C-to-Python Transpiler - Report

Implementation of a source-to-source compiler from C to Python
for the course Formal Languages and Compiler at Polytechnic
University of Bari

Domenico de Gioia, Ivan Maddalena

November 15, 2023

Contents

Chapter 1

Overview

This report presents the Formal Languages and Compilers Project which involved the development of a C to Python transpiler. The main goal of this project was to create a software that can translate source code written in the C programming language into equivalent Python code. This transpiler was designed to ease the migration of existing C applications to the Python development environment by providing an efficient way to convert code without having to completely rewrite it.

The goal of this work is to make a one-way translation between two high-level programming languages to simulate the behavior of a compiler. The compiler was built in C language with the help of the automatic tools Flex scanner generator and the Bison parser generator.

1.1 Elements of source language

Our transpiler works only on a subset of the C language. The restriction of source language, agreed for our project, is composed by the following:

- Single-line and multi-line comment;
- Data types: `int` (32-bit signed integer values), `float` (32-bit floating point values), `char*`;
- Variable types: variables and arrays;
- Single and multiple declarations of variables;
- Single declarations of arrays;
- Arithmetic operators: addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`);
- Comparison operators: greater than (`>`), greater than or equal (`>=`), less than (`<`), less than or equal (`<=`), equal (`==`), not equal (`!=`);

- Logical operators: and (`&&`), or (`||`), not (`!`);
- Reference operator (`&`);
- Assignment operator (`=`);
- Expressions: the use of logical, arithmetic, and comparison operators, and they can include also variables, array elements, and function calls;
- Assignment of an expression value to a variable or an array element;
- Branching: `if-else` statement;
- Loop/iteration: `for` statement;
- Function: declarations and calls;
- `return` statement;
- Input: input from the user with built-in function `scanf`;
- Output: standard output in console with built-in function `printf`.

1.2 Project structure

The project consists of several files, listed below

1. `global.h`: This file contains global variables and function prototypes used in multiple files (for example, defining color constants in error messages and function prototypes for creating and presenting those messages).
2. `scanner.l`: This file contains the specifications for the scanner build using the Flex tool. In addition, it contains function definitions that allow you to manage the creation and presentation of error messages (for example, a function that prints errors on standard error and maintains an error counter or a function that prints warnings on standard error).
3. `parser.y`: This file contains the specifications needed to generate the parser using the Bison tool. These specifications include the grammar rules that define the syntactic structure of the source language, but also other auxiliary instructions and functions to support the creation and closing of scopes in the source code.
4. `ast.h` and `ast.c`: These files contain the definition of the structure of the Abstract Syntax Tree (AST) nodes and the functions needed to create and modify their attributes. The `ast.h` file defines the data structures to represent the different types of nodes of the AST, while the `ast.c` file implements the functions that allow the creation of AST nodes and the modification of their attributes.

5. `symbol_table.h` and `symbol_table.c`: These files contain the definition of the Symbol Table structure and the functions that allow the management of the tables and their symbols. Furthermore, they include structures and functions to manage the hierarchy of tables according to the scopes. The `symbol_table.h` file provides the declarations of the data structures and functions associated with the symbol table, while the `symbol_table.c` file contains the implementation of the functions defined in the `symbol_table.h` file.

6. `semantic.h` and `semantic.c`: These files contain the functions and definitions needed to implement semantic checks in the compiler. The `semantic.h` file is the header file that contains the declarations of the functions and data structures used to perform the semantic checks, while the `semantic.c` file contains the implementation of the functions defined in the `semantic.h` file ".

7. `translate.h` and `translate.c`: These files contain the functions used when translating the source code into another target language. The `translate.h` file is the header file that contains the declarations of the functions and data structures used for translation, while the `translate.c` file contains the implementation of the functions defined in the `translate.h` file (functions in the `translate.c` file may include, for example, functions for translating variable declarations and expression handling functions).

8. `uthash.h`: This is a library that is widely used to implement hash tables in the C language. It is often used as a helper for handling Symbol Tables in compilers or other projects.

Chapter 2

Implementation

This chapter will describe the steps of a compiler, paying particular attention to the data structures used.

The implementation phase of a transpiler from C to Python is a complex process that requires careful planning and a solid knowledge of both programming languages.

The transpiler analyzes the source code in C to identify control structures, variable declarations, functions and other key elements. Next, use this information to generate a Python equivalent and as similar as possible that performs the same operations.

A crucial aspect of implementing a transpiler from C to Python is managing the differences between the two languages.

- First of all, C is a strongly typed language, while Python is dynamically typed: this means that the transpiler must be able to infer the type of variables based on the context in Python.
- In C, the type of a variable must be declared when it is created, and only values of that type must be assigned to it. In Python, variables are untyped. There is no need to declare the type of variable. A given variable can be stuck on values of different types at different times during the program execution.
- Pointers are available in C, but not in Python.

However, implementing a compiler is a complex task that involves several stages as explained in the following paragraphs.

2.1 Lexical Analysis

The first step is to break down the input source code into a stream of tokens. It involves identifying keywords, identifiers, literals, operators, and other tokens

that make up the language. Tokens can be represented as data structures that contain information such as token type, value, and location in the source code.

The **Flex** tool (Fast LEXical analyzer generator) was used to generate the scanner based on lexical specifications.

The `scanner.l` file refers to the lexical specification file used to define the lexical rules for a scanner or lexical analyzer. It contains a set of rules written in a specific syntax understood by Flex. These rules define patterns and corresponding actions for recognizing and tokenizing the input text. Each rule consists of a regular expression pattern and an associated action. Regular expressions define the patterns to match the input text, and actions specify the code to execute when a pattern is matched. This file is generally composed of the following sections: declarations, rules and user code.

2.1.1 Declarations

This section can include directives for including header files or defining macros and variable or function declarations.

- The `%option yylineno` directive is used to enable the tracking of line numbers during lexical analysis, keeping this value in the variable `yylineno`. This variable is used to make the error diagnosis more accurate by indicating the line number in particular in the error and warning messages.
- Flex provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with `<sc>` will only be active when the scanner is in the start condition named `sc`. In addition to the `INITIAL` state, defined default by Flex, two exclusive states are defined:
  - `MULTI_LINE_COMMENT`: used for the recognition of multiline comments, inserted within the source program, which will be ignored;
  - `QUOTE`: used for the recognition of strings, whose content is passed to the parser as the `STRING` token attribute.
- The inclusion directive `#include "token.h"` of the extra output file `token.h` generated by Bison.

2.1.2 Rules

This section can include definitions of tokens, with the corresponding actions to be performed. The most complex rules are given below.

- In the event of an error, in order to show the user the content of the line in which this error occurred, it was necessary to use the `copy_line()` function. This function is invoked according to the following rule:

```
^.*                                    { copy_line(); }
```

- The use of the `MULTI_LINE_COMMENT` status allows the recognition of comment and error in case of failure to close a comment. In general, both single-line comments and multi-line comments in the source program are ignored and are not translated into the final program.

```
    /* SINGLE-LINE COMMENTS */
"//".*                          { }

    /* MULTI-LINE COMMENTS */
"/*"                            { BEGIN MULTI_LINE_COMMENT; }
<MULTI_LINE_COMMENT>[^*\n]      { }
<MULTI_LINE_COMMENT>"/*"        { BEGIN INITIAL; }
<MULTI_LINE_COMMENT><<EOF>>     { yyerror("Unterminated
                                comment."); BEGIN INITIAL; }
```

- The use of the `DQUOTE` status allows the recognition of error in case of failure to close a string.

```
    /* STRINGS */
\"\"                            { yylval.s = " ";
                                return STRING_VALUE;}
\"                              { BEGIN QUOTE; }
<QUOTE>([^\\\n]|\\.)*           { yylval.s = strdup(yytext);
                                return STRING_VALUE;}
<QUOTE>\"                       { BEGIN INITIAL; }
<QUOTE><<EOF>>                  { yyerror("missing terminating
                                \" character"); BEGIN INITIAL; }
```

- The rule to ignore whitespaces has been included.
- The following rule has been inserted to report an unrecognized character error in the previous rules.

```
.           { yyerror(error_string_format("Character" BOLD " %s not
            recognized" RESET, yytext)); }
```

The variable `yytext` is a pointer to the matched string. It holds the text matched by the current token. To keep the string value of the token pointed to, the statement `yylval.s = strdup(yytext);` is used to memorize the semantic value associated to `ID`, `INT_VALUE`, `FLOAT_VALUE` and `STRING_VALUE` in the corresponding member of the Bison global variable `yylval` declared as union in the parser.

2.1.3 User code

This section is commonly used for defining companion routines that interact with or are invoked by the scanner. However, including this section is optional. In this project, the following functions has been defined.

- `copy_line`: this function initially deallocates the dynamically allocated memory reducing memory wastage (`free` function), it dynamically allocates a block of memory with the corresponding size (`malloc` function), then it copies the entire input string `yytext` (`strcpy` function) and finally it returns the line to the input buffer to match it with the successive rules (`yyless(0)` function);
- `error_string_format`: it is a variadic function to support the functions for printing errors, warnings and notes with a string already formatted;
- `yyerror`: it prints errors on the standard error, updating the error counter;
- `yywarning`: it prints warnings on the standard error;
- `yynote`: it prints notes on the standard error.

2.2 Syntactic analysis

The syntactic analysis phase has the purpose of determining whether a sequence of tokens, received as input, constitutes a valid sentence according to the syntax of the language. In the context of our project, we have chosen to use the Bison parser generator, which is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR (Look-Ahead, Left-to-right) parser tables.

2.2.1 Description of parser.y file

The "parser.y" file represents the input to Bison and contains the productions that define the syntax of the source language. These productions are expressed using the Backus-Naur form.

The "parser.y" file contains the definitions of the grammatical productions and the corresponding semantic actions for the syntactic analysis of the source language, and represents the starting point for generating the parser using Bison. The Bison parser generator relies on the information provided by this file to

create a parser that is able to parse the syntactic structure of the source language and handle any syntax errors.

In the following paragraphs of this report the file will be explained in more detail.

2.2.2 Prologue of the `parser.y` file

The "Prologue" of the `parser.y` file represents an initial section of code that contains several key declarations and definitions necessary for the transpiler to function. This section is fundamental to establish the working environment and prepare all the resources necessary for the syntactic analysis of the C source code and the subsequent translation into Python.

The Prologue includes headers from standard C libraries, such as `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<ctype.h>`, which provide essential functions and data structures for manipulation of strings, memory management and other basic operations. These libraries are required for implementing parsing and translation functionality. Furthermore, this section also includes some header files, such as `symbol_table.h`, `ast.h`, `global.h`, `semantic.h`, and `translate.h`. These files contain definitions of data structures and functions that play a crucial role in syntactic analysis, scope management, AST (Abstract Syntax Tree) generation, semantic verification, and translation of C code into Python.

```
// Prologue

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "symbol_table.h"
#include "ast.h"
#include "global.h"
#include "semantic.h"
#include "translate.h"
```

Table 2.1: Prologue of the `parser.y` file.

Some variables and data structures are also declared in the Prologue, such as `filename`, `error_num`, `main_flag`, `current_scope_lvl`, `current_symbol_table`, and others. These variables perform important tasks, such as tracking the name of the input file, counting errors during parsing, detecting the main function in C code, managing scopes, and storing the pointer to the current symbol table.

In this section, we find variable declarations and initializations essential for syntactic analysis and the generation of the Abstract Syntax Tree (AST). These variables include:

- `struct AstNode *root`: This variable represents the root node of the AST, which provides a structured representation of the C source code syntax.
- `struct AstNode *param_list`: A pointer to a list of nodes that contains information about function parameters. This list is used to store parameters during parsing.
- `enum TYPE ret_type`: This enumerator keeps track of the data type returned by a declared function. This is crucial for ensuring consistency in the data types returned by functions.

Within the Prologue, functions such as `scope_enter()`, `scope_exit()`, and `fill_symbol_table()` are declared. These functions play key roles in managing scopes, adding symbols to the symbol table, and exiting analysis scopes.

2.2.3 Grammar rules

This section defines the grammar rules of the source language using Backus-Naur Form (BNF) notation. Each rule specifies how tokens must be combined to form valid expressions according to the syntax of the language. Rules can include terminals (tokens) and non-terminals (symbols representing other productions).

Productions for the scopes

A fundamental part of our project concerns scope management, i.e. the visibility and accessibility of variables and functions declared in the C source code. To address this challenge, we defined productions for scopes within our `parser.y` file. This aspect of the parser plays a crucial role in ensuring that variables and functions are correctly declared and used in the resulting Python code.

In our parser, we have created several productions to handle variable declaration, variable initialization, and function declaration within the C source code. In particular, we have the following productions:

```
program
  :  {scope_enter();} global_declaration_list    {root = $2;
scope_exit();}
  ;

global_declaration_list
  :  global_declaration
  |  global_declaration global_declaration_list    {$$ =
link_AstNode($1, $2);}
  ;

global_declaration
  :  var_initialization
  |  var_declaration
  |  func_declaration    {check_main($1);}
  ;
```

Table 2.2: Productions for `program` and global declarations in the `parser.y` file

The `program` production represents the entry point of our parser and is responsible for initializing and exiting the global scope. It saves the root of the Abstract Syntax Tree in the respective root global variable, used to traverse the tree in the code generation phase. The opening and closing of the global scope occurs through the use of the `scope_enter()` and `scope_exit()` functions:

- The `scope_enter()` function creates a new Symbol Table associating it with a unique integer that identifies the current scope. In addition, the function establishes a link between the new symbol table and the previous symbol table, allowing you to maintain a tree structure of symbol tables, where each symbol table represents a specific scope.
- On the other hand, the `scope_exit()` function takes care of deleting the current symbol table when exiting the current scope. This could lead to the freeing of resources allocated for the symbol table or the removal of the associations between the variables declared in the current scope.

The two functions also take care of respectively increasing and decreasing the identification number of the current scope, initialized to zero for the global scope : each time `scope_enter()` is called, the identification number of the current scope is incremented, and when `scope_exit()` is called, the number is decremented.

These functions therefore manage the scope of the entire program, and are called in correspondence with the productions for managing the scopes. Every time the `scope_enter` production, formed by the '{' terminal, is recognized, a new scope is opened, which is closed when the `scope_exit` production, in turn formed by the '}' terminal, is recognized:

```
compound_statement
  :  scope_enter statement_list scope_exit    {$$ = $2;}
  ;

scope_enter
  :  '{'    {scope_enter();}
  ;

scope_exit
  :  '{'    {scope_exit();}
  ;
```

Table 2.3: Productions for the scopes in the `parser.y` file.

Productions for statements

The non-terminal `statement_list` contains a list of statements, which also includes `compound_statement` itself, which is therefore a recursive production. This allows for multiple nested scopes, which improves clarity, variable management, readability, and structure of the code.

```
statement_list
  :  statement
  |  statement statement_list    {$$ = link_AstNode($1, $2);}
  ;

statement
  :  var_declaration
  |  compound_statement
  |  expr_statement
  |  selection_statement
  |  iteration_statement
  |  return_statement
  |  printf_statement
  |  scanf_statement
  |  assignment_statement
  ;
```

Table 2.4: `statement_list` in the `parser.y` file.

Productions for FOR loops

A particular case concerns the `for` statement: when there is a declaration inside the `init`, for this type of statement a new scope is opened immediately after the opening round bracket.

```
iteration_statement
  :  FOR '(' init ';' cond ';' update ')' embedded_statement
{$$ = new_for_node(FOR_T, $3, $5, $7, $9); scope_exit();}
  ;
```

Table 2.5: `iteration_statement` in the `parser.y` file

Productions for variable declarations

The productions for `var_declaration` in this code excerpt handle variable declaration based on the specified types and initializers. These productions are critical for parsing and translating variable declarations within C source code into Python.

First, productions define different cases depending on the type and initialization of the variables:

- In the first case, a variable of specified type is declared with or without initialization. This behavior is based on the specified type and the resulting AST is constructed accordingly. The `fill_symbol_table()` function is then called to insert the corresponding symbols into the current symbol table.
- In the second case, a pointer to a specified type is declared. As in the previous case, the `fill_symbol_table()` function manages the insertion of the symbol and the associated type.
- In the third and fourth cases, the variables are declared with initializers, but the behavior is similar to the first two cases in terms of type and variable handling. Specific error checking logic is provided to detect unexpected `char` variables.
- Finally, if an identifier (ID) is encountered that does not match a known type, an error is raised and an unknown type is reported.

These productions play an essential role in analyzing C source code, identifying variable declarations and their typing, allowing the transpiler to correctly generate equivalent Python code.

Productions for Printf and Scanf functions

These productions describe how these input/output functions, `printf` and `scanf`, are used within C source code and how they are handled during parsing.

Regarding the `printf` statement, the productions capture several situations:

- The first production addresses the simplest case, where `printf` is used with a format string in parentheses. This format string is represented by `$3`, and production creates a new node of the AST for the `printf` call. A check on the correctness of the format is also carried out with `check_format_string`.
- The second production handles the case where there are arguments to print along with the format string. The parameters to print are represented by

`$5`, and production creates a new AST node for the `printf` call with all arguments. Also in this case, the format control is carried out.

- The third production reports an error when the `printf` statement has no arguments, indicating that there are `"too few arguments to the printf function"`. They will be explored in more detail in the "Productions for Errors" section.

As for the `scanf` statement, the productions follow a similar pattern:

- The first production handles the use of `scanf` with a format string and a list of variables to fill. The format string is represented by `$3`, and the variable list is represented by `$5`. Production creates a node of the AST for the `scanf` call and checks the correctness of the format with `check_format_string`.
- The second production deals with the case where `scanf` is used only with the format string and has no variables to fill. The format string is represented by `$3`.
- The third production reports an error when the `scanf` statement has no variables to fill, indicating that there are `"too few arguments to the scanf function."`. Also in this case, the issue will be addressed in more detail in the following paragraphs.

```
printf_statement
  :  PRINTF '(' format_string ')' ';'    { $$ =
new_func_call_node(FCALL_T, $1, $3); check_format_string($3, NULL,
PRINTF_T); }
  |  PRINTF '(' format_string ',' args ')' ';'    { $$
= new_func_call_node(FCALL_T, $1, link_AstNode($3,$5));
check_format_string($3, $5, PRINTF_T); }
  |  ...
  ;

scanf_statement
  :  SCANF '(' format_string ',' scanf_var_list ')'
';'    { check_format_string($3, $5, SCANF_T); $$ =
new_func_call_node(FCALL_T, $1, link_AstNode($3, $5)); }
  |  SCANF '(' format_string ')' ';'    { check_format_string($3,
NULL, SCANF_T); $$ = new_func_call_node(FCALL_T, $1, $3); }
  |  ...
  ;
```

Table 2.6: Productions for `printf` and `scanf` in the `parser.y` file

The `scanf_var_list` production is responsible for managing the variables within the `scanf` argument list. In particular, it determines whether a variable should be passed by reference or by value.

Here is a more detailed explanation of the four possible rules:

- `& var`: This rule indicates that a variable must be passed by reference. The `check_var_reference($2)` function is called to ensure that the variable is valid, and then `by_reference($2)` is used to indicate that the variable should be passed by reference. The variable is then assigned to $$.

- `var`: This rule indicates that a variable must be passed by value. The `check_var_reference($1)` function is called to check the validity of the variable, but it is not explicitly stated that the variable should be passed by reference. The variable is then assigned to `$$`.

- `& var ',' scanf_var_list`: This rule indicates that a variable should be passed by reference, and the process repeats for the rest of the list (`$4`) to handle subsequent variables. The `check_var_reference($2)` function is called to check the validity of the variable, and `by_reference($2)` is used to indicate that the variable should be passed by reference. Then, `link_AstNode($2, $4)` creates a linked list of variables that are passed by reference.

- `var ',' scanf_var_list`: This rule indicates that a variable should be passed by value, and the process repeats for the rest of the list (`$3`) to handle subsequent variables. The `check_var_reference($1)` function is called to check the validity of the variable, but it is not explicitly stated that the variable should be passed by reference. Then, `link_AstNode($1, $3)` creates a linked list of variables that are passed by value.

```
scanf_var_list
  :  '&' var    { check_var_reference($2); by_reference($2); $$ =
$2; }
  |  var    { check_var_reference($1); $$ = $1; }
  |  '&' var ',' scanf_var_list    { check_var_reference($2);
by_reference($2); $$ = link_AstNode($2, $4); }
  |  var ',' scanf_var_list    { check_var_reference($1);
link_AstNode($1, $3); }
  ;
```

Table 2.7: `scanf_var_list` from `parser.y` file

Productions for errors allow to generate meaningful, detailed error messages when the C source code contains errors or is invalid. This can help you find and fix errors in C code more efficiently, improving the overall user experience. In this case some productions have been inserted, extending the grammar, to highlight some of the more common errors and generate more specific error messages.

```
printf_statement
  :  ...
  |  ...
  |  PRINTF '(' ')' ';'    { $$ = new_error_node(ERROR_NODE_T);
yyerror("too few arguments to function" BOLD " printf" RESET); }
  ;

scanf_statement
  :  ...
  |  ...
  |  SCANF '(' ')' ';'    { $$ = new_error_node(ERROR_NODE_T);
yyerror("too few arguments to function" BOLD " scanf" RESET); }
  ;
```

Table 2.8: Example of Productions for errors in the `parser.y` file

Productions for Expressions

In the `parser.y` file, expression productions define how to parse and handle expressions in C when converting to Python. Expressions can be variables, numbers, format strings, function calls, or arithmetic and logical operations.

Productions follow operator precedence rules (specifically analyzed in the next paragraph), ensuring that operations are performed in the right order. For example, productions for addition, subtraction, multiplication, and division operations consider operator precedence and create nodes in the AST that represent those operations.

Additionally, the productions handle logical expressions, such as NOT, AND, and OR, comparisons such as greater than (">"), less than ("<"), equal to ("=="), and not equal to ("!="). Production for parentheses allows you to define groups of expressions, ensuring the correct order of operations.

Finally, productions address unary expressions, such as the negation operator ("-"), with specific precedence.

These productions are critical to ensuring that C expressions are translated correctly into Python, respecting the differences in syntax and semantics between the two languages. Each production creates nodes in the AST that reflect the structure of the original expression, allowing the transpiler to generate equivalent Python code accurately.

Operator Precedence

The precedence of C operators affects the grouping and evaluation of operands in expressions. The precedence of an operator is significant only if there are other operators with higher or lower precedence. Expressions with higher precedence operators are evaluated first. To avoid conflicts and ambiguities in the grammar of this project, obviously, it was necessary to use the precedence operators.

```
    // PRECEDENCE AND ASSOCIATIVITY
%left    '+' '-' '*' '/'
%right    '='
%left    AND OR NOT
%nonassoc    MINUS
%left    '>' '<' GE LE EQ NE
%left    '[' ']' '(' ')' ','
```

Table 2.9: Operator precedence.

Generation of Abstract Syntax Tree nodes

During this phase, the Abstract Syntax Tree is created in memory. Generally among the actions associated with productions, there is the creation of the related node of the Abstract Syntax Tree and the connection with its child nodes.

2.3 Abstract Syntax Tree

The Abstract Syntax Tree (AST) is a representation of grammar constructs in the form of a tree. Each node of the tree denotes a construct occurring in the source code. Abstract syntax trees are data structures widely used in compilers to represent the structure of program code. This data structure is indispensable for the semantic analysis phase and then for the translation phase.

2.3.1 Structure of a generic Abstract Syntax Tree node

It was chosen to have a single structure for the nodes of AST, called `AstNode`, to simplify the creation and management of AST. The structure of this node is visible within the file `ast.h` and includes:

- `node_type` It is the type of node on which the enhanced structures within the union node depend. The 10 possible values are: `EXPR_T`, `IF_T`, `VAL_T`, `VAR_T`, `DECL_T`, `RETURN_T`, `FCALL_T`, `FDEF_T`, `FOR_T`, `ERROR_NODE_T`.

AstNode		
node_type	node	next

Table 2.10: Structure of `AstNode`.

- `node` It is a `union` containing different structures related to different types of node. The pointers to the child nodes of the current node are contained within such structures.
- `next` It is the pointer to the sibling node of the current node, which will be in turn `AstNode` type and is enhanced through the function `link_AstNode()`.

2.3.2 Structure of each specific node

The following are the structures of the different node types. Each one is used for a particular piece of our restriction of source language.

variable		
name	by_reference	array_dim

Table 2.11: Structure of `variable`.

expression		
expr_type	l	r

Table 2.12: Structure of `expression`.

funcCall	
name	args

Table 2.13: Structure of `funcCall`.

value	
val_type	string_val

Table 2.14: Structure of `value`.

funcDef			
ret_type	name	params	code

Table 2.15: Structure of `funcDef`.

declaration	
type	var

Table 2.16: Structure of `declaration`.

ifNode		
cond	body	else_body

Table 2.17: Structure of `ifNode`.

forNode			
init	cond	update	stmt

Table 2.18: Structure of `forNode`.

returnNode
expr

Table 2.19: Structure of `returnNode`.

2.4 Symbol Table

The Symbol Table is a data structure used by the compiler to track the semantics of identifiers. Each identifier (or symbol), constant, procedure and function in a program's source code is associated with information relating to its declaration or appearance in the source. In other words, the entries of a symbol table store the information related to the entry's corresponding symbol.

The structure chosen for the Symbol Table is a hash table because of its efficiency and simple code management. To manage the hash table, the `uthash.h` support library was chosen. In particular, it allows to:

- Add an item to a hash.
- Look up an item in a hash.
- Delete an item from a hash.

2.4.1 Structure of records of Symbol Table

Each Symbol Table contains a record for each identifier, with fields for attributes. Specifically, records are created during the parsing phase. The record structure is located within the `symbol_table.h` file, and is represented by the `struct symbol`. Its fields are as follows:

name	type	sym_type	params_list	used	array	lineno	line

Table 2.20: Structure of records of Symbol Table.

- `name` It represents the symbol name and it is used as key for the hash table.
- `type` It represents the type for variables and parameters, while for functions it represents the return type, as well as for the special symbol `return`.
- `sym_type` It is the symbol type. The possible types are `VARIABLE`, `FUNCTION`, `PARAMETER` and `F_RETURN`. This symbol is inserted within the scope table of a function. When the `return` statement is called, it is used to check the coherence of the type returned by a function with its declaration.
- `pl` It is a pointer to the function parameter list.
- `used` It is a flag indicating whether or not a variable is used within a scope.
- `array` It is flag indicating whether a variable or parameter is an array or not.
- `lineno` It specifies the line number in which the symbol is declared, in order to give more information in case of error.
- `line` It stores a copy of the line in which the symbol is declared, in order to give more information in case of error.

2.4.2 Operations on Symbol Table

In `symbol_table.c`, there are several functions related to possible operations on the Symbol Table.

Functions for table operations

- `create_symbol_table`: function to create a new symbol table;
- `delete_symbol_table`: function to delete a symbol table;

- `find_symbol_table`: function to search symbols within all currently open scopes starting from the current scope;
- `print_symbol_table`: function to print the symbol table;
- `check_usage`: function to check if all the variables present in the Symbol Table have been used.

Functions for symbol operations

- `find_symbol`: function to search a symbol in the current scope;
- `insert_symbol`: function to insert a symbol in the current scope;
- `remove_symbol`: function to remove a symbol from the passed symbol table.

2.4.3 Scope management

Regarding scope management, it was chosen to use a separate symbol table for each scope of the program. Their hierarchy is managed by a stack, that is implemented as a linked list. The list structure is defined in the `symbol_table.h` file and involves the `struct symbol_list`:

scope	symbol_table	next

Table 2.21: Structure of `symbol_table`.

- `scope` It represents the scope number.
- `symbol_table` It is a pointer to the current table.
- `next` It is a pointer to the successive symbol table.

Functions for scope management

The operations required for scope management are:

- `scope_enter()`: This function creates a new empty symbol table thanks to the `create_symbol_table()` function and then increments the current scope level. If the new scope is a function scope, it inserts possible function parameters and return type in `current_symbol_table`.
- `scope_exit()`: This function closes the current scope and decreases the current scope level. Before doing this, it calls the `check_usage` to check if there are some variables present in the Symbol Table which have never been used.

2.5 Semantic Analysis

2.5.1 Array Checks

This section of code in the `semantic.c` file handles checks on arrays, verifying the correctness of the dimensions or indices of arrays within the C source code. Arrays are a common data structure in C, but in Python they are usually handled in a different way.

- In the `check_array` function, the main check is made on the dimensions or indices of the array. If a dimension or index is negative, an error is thrown because C arrays and Python lists do not support negative indexes and using negative dimensions can result in unexpected behavior.
- The `check_array_dim` function is used to check whether only integer values and integer variables are used in `array_dim` expressions. The use of mathematical and comparison operators is permitted, but the use of assignment operators is not permitted. The function is intended to ensure that the dimensions specified for arrays consist of integer values and variables, and that they are consistently defined using mathematical and comparison operators. If an expression does not meet these requirements, the function will generate an error to indicate the non-compliance.
- Finally, the `eval_array_dim` function is used to evaluate constant expressions of type integer. This is important to get the actual expression of the size or index of the array, which will then be used to generate the Python code.

2.5.2 Expression Checks

In the `semantic.c` file we have also implemented a series of functions to evaluate constant expressions, check divisions by zero and ensure that operations are consistent.

- One of the key functions is `eval_constant_expr`, which evaluates the result of a constant expression. This function considers the data types and operations involved to calculate the correct result. It is used to handle arithmetic operations on constants and to ensure that the results are compatible with Python semantics.
- Another crucial function is `check_division`, which checks an expression for divisions by zero. If a division by zero is detected, this function generates an error, ensuring that the behavior in Python mirrors that in C.
- Regarding the evaluation of the type of an expression and whether it is constant or not, the `eval_expr_type` function plays a decisive role. It returns a `complex_type` value that contains information about the type and constancy of the expression. This is critical to ensuring that operations are appropriate.

- Additionally, the `kind` field in this data structure indicates whether an expression is of type "`N`" or "`V`". `kind = N` indicates that the expression is constant, that is, a value known at compile time. This could be an integer or float or a string of characters. In other words, the expression does not depend on dynamic variables or values during program execution. `kind = V` indicates that the expression is of type variable, meaning that it depends on one or more variables or dynamic values during program execution. In this case, the result of the expression may change based on the values of the variables involved. The `type` field represents the data type of the expression, which can be an integer type (`INT_T`), a float type (`FLOAT_T`), a string (`CHAR_T`), or even an error type (`ERROR_T`) in case the expression is invalid or not compatible with the intended operations.
- Additional functions such as `eval_comparison_op_type` are responsible for evaluating the types resulting from comparison and arithmetic operations, respectively. These functions help ensure that operands are compatible and that operations are defined on the types involved.

Overall, this section of code was designed to address expression and operation challenges when translating from C to Python.

2.5.3 Assignment Checks

The `eval_ass_op` function is responsible for evaluating the result of an assignment operation based on the data types involved. To do this, it takes into consideration the types of the operands involved, i.e. the variable to which a value is being assigned (`l`) and the value or expression itself to be assigned (`r`). If the expression is not constant, the two operands must be of the same type. If the expression is constant, this function checks whether it can be represented in the type of the target variable.

In particular, the function handles dynamic typing and assignment checking. Ensures that the types of the variables are compatible and returns an error if they are not. It also takes into account special situations, such as the possibility of possible truncation of constants and the treatment of special types such as `CHAR_T` and `STRING_T`.

2.5.4 Function Calls Checks

The section of code within the `semantic.c` file that deals with checks on function calls plays an essential role in ensuring consistency between function calls in the C source code and the definitions of the functions themselves. This function, called `check_fcall`, evaluates whether function calls are correct in terms of declaration and argument types.

The function begins by searching the current symbol table for the function name. If the function has not been declared, an error will be thrown indicating that the function is undefined. Next, it checks whether the symbol corresponding to the name is actually a function, reporting an error if the symbol found does not correspond to a function.

Processing continues by comparing the types and quantities of arguments passed to the function with the parameters of the function declaration. The `check_fcall` function performs detailed checks to ensure that the types of the arguments are compatible with the types declared in the function and handles scenarios where the arguments can be variables or constants. It also examines constant truncation if you assign values to variables of different types.

In case of any inconsistency between function calls and definitions, `check_fcall` will generate appropriate errors to notify the programmer of the detected discrepancies.

2.5.5 Function Declarations Checks

The section of code in the `semantic.c` file that addresses checks on function declarations plays an essential role in ensuring the semantic correctness of functions defined in the C source code and their calling. The `check_return` function is responsible for verifying that the return type of a function matches the one declared previously and is also responsible for handling any errors resulting from an inconsistent declaration.

The function begins by looking for the return symbol in the current symbol table, which represents the return type of the function currently under consideration. Next, `check_return` performs several checks based on the presence or absence of an expression in the return statement:

- If there is no expression, but the function should return a value (the return type is not void), an error is raised indicating that the return value is missing.
- If an expression is present, the function evaluates the return type of the expression through the `eval_expr_type` function and compares it to the declared type of the function. If there is a mismatch between the types, errors are generated.
- In case the expression is constant and must be assigned to a variable of a different type (for example, returning a float value in a function declared with return type int), the `check_return` function takes care of managing the truncation of constants, if necessary, reporting errors.

Furthermore, the `check_func_return` function checks that all functions have an execution path that returns a value, if not declared with type void. It examines the body of the function to ensure that it contains a return statement or a series of conditional statements (for example, within an if structure) that lead to a return. If a function does not return a value, an error is raised indicating the lack of a return at the end of the function.

2.5.6 `printf` and `scanf` Functions Checks

The section of code in the `semantic.c` file dedicated to checks on the `scanf` and `printf` functions plays a critical role in ensuring consistency between the types specified in the format specifiers and the arguments passed to these functions in calls within the C source code.

The `check_format_string` function takes the format string (`format_string`) and arguments passed to the function as arguments. The `f_type` argument indicates whether the called function is `scanf` or `printf`, and specific checks are performed based on this. The function extracts the format specifiers from the format string and compares them to the types of the arguments.

Format Specifier (C)	Python Equivalent	Explanation
%s	%s	Represents a string in C, and in Python, it is used as a placeholder in the string formatting operation.
%d	%d	Represents an integer in C and Python.
%i	%d	%i is analogous to %d and represents an integer.
%f, %F	%f, %F	Represents a floating-point number in C and Python.
%e, %E	%e, %E	Represents a number in scientific notation in C and Python.
%g, %G	%g, %G	Represents a number in floating point or scientific notation, whichever representation is more compact.
%l	%l	Represents a long integer in C, and in Python, it is used as a placeholder in the string formatting operation.
%%	%	%% in C represents a literal '%' character, which is simply translated as '%' in Python, since it is not a placeholder.

Table 2.22: Format specifiers.

In the case of `scanf`, format specifiers should correspond to pointers to appropriate types. The function checks whether the reference operator (`&`) is used with `scanf` arguments. Otherwise, a warning is generated, because `scanf` requires arguments by reference. Additionally, type checks are performed to ensure that format specifiers match the passed arguments.

For `printf`, the format specifiers must be compatible with the arguments passed. The function checks if there are incompatibilities between the arguments and format specifiers, for example, if an integer is requested but another type is passed. Errors are generated to report any discrepancies.

Finally, the function also checks whether the number of arguments passed to the function matches the number of format specifiers. If there are too many or too few arguments, appropriate errors are generated.

If format specifiers other than those indicated in the table are used (such as `%a` and `%t`, for example), an error will be reported, indicating that the character type is not recognized in the `format_string`.

2.5.7 Variable Declarations Checks

The section of code in the `semantic.c` file dedicated to checks on variable declarations plays an essential role in ensuring that variables are used correctly and that they have been previously declared. The `check_var_reference` function is responsible for performing these checks and setting the `used` flag of the variable when it is actually used.

The function begins by searching for the symbol corresponding to the variable within the current symbol table using the variable name. If the variable is not found, an error is generated indicating that the variable was used without having been previously declared.

If the variable is found in the symbol table, the variable's `used` flag is set to 1, indicating that the variable has been used within the code. This is useful for detecting variables that have been declared but never used, which may be an error or a problem with unoptimized code.

2.6 Code generation

The code generation process in our transpiler follows a well-defined approach, consisting of two distinct phases:

- During the syntax analysis phase, we create a tree representation called "Abstract Syntax Tree" (AST).
- Next, we use the AST as the basis to generate the final source code in the target language.

In particular, our transpiler operates on an AST as an intermediate representation, an appropriate choice for a source-to-source compiler. This means that before translating the source code into another language, we verify that the original code is correctly analyzed and represented in the form of AST. In the first phase, therefore, the transpiler examines the source code provided as input. During this process, the transpiler identifies various elements of the language, including identifiers, declarations, expressions, and instructions. While the source code is analyzed, the transpiler builds the AST. This tree represents the logical structure of the source program and captures the relationship between different parts of the code. When creating the AST, the transpiler checks

that the source code respects the syntactic rules of the source language. If syntax errors occur, error messages are reported to indicate the location and nature of the errors.

The code translation, therefore, occurs only if the previous analysis phases have not detected any errors. For the translation part, we have developed a series of specific functions and logic that take care of converting the AST into source code into the target language. These functions can be found in files named `translate.h` and `translate.c`.

Our code generation process is therefore based on initial syntactic analysis, followed by the creation of an AST and finally translation into final source code only if the initial source code is error-free.

To keep track of the presence of the `main` function in the source file, a variable called `main_flag` was introduced. This variable plays a crucial role in the process of transposing from the source language to Python. Specifically, `main_flag` is used to determine whether the `main` function is defined in the original source code.

During the analysis and translation phase, the transpiler examines the source code looking for the `main` function declaration. If the `main` function is found, the transpiler sets the `main_flag` variable to a value that indicates that the `main` function is present. In that case, the transpiler is aware that the `main` function is intended to be the entry point of the translated program. Subsequently, when the translated Python code is executed, the condition if `__name__ == "__main__":` is used to allow the `main` function to run. This was necessary to have the equivalent of defining a `main` function in Python as well. In Python, in fact, it is not necessary to specify a return type for the main function.

In C, you must explicitly declare variables, including the data type, before using them. For example, you can declare an integer variable in C as `int number = 42;`. Python, on the other hand, is a dynamically typed language, which means that you don't need to explicitly declare the type of a variable. You can assign an integer value to a variable without declaring the type: `number = 42`. Additionally, in C you have to manually handle conversion between types when necessary. For example, to convert an `int` to a `double`, you must use a cast. In Python, conversion between types is often handled implicitly. For example, you can perform operations between `int` and `float` without having to do explicit casts. Other than this, the `int` type in C and `int` in Python are similar in their basic utility, which is to represent integers. The `float` type in C and the `float` type in Python are also similar in their basic function, which is to represent floating-point numbers. However, the `float` type in C language is a single-precision floating-point number, which typically uses 32 bits (4 bytes) for representation. This offers limited precision compared to double-precision floating-point numbers (`double`). Python floats are going to be identical to C

`double` s: in Python, in fact, the float type uses double-precision floating-point arithmetic, usually represented with 64 bits. This offers high accuracy. It is for this reason that we chose to translate the `float` type in C with the float type in Python.

In C, strings are often represented as arrays of characters terminated by a null character, and pointers are used to access such strings. A char pointer is a variable that stores the memory address of a variable of type char or a sequence of characters (such as a string). A pointer does not have a fixed size and is used to refer to other variables or data in memory. Pointers can be used to create, manipulate, and access character strings dynamically. An example of declaring and using a character pointer is as follows: `char *myPointer = "World";` . So a character array is a static data structure with a fixed size, while a character pointer is a variable that can refer to char or string variables and offers more flexibility for string manipulation. In this project we chose to translate char pointers into Python strings. In Python, by contrast, a string is a built-in data type and can be declared and used without having to directly manage characters or memory. For example: `string = "Hello"`. Individual characters of a string can be accessed in Python using indexing, for example `string[0]` will return the first character. Additionally, strings in Python can be iterated directly with a `for` loop.

In C, it is common to declare variables explicitly, specifying their type and name before using them. In Python, explicit variable declarations are not necessary. Variables are created dynamically when you assign a value to them. For simple variable declarations, we implemented the translation like this: for example, in the case of `int a, b;`, in our transpiler it becomes `a, b = int(), int()`, also allowing multiple declarations on the same line. This way, variables are initialized with an object of type `int` in Python.

For array declarations, we followed a similar strategy. For example, for `int v1[size];`, in our transpiler we translated the declaration to `v1 = [int()] * int(size)`. This creates a list in Python where each element is initialized as an `int` object, replicated size times. It is important to note that, unlike C, multiple array declarations on the same line, such as `int v1[size1], v2[size2];`, are not supported in our transpiler. As for arrays, it must be said that there is no exact equivalent of this data structure in Python. However, Python offers a data structure called "`list`" that is often used to group elements. Lists in Python are dynamic and can contain elements of different types. We therefore chose to translate arrays into Python lists.

In C, the `for` loop is very flexible and allows you to specify initializations and updates based on a control variable, along with arbitrary conditions to determine its execution. In Python, the `for` loop is based on the `range` function, but this function imposes limits on the definition of conditions. The `range` function creates a numeric sequence, but the termination condition of the for

loop is determined by the number of elements in the sequence. This can make directly translating C for loops into Python difficult, especially if there are complex initializations or updates based on different variables.

Therefore, to address this challenge, we chose to translate C's for loops into Python's while loops. This approach allows you to maintain the flexibility of C for loops and handle more complex initializations and updates. In this way, the semantic equivalence between the original loop in C and the one translated into Python was preserved, while adapting the loop structure to the Python context.

The `scanf` and `printf` functions are considered to be "built-in functions" (or "internal library functions"). This means that these functions are available by default in the source language (in this case, C) without the need to include external libraries or manually declare these functions. Their translation was handled in a congenial manner, therefore translating them both with their respective equivalent functions in Python, namely `input()` and `print()`.

Chapter 3

Test

After implementing the transpiler, it is critical to test the generated code to ensure that it works properly and produces consistent results with the source code in C. Testing must cover a wide range of scenarios, including limit cases and complex situations.

In our case, we verified the correct translation of the C source code into Python code through a series of tests carried out, for example, on declarations, expressions, for loops, functions, printf, scanf, scope and lot of limit cases regarding each test object.

Test files are organized in the test folder within the project's root directory. The tests are organized hierarchically by a subdivision into sub-folders that identify the test object. In each subfolder, we distinguish valid tests from errors and any tests that present warnings.

3.1 Failure on variable declarations with assignment

In the design phase of the grammar, even the declarations with assignments have been included beyond the normal variable declarations, single or multiple. The grammar rule is the following:

```
   var_declaration

:  type declarator_list ';' {$$ = new_declaration_node(DECL_T, $1, $2);
fill_symbol_table(current_symbol_table, $$, -1, VARIABLE); }

| type initializer_list ';' { $$ = new_declaration_node(DECL_T, $1,
$2); fill_symbol_table(current_symbol_table, $$, -1, VARIABLE); }

| ID declarator_list ';' { $$ = new_error_node(ERROR_NODE_T);
```

```
yyerror(error_string_format("Unknown type name:  " BOLD "%s" RESET,
$1 )); } ;
```

In particulare, the grammar rule `var_declaration` includes a component for declarations without assignment and a component for declarations with assignment.

The first component uses the non-terminal `declarator_list` and it is structured in the following way:

```
    declarator_list
:  var
| var ',' declarator_list { $$ = link_AstNode($1, $3); }
;
```

The second component uses the non-terminal `initializer_list` and it is structured in the following way:

```
    initializer_list
:  var
| var ',' initializer_list { $$ = link_AstNode($1, $3); }
;
```

Moreover, the non-terminal `var` is structured in the following way:

```
    var
:ID {$$ = new_variable_node(VAR_T, $1, NULL); }
|ID '[' expr ']' {$$ = new_variable_node(VAR_T, $1, $3); check_array($3);
}
;
```

Currently, the compiler cannot recognize variable declarations with assignments, generating a syntax error.

Chapter 4

User Guide

This guide is designed for the use of transpiler in Linux environment.

4.1 Compilation

Source codes must be compiled before the transpiler can be used. This can be done manually or by using the Makefile.

Makefile compilation

1. Access the directory `transpiler_C2Python` from the terminal.
2. Run the `make` command that will allow you to automatically compile the source files.

Manual compilation

1. Access the directory `transpiler_C2Python` from the terminal.
2. Generate `parser.c` and its relative header file with the command: `bison -defines=token.h -o parser.c parser.y`.
3. Generate `scanner.c` with the command: `flex -o scanner.c scanner.l`.
4. Compile the transpiler with the command `gcc parser.c scanner.c ast.c symbol_table.c semantic.c translate.c -o compiler`.

4.2 Usage

The transpiler can be executed with the following command:

```
./compiler path_input
```

After execution, in absence of errors, the target program will be generated in the file `translation.py`.

It is possible to display a guide for the use of the transpiler with the flag `--help` or `-h`.

```
./compiler --help
```

The compiler flags were used during the debugging phase to verify the correct behavior.

- `-s`: this flag allows to show on the standard output the contents of the symbol table when closing each scope.
- `-t`: this flag allows to show on the standard output a representation of the source code obtained through the abstract syntax tree.

Chapter 5

Final considerations

The project of a C to Python transpiler was certainly one of the most complicated and challenging ever undertaken by the Team.

The Team is satisfied with the results obtained and with the achievement of the main objectives set. In the future, the transpiler could be extended to support more C language features, and with further development and optimizations, it could become an extremely useful tool for developers who want to leverage the power of Python while maintaining their existing C code.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2007.

[2] D. Thain. *Introduction to Compilers and Language Design (2nd Edition)*. LULU Press, 2020.

[3] Floriano Scioscia. *Formal Languages and Compilers - lecture materials*. URL: `https://sisinflab.poliba.it/people/floriano-scioscia/`. 2022.

[4] *Lexical Analysis With Flex, for Flex 2.6.2*. URL: `https://westes.github.io/flex/manual/`. 2022.

[5] Free Software Foundation. *GNU Bison - The Yacc-compatible Parser Generator*. URL: `https://www.gnu.org/software/bison/manual/`. 2021.

[6] John R. Levine. *Flex & Bison*. O'Reilly Media, Inc., 1st edition, 2009.

[7] Anthony A. Aaby. *Compiler Construction using Flex and Bison*. 2004.

[8] Troy D. Hanson, Arthur O'Dwyer. *uthash User Guide*. URL:`https://troydhanson.github.io/uthash/userguide.html`. 2021.

[9] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language (2nd edition)*. Prentice Hall Professional Technical Reference, 1988.

[10] *Python 3.11.3 documentation*. URL: `https://docs.python.org/3.11/`.