

Toy Compiler

Luigi Crisci

Alessio Cuccurullo

Abstract

A compiler for the Toy language in pure Java. It compiles to C as Intermediate code representation, then the *Clang* compiler is used to generate machine code

Contents

Toy language Compiler	3
Build	3
Requirements	3
Build instructions	3
Assignment overview	3
Differences with the assignment	3
Lexical analysis	5
Lexical specification	5
Syntactic analysis	7
Syntactic specification	7
The Abstract Syntax Tree	10
Tree visualization	10
Semantic Analysis	11
Inference table	11
Inference rules	11
Translating to the C language	13
Notably differences between Toy and C	13
Multiple return types	13
Memory allocation for strings	14
Strings spanning on multiple lines	14
Boolean shenanigans	15

Toy language Compiler

A compiler for the Toy language, made with Java CUP and JFlex. This implementation results in the creation of an executable file given a compliant Toy file.

Build

Requirements

- Java 11
- Maven

Build instructions

To build this project, just type

```
mvn package
```

A jar *with all dependencies* will be created under the **release/** directory.

Note that code generation for *Javacup* and *jflex* is disabled by default. If you want to manually generate the sources for Javacup and Jflex, type “*mvn generate-sources -P source-gen*” and the .java classes will be placed under *target* folder.

A compiled version of this compiler can be found under the **release/** directory. The release is composed of:

- ***Toy2C.jar***: a pre-compiled version of this software
- ***Toy2C***: an executable script that, given a .toy file, compiles it into a .c file and then call clang to output a machine-code executable
- ***install_native_libraries***: an executable script that create a hard link under `/usr/local` directory for the `toy_functions.h` library, a native library needed in the compilation phase.
- ***uninstall_native_libraries***: an executable script that remove the hard link under `/usr/local` created by *install_native_libraries*.

An example of the *Toy2C* usage:

```
./Toy2C FILENAME
```

The output file is placed in the same directory of the input file.

Assignment overview

This compiler translates a .toy program into a Clang-compliant C program. The generated .c file is then compiled and, after that, it's ready to run. The stages of the execution are the following:

- Lexical analysis
- Syntactic analysis
- (Optional) AST visualization
- Semantic analysis
- Toy2C translation

Differences with the assignment

This implementation doesn't go that far from the assignment. Although, some variations have been made by the authors:

- The token MAIN has been introduced;
- The productions “*Main ::= PROC MAIN ...*” have been added, slightly changing the syntax of the language. These productions, along with the *ProcList*, make every Toy program syntactically compliant when the procedure *Main*:
- appears one single time,

- is the last one in the file,
- returns an INT.
- The production “CallProcedureStatement” has been duplicated in the production “CallProcedure-Expression” because of a parsing problem derived from our class configuration

Lexical analysis

This step is carried out by a Lexer written in **flex** and compiled with **Jflex** (an open source tool for generating Lexer in Java).

It is composed of a single **Lexer.flex** source file containing all the logic to generate a Lexer class, which is crucial for the next stage. The Lexer resulting by the compiling does the whole Lexical analysis.

Lexical specification

This section describes the set of tokens and their corresponding pattern.

```
1 //Procedures
2 PROC "proc"
3 CORP "corp"
4 MAIN "main"
5
6 //Type
7 INT "int"
8 FLOAT "float"
9 BOOL "bool"
10 STRING "string"
11 VOID "void"
12
13 //Statements
14 WHILE "while"
15 DO "do"
16 OD "od"
17 READ "readln"
18 WRITE "write"
19 ASSIGN "assign"
20 IF "if"
21 THEN "then"
22 FI "fi"
23 ELSE "else"
24 ELIF "elif"
25
26 //Separators
27 LPAR "("
28 RPAR ")"
29 COLON ":"
30 COMMA ","
31 SEMI ";"
32
33
34 //Operators
35 ASSIGN ":="
36 PLUS "+"
37 MINUS "-"
38 TIMES "*"
39 DIV "/"
40 EQ "="
41 NE "<>"
42 LT "<"
43 LE "<="
44 GT ">"
```

```
45     GE ">="
46     AND "&&"
47     OR "||"
48     NOT "!"
49     NULL "null"
50     TRUE "true"
51     FALSE "false"
52     RETURN "->"
53
54
55     ALPHA=[A-Za-z]
56     DIGIT=[0-9]
57     NONZERO_DIGIT=[1-9]
58     NEWLINE=\\r\\n|\\r\\n\\n
59     WHITESPACE = | [ \\t\\f]
60     ID = (|_)*
61     INT = ((*)|0)
62     FLOAT = +)
63     STRING_TEXT = [^\\"]*
64     COMMENT_TEXT = [\\w\\.\\@]*
```

Syntactic analysis

Just like the lexical analysis, the syntactic analysis is done by a generated Java class. This has been done with CUP (Construction of Useful Parsers), which implements *LALR(1)* parsing.

Syntactic specification

This section describes the whole syntactic specification of the Toy language that was implemented. The grammar as-is doesn't allow *LALR(1)* parsing. In order to generate the parser, the *PEM-DAS* (Parenthesis, Exponents, Multiplications/Divisions, Additions/Subtractions) rule has been introduced.

```
1 Program ::= VarDeclList ProcList
2         ;
3
4     VarDeclList ::= /* empty */
5                 | VarDecl VarDeclList
6                 ;
7
8     VarDecl ::= Type IdListInit SEMI
9             ;
10
11    ProcList ::= Main
12             | Proc ProcList
13             ;
14
15    Type ::= INT
16          | BOOL
17          | FLOAT
18          | STRING
19          ;
20
21    IdListInit ::= ID
22               | IdListInit COMMA ID
23               | ID ASSIGN Expr
24               | IdListInit COMMA ID ASSIGN Expr
25               ;
26
27    Proc ::= PROC ID LPAR ParamDeclList RPAR ResultTypeList COLON
28           VarDeclList StatList RETURN ReturnExprs CORP SEMI
29           | PROC ID LPAR RPAR ResultTypeList COLON
30           VarDeclList StatList RETURN ReturnExprs CORP SEMI
31           | PROC ID LPAR ParamDeclList RPAR ResultTypeList COLON
32           VarDeclList RETURN ReturnExprs CORP SEMI
33           | PROC ID LPAR RPAR ResultTypeList COLON
34           VarDeclList RETURN ReturnExprs CORP SEMI
35           ;
36
37    Main ::= PROC MAIN LPAR ParamDeclList RPAR INT COLON
38           VarDeclList StatList RETURN ReturnExprs CORP SEMI
39           | PROC MAIN LPAR RPAR INT COLON
40           VarDeclList StatList RETURN ReturnExprs CORP SEMI
41           | PROC MAIN LPAR ParamDeclList RPAR INT COLON
42           VarDeclList RETURN ReturnExprs CORP SEMI
43           | PROC MAIN LPAR RPAR INT COLON
44           VarDeclList RETURN ReturnExprs CORP SEMI
45           ;
```

```

46
47 ResultTypeList ::= ResultType
48     | ResultType COMMA ResultTypeList
49     ;
50
51 ResultType ::= Type
52     | VOID
53     ;
54
55 ReturnExprs ::= ExprList
56     | /* empty */
57     ;
58
59 ParamDeclList ::= ParDecl
60     | ParamDeclList SEMI ParDecl
61     ;
62
63 ParDecl ::= Type IdList
64     ;
65
66 IdList ::= ID
67     | IdList COMMA ID
68     ;
69
70 StatList ::= Stat
71     | Stat StatList
72     ;
73
74 Stat ::= IfStat SEMI
75     | WhileStat SEMI
76     | ReadlnStat SEMI
77     | WriteStat SEMI
78     | AssignStat SEMI
79     | CallProc SEMI
80     ;
81
82 WhileStat ::= WHILE StatList RETURN Expr DO StatList OD
83     | WHILE Expr DO StatList OD
84     ;
85
86 IfStat ::= IF Expr THEN StatList ElifList Else FI
87     ;
88
89 ElifList ::= /* empty */
90     | Elif ElifList
91     ;
92
93 Elif ::= ELIF Expr THEN StatList
94     ;
95
96 Else ::= /* empty */
97     | ELSE StatList
98     ;
99
100 ReadlnStat ::= READ LPAR IdList RPAR
101     ;

```



```

102
103 WriteStat ::= WRITE LPAR ExprList RPAR
104         ;
105
106 AssignStat ::= IdList ASSIGN ExprList
107         ;
108
109 CallProc ::= ID LPAR ExprList RPAR
110           | ID LPAR RPAR
111           ;
112
113 ExprList ::= Expr
114           | Expr COMMA ExprList
115           ;
116
117 Expr ::= NULL
118       | TRUE
119       | FALSE
120       | INT_CONST
121       | FLOAT_CONST
122       | STRING_CONST
123       | ID
124       | ID LPAR ExprList RPAR
125       | ID LPAR RPAR
126       | Expr1 PLUS Expr2
127       | Expr1 MINUS Expr2
128       | Expr1 TIMES Expr2
129       | Expr1 DIV Expr2
130       | Expr1 AND Expr2
131       | Expr1 OR Expr2
132       | Expr1 GT Expr2
133       | Expr1 GE Expr2
134       | Expr1 LT Expr2
135       | Expr1 LE Expr2
136       | Expr1 EQ Expr2
137       | Expr1 NE Expr2
138       | MINUS Expr
139       | NOT Expr
140       ;

```

The Abstract Syntax Tree

The implemented Grammar has been enhanced with a series of *actions*, one for each production.

Generally, these actions instantiate a *Node* object related to each *element* appearing in the right-hand side of the production itself. There are several different actions in this parser.

For example:

```
1 VarDeclList ::= /* empty */  {: RESULT = new
    LinkedList<VariableDeclarationNode>(); :}
2 | VarDecl VarDeclList      {: v1.add(vd); RESULT = v1; :}
3 ;
```

The *empty* production creates a new list containing the variables declared, while the second one appends a given variable to the list. In a successful situation, one would expect the process to eventually resolve in the *empty* statement, thus instantiating the list and adding the items found.

As the *Parser* elaborates a given source file, the corresponding (and unique) *Syntax Tree* is generated recursively. With the completion of the parsing process, a pointer to the root of the *Syntax tree* is returned, which comes in handy for the next step of the compiler.

Tree visualization

Moreover, this implementation provides with a visualization of the *Syntactic Tree*, constructed in the previous step, via XML. The *Visitor* pattern fits this role perfectly. Once the parser has finished its job, the user can call the *ASTVisitor* on the root of the tree, which will generate a .xml file based on the instance of the *Syntactic tree*. The user can open the generated file using any Web Browser.

Semantic Analysis

In order to check whether the input program is semantically compliant, a *SemanticVisitor* object is created and invoked on the root of the AST generated by the parser. Given the set of rules, a single visit of such tree is sufficient towards the semantic analysis.

Inference table

The following tables describe each and every inference rule required to the type checking. The implementation of these rules can be found in the *TypeCheck* class.

Binary_op	First operand type	Second operand type	Resulting type
:=	int	int	int
:=	float	float	float
:=	string	string	string
:=	boolean	boolean	boolean
/ + -	int	int	int
/ + -	int	float	float
/ + -	float	int	float
/ + -	float	float	float
<= < == <> > >=	int	int	boolean
<= < == <> > >=	int	float	boolean
<= < == <> > >=	float	int	boolean
<= < == <> > >=	float	float	boolean
&&	boolean	boolean	boolean

Unary_op	Operand type	Resulting type
-	int	int
-	float	float
!	boolean	boolean

Inference rules

These are the inference rules implemented in the *SemanticVisitor* class.

- Expressions

$$\frac{(expr : \tau) \in \Gamma}{\Gamma \vdash expr : \tau}$$

- Binary operation

$$\frac{\Gamma \vdash expr_1 : \tau_1 \quad \Gamma \vdash expr_2 : \tau_2 \quad \Gamma \vdash binary_op(binOp, \tau_1, \tau_2) = \tau}{\Gamma \vdash (expr_1 binOp expr_2) : \tau}$$

- Unary operation

$$\frac{\Gamma \vdash expr : \tau_1 \quad \Gamma \vdash unary_op(unOp, \tau_1) = \tau}{\Gamma \vdash (unOp expr) : \tau}$$

- Call procedure

$$\frac{\Gamma \vdash proc : \tau_i^{i \in 0 \dots n} \rightarrow \tau_j^{j \in 0 \dots m} \quad \Gamma \vdash par_i^{i \in 0 \dots n} : \tau_i}{\Gamma \vdash proc(par_i^{i \in 0 \dots n}) : \tau_j^{j \in 0 \dots m}}$$

- While statement

$$\frac{\Gamma \vdash cnd_expr : \mathbf{boolean} \quad \Gamma \vdash while_stmt_i^{i \in 1 \dots n} \quad \Gamma \vdash do_stmt_j^{j \in 0 \dots n}}{\Gamma \vdash \mathbf{while} (while_stmt_i^{i \in 1 \dots n}) \mathbf{do} (do_stmt_j^{j \in 0 \dots n}) \mathbf{od}}$$

- Assign statement

$$\frac{(x_i^{i \in 1 \dots n} : \tau_i) \in \Gamma \quad \Gamma \vdash (expr_j^{j \in 1 \dots n}) \rightarrow \tau_i^{i \in 1 \dots n}}{\Gamma \vdash x_i^{i \in 1 \dots n} = expr_j^{j \in 1 \dots n}}$$

- If statement

$$\frac{\Gamma \vdash cnd_expr : \mathbf{boolean} \quad \Gamma \vdash then_stmt_x^{x \in 0 \dots n} \quad \Gamma \vdash elif_stmt_y^{y \in 0 \dots n} \quad \Gamma \vdash else_stmt_z^{z \in 0 \dots n}}{\Gamma \vdash \mathbf{if} \ cnd_expr \ \mathbf{then} \ (then_stmt_x^{x \in 0 \dots n}) \ \mathbf{elif} \ (elif_stmt_y^{y \in 0 \dots n}) \ \mathbf{else} \ (else_stmt_z^{z \in 0 \dots n}) \ \mathbf{fi}}$$

- Write statement

$$\frac{\Gamma \vdash expr_i : \tau_i^{i \in 0 \dots n}}{\Gamma \vdash \mathbf{write}(expr_i^{i \in 0 \dots n})}$$

- Read statement

$$\frac{(x_i : \tau_i^{i \in 0 \dots n}) \in \Gamma}{\Gamma \vdash \mathbf{readln}(x_i^{i \in 0 \dots n})}$$

Translating to the C language

As the previous phases, a visitor has been implemented to translate the AST into a valid C program. To do so, a simple library called `toy_functions.h` has been written to handle some functions in a more easy way. It is composed of the following methods:

- `char* readln()`: read a sequence of character from `stdin` until the `\n` character and return the corresponding string;
- `int string_to_int(char* s)`: a `strtol()` alias that also deallocates the input string;
- `int string_to_float(char* s)`: a `strtof()` alias that also deallocates the input string;
- `int string_to_bool(char* s)`: convert *“true”* to 1 and *“false”* to 0 and deallocates the input string.

The reason why these functions are needed is explained here.

An important feature of this visitor is that insert a predefined *prefix* `__t2c__` at every identifier found during the AST traversal: this is needed for 2 reasons:

- Every generated names during translation does not have the *prefix*, ensuring that no conflicts could emerge with other identifiers;
- To avoid conflicts with *C names*

```
1   For example, in Toy is possible to declare a function named exit(), even if
    it is forbidden in C
```

Notably differences between Toy and C

The two languages differ in various aspects. Notably:

- Toy allows the programmer to write a function with multiple return types, while C doesn't;
- Toy boolean variables are `true` and `false`, while C handles them as 1 and !1;
- Toy doesn't require the programmer to explicitly allocate memory when declaring a new string variable;
- Toy allows the programmer to open and close a string on two different lines of code.

For each of them, the solution is explained below.

Multiple return types

This is a simple Toy function that returns three integer variables.

```
1 proc multAddDiff()int, int, int :
2   int primo, secondo, mul, add, diff;
3
4   write("Inserire il primo argomento:\n");
5   readln(primo);
6   write("Inserire il secondo argomento:\n");
7   readln(secondo);
8   mul, add, diff := primo*secondo, primo + secondo, primo - secondo;
9   -> mul, add, diff
10 corp;
```

When translating this snippet, this is what gets generated:

```
1 typedef struct function_struct_t2cmultAddDiff
2 {
3   int p_0;
4   int p_1;
5   int p_2;
6 } function_struct_t2cmultAddDiff;
```

```

7
8 function_struct_t2cmultAddDiff t2cmultAddDiff()
9 {
10     int t2cprimo, t2csecondo, t2cmul, t2cadd, t2cdiff;
11     printf("%s", "Inserire il primo argomento:\n");
12     t2cprimo = string_to_int(readln());
13     printf("%s", "Inserire il secondo argomento:\n");
14     t2csecondo = string_to_int(readln());
15     t2cmul = t2cprimo * t2csecondo;
16     t2cadd = t2cprimo + t2csecondo;
17     t2cdiff = t2cprimo - t2csecondo;
18     function_struct_t2cmultAddDiff
        function_struct_t2cmultAddDiff396874f9a95149b6be1614ee5e99fed5;
19     function_struct_t2cmultAddDiff396874f9a95149b6be1614ee5e99fed5.p_0 = t2cmul;
20     function_struct_t2cmultAddDiff396874f9a95149b6be1614ee5e99fed5.p_1 = t2cadd;
21     function_struct_t2cmultAddDiff396874f9a95149b6be1614ee5e99fed5.p_2 = t2cdiff;
22     return function_struct_t2cmultAddDiff396874f9a95149b6be1614ee5e99fed5;
23 }

```

Firstly, a `struct` with three `int` fields has been defined. It handles the return statement of the Toy function `multAddDiff` by getting returned by such function. The flow of the translated function is pretty much the same as the original, but the return statement explodes into a series of assignments.

A mandatory note about the name of the `struct` variable: a uniquely generated string has been appended to the actual function. This verbose act prevents multiple declarations of the same `c` variable, in case of subsequent calls of the same function in a given scope (recursion!!).

Lastly, the filled `struct` is returned and it's used as follows:

```

1     int __t2c__a, __t2c__b, __t2c__c;
2     function_struct__t2c__multAddDiff
        multAddDiffd3eae84310004a499bdcd38c5f9e2073 = __t2c__multAddDiff();
3     __t2c__a = multAddDiffd3eae84310004a499bdcd38c5f9e2073.p_0;
4     __t2c__b = multAddDiffd3eae84310004a499bdcd38c5f9e2073.p_1;
5     __t2c__c = multAddDiffd3eae84310004a499bdcd38c5f9e2073.p_2;

```

Memory allocation for strings

Toy doesn't require the programmer to explicitly indicate the length of a string. Of course this is a huge problem when translating to C. In order to solve this issue, the function `readln` has been defined which reads characters from `stdin` and allocate enough memory to store the string read.

An important flaw: We are aware of a huge problem in this implementation: **no string gets deallocated**.

With a single visit to the AST is impossible to determine for all strings if a certain string can be deallocated or *when* it can be deallocated.

To solve this issue, a *Garbage collector* fits perfectly but its implementation was beyond the scope of this project.

Strings spanning on multiple lines

Notably, C doesn't allow string delimiters to be on two distinct lines of code.

```

1 //This doesn't work
2 char *str ="C doesn't like
3 enjambement";

```

On the other hand, Toy has no strict rule in that regard. For example, the following string is perfectly correct:

```
1 string fullName := " First name
2 Last name";
```

The `ToyToCVisitor` translates such strings by simply replacing each `\n` sequence with `\\n`. Its corresponding C string would be:

```
1 char *fullName = "First name \\n Last name";
```

Boolean shenanigans

Toy handles boolean variables as “true” and “false”, but C does not. In order to actually print these variables, a simple trick has been used. Supposing the compiler runs into a `write` statement that contains a boolean variable, the translation of such block of code would look something like this:

```
1 int __t2c__a_boolean=1;
2 printf("%s%s", "This is a boolean: ", __t2c__a_boolean== 1 ? "true" :
   "false");
```

By using the ternary operator `?` it's easy to return to the original toy-ish boolean values on a single line.