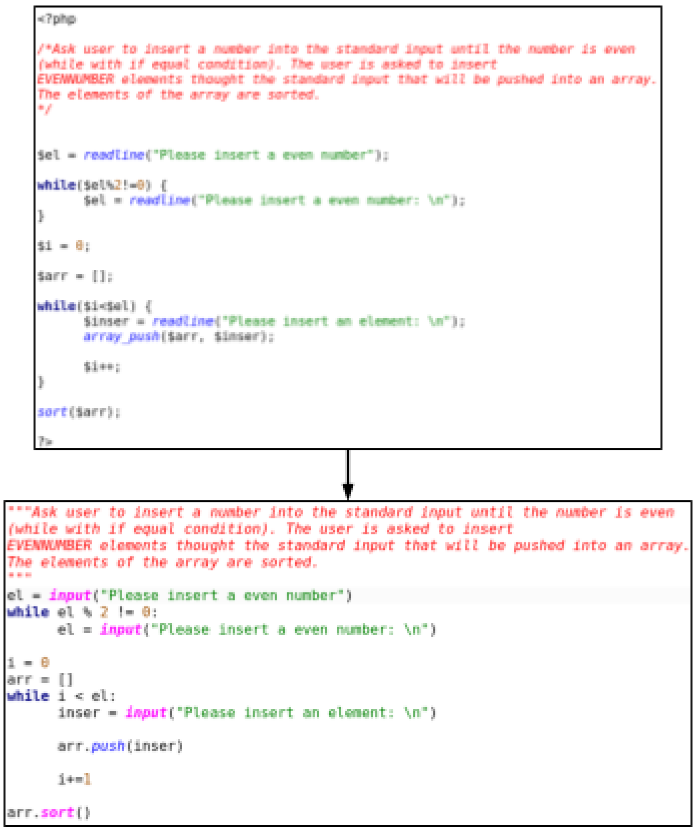
Use case 1



```
<?php

/*For each element of an array, check for the condition 'element mult 4 less than
 25'. If the condition is respected, print on the standard output the string
 'Condition respected for element INDEX'. */


$arr = [4,55,12,20];
$cont = 0;

foreach ($arr as $el) {

      if($el*4<25) {
            print("Condition respected for element ".$cont);
      }

      $cont++;

}


?>
```

```
"""For each element of an array, check for the condition 'element mult 4 less than
 25'. If the condition is respected, print on the standard output the string
 'Condition respected for element INDEX'. """
arr = [4, 55, 12, 20]
cont = 0
for el in arr:

      if el * 4 < 25:
            print("Condition respected for element " + cont)

      cont+=1
```

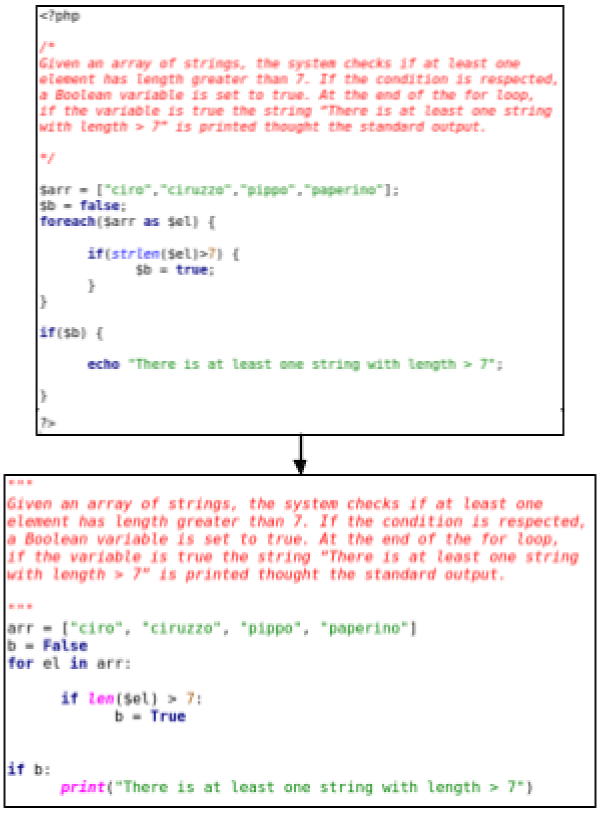
Use case 2



```php
<?php

/*Ask user to insert a number into the standard input until the number is even
(while with if equal condition). The user is asked to insert
EVENNUMBER elements thought the standard input that will be pushed into an array.
The elements of the array are sorted.
*/


$el = readline("Please insert a even number");

while($el%2!=0) {
	$el = readline("Please insert a even number: \n");
}

$i = 0;

$arr = [];

while($i<$el) {
	$inser = readline("Please insert an element: \n");
	array_push($arr, $inser);

	$i++;
}

sort($arr);

?>
```

```python
"""Ask user to insert a number into the standard input until the number is even
(while with if equal condition). The user is asked to insert
EVENNUMBER elements thought the standard input that will be pushed into an array.
The elements of the array are sorted.
"""
el = input("Please insert a even number")
while el % 2 != 0:
    el = input("Please insert a even number: \n")

i = 0
arr = []
while i < el:
    inser = input("Please insert an element: \n")

    arr.push(inser)

    i+=1

arr.sort()
```

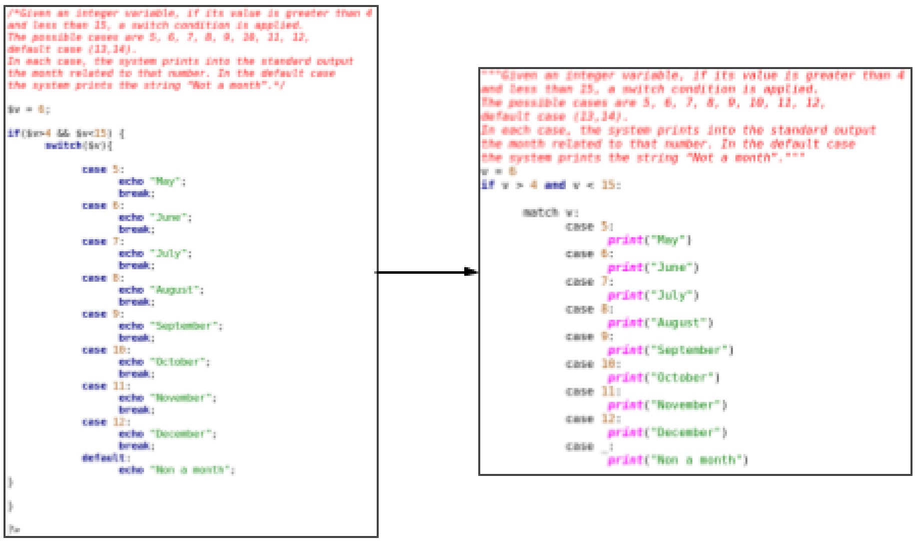
Use case 3



```php
<?php

/*
Given an array of strings, the system checks if at least one
element has length greater than 7. If the condition is respected,
a Boolean variable is set to true. At the end of the for loop,
if the variable is true the string "There is at least one string
with length > 7" is printed thought the standard output.

*/

$arr = ["ciro","ciruzzo","pippo","paperino"];
$b = false;
foreach($arr as $el) {

	if(strlen($el)>7) {
		$b = true;
	}
}

if($b) {

	echo "There is at least one string with length > 7";

}
?>
```

```python
"""
Given an array of strings, the system checks if at least one
element has length greater than 7. If the condition is respected,
a Boolean variable is set to true. At the end of the for loop,
if the variable is true the string "There is at least one string
with length > 7" is printed thought the standard output.

"""
arr = ["ciro", "ciruzzo", "pippo", "paperino"]
b = False
for el in arr:

    if len($el) > 7:
        b = True


if b:
    print("There is at least one string with length > 7")
```

Use case 4



```php
/*Given an integer variable, if its value is greater than 4
and less than 15, a switch condition is applied.
The possible cases are 5, 6, 7, 8, 9, 10, 11, 12,
default case (13,14).
In each case, the system prints into the standard output
the month related to that number. In the default case
the system prints the string "Not a month".*/

$v = 6;

if($v>4 && $v<15) {
        switch($v){

                case 5:
                        echo "May";
                        break;
                case 6:
                        echo "June";
                        break;
                case 7:
                        echo "July";
                        break;
                case 8:
                        echo "August";
                        break;
                case 9:
                        echo "September";
                        break;
                case 10:
                        echo "October";
                        break;
                case 11:
                        echo "November";
                        break;
                case 12:
                        echo "December";
                        break;
                default:
                        echo "Non a month";
}

}

?>
```

```python
"""Given an integer variable, if its value is greater than 4
and less than 15, a switch condition is applied.
The possible cases are 5, 6, 7, 8, 9, 10, 11, 12,
default case (13,14).
In each case, the system prints into the standard output
the month related to that number. In the default case
the system prints the string "Not a month"."""
v = 6
if v > 4 and v < 15:

        match v:
                case 5:
                        print("May")
                case 6:
                        print("June")
                case 7:
                        print("July")
                case 8:
                        print("August")
                case 9:
                        print("September")
                case 10:
                        print("October")
                case 11:
                        print("November")
                case 12:
                        print("December")
                case _:
                        print("Non a month")
```

Conclusion

In conclusion, we are satisfied by the outcome derived by the design and development of our PHP to Python language transpiler. We are aware of our reported limits. We think that the most challenging task was the initial approach to the compiler's world but also the parsing step.
Future versions of our project could include fixes over the reported limitations and introduce new constructs to support even more the possible structure, data types and conditions provided by the PHP language.

References

1. Python documentation. Accessed January 24, 2023. https://docs.python.org/3/.
2. Aaby, Anthony A. "Compiler Construction using Flex and Bison." ADMB. Accessed January 24, 2023. https://www.admb-project.org/tools/flex/compiler.pdf.
3. "Bison Tutorial." UCR CS. Accessed January 24, 2023. http://alumni.cs.ucr.edu/~lgao/teaching/bison.html#link.
4. Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C programming language*. 2nd ed. N.p.: Prentice-Hall.
5. Paxson, Vern, and Jef Poskanzer. "Flex - a scanner generator." Princeton University Computer Science. Accessed January 24, 2023. https://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html.
6. "PHP Manual - Manual." PHP. Accessed January 24, 2023. https://www.php.net/manual/en/.
7. Stallman, Richard, and Charles Donnelly. 1999. *The Bison Manual: Using the YACC - Compatible Parser Generator for Version 1.29*. N.p.: Gnu Press.
8. "uthash User Guide." 2022. Troy D. Hanson. https://troydhanson.github.io/uthash/userguide.html.