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

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

IntelliJ Configuratiom

This project is provided with a set of *IntelliJ Configurations:*

> To run it just import it as a *Maven project* and click *Run*

Maven Build

Alternatively you can compile manually typing:

```
mvn package
```

The *jar* file will be placed under the *target/* directory.

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.

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 contaning 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.

```
    //Procedures
    PROC "proc"
    CORP "corp"
    MAIN "main"

    //Type
    INT "int"
    FLOAT "float"
    BOOL "bool"
    STRING "string"
    VOID "void"

    //Statements
    WHILE "while"
    DO "do"
    OD "od"
    READ "readln"
    WRITE "write"
    ASSIGN "assign"
    IF "if"
    THEN "then"
    FI "fi"
    ELSE "else"
    ELIF "elif"

    //Separators
    LPAR "("
    RPAR ")"
    COLON ":"
    COMMA ","
    SEMI ";"


    //Operators
    ASSIGN ":="
    PLUS "+"
    MINUS "-"
    TIMES "*"
    DIV "/"
    EQ "="
    NE "<>"
    LT "<"
    LE "<="
    GT ">"
```

```
GE ">="
AND "&&"
OR "||"
NOT "!"
NULL "null"
TRUE "true"
FALSE "false"
RETURN "->"


ALPHA=[A-Za-z]
DIGIT=[0-9]
NONZERO_DIGIT=[1-9]
NEWLINE=\\r|\\n|\\r\\n
WHITESPACE =  | [ \\t\\f]
ID = (|_)*
INT = ((*)|0)
FLOAT = +)
STRING_TEXT = [^\\"]*
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 *PEMDAS* (Parenthesis, Exponents, Multiplications/Divisions, Additions/Subtractions) rule has been introduced.

```
Program ::= VarDeclList ProcList
        ;

    VarDeclList ::= /* empty */
        | VarDecl VarDeclList
        ;

    VarDecl ::= Type IdListInit SEMI
        ;

    ProcList ::= Main
        | Proc ProcList
        ;

    Type ::= INT
            | BOOL
            | FLOAT
            | STRING
        ;

    IdListInit ::= ID
        | IdListInit COMMA ID
        | ID ASSIGN Expr
        | IdListInit COMMA ID ASSIGN Expr
        ;

    Proc ::= PROC ID LPAR ParamDeclList RPAR ResultTypeList COLON
            VarDeclList StatList RETURN ReturnExprs CORP SEMI
        | PROC ID LPAR RPAR ResultTypeList COLON
            VarDeclList StatList RETURN ReturnExprs CORP SEMI
        |   PROC ID LPAR ParamDeclList RPAR ResultTypeList COLON
            VarDeclList RETURN ReturnExprs CORP SEMI
        | PROC ID LPAR RPAR ResultTypeList COLON
            VarDeclList RETURN ReturnExprs CORP SEMI
        ;

    Main ::= PROC MAIN LPAR ParamDeclList RPAR INT COLON
            VarDeclList StatList RETURN ReturnExprs CORP SEMI
        | PROC MAIN LPAR RPAR INT COLON
            VarDeclList StatList RETURN ReturnExprs CORP SEMI
        |   PROC MAIN LPAR ParamDeclList RPAR INT COLON
            VarDeclList RETURN ReturnExprs CORP SEMI
        | PROC MAIN LPAR RPAR INT COLON
            VarDeclList RETURN ReturnExprs CORP SEMI
        ;
```

```

ResultTypeList ::= ResultType
    | ResultType COMMA ResultTypeList
    ;

ResultType ::= Type
        | VOID
    ;

ReturnExprs::=  ExprList
    | /* empty */
    ;

ParamDeclList ::= ParDecl
    | ParamDeclList SEMI ParDecl
    ;

ParDecl ::= Type IdList
    ;

IdList ::= ID
    | IdList COMMA ID
    ;

StatList ::= Stat
    | Stat StatList
    ;

Stat ::= IfStat SEMI
    | WhileStat SEMI
    | ReadlnStat SEMI
    | WriteStat SEMI
    | AssignStat SEMI
    | CallProc SEMI
    ;

WhileStat ::= WHILE StatList RETURN Expr DO StatList OD
    | WHILE Expr DO StatList OD
    ;

IfStat ::= IF Expr THEN StatList ElifList Else FI
    ;

ElifList ::= /* empty */
    | Elif ElifList
    ;

Elif ::= ELIF Expr THEN StatList
    ;

Else ::= /* empty */
        | ELSE StatList
    ;

ReadlnStat ::= READ LPAR IdList RPAR
    ;
```

```

WriteStat ::=  WRITE LPAR ExprList RPAR
    ;

AssignStat ::= IdList ASSIGN ExprList
    ;

CallProc ::= ID LPAR ExprList RPAR
    | ID LPAR RPAR
    ;

ExprList ::= Expr
    | Expr COMMA ExprList
    ;

Expr ::= NULL
    | TRUE
    | FALSE
    | INT_CONST
    | FLOAT_CONST
    | STRING_CONST
    | ID
    | ID LPAR ExprList RPAR
    | ID LPAR RPAR
    | Expr1  PLUS Expr2
    | Expr1  MINUS Expr2
    | Expr1  TIMES Expr2
    | Expr1  DIV Expr2
    | Expr1  AND Expr2
    | Expr1  OR Expr2
    | Expr1  GT Expr2
    | Expr1  GE Expr2
    | Expr1  LT Expr2
    | Expr1  LE Expr2
    | Expr1  EQ Expr2
    | Expr1  NE Expr2
    | MINUS Expr
    | NOT Expr
    ;
```

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 of the *non-terminals* appearing in the right-hand side of the production itself. There are several different actions in this parser.

For example:

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

The *empty* production creates a new list containing the variables declarated, 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.

- TypeCheck rules

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

- 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

$$\frac{\Gamma \vdash proc : \tau_i^{i \in \mathbb{N}} \to \tau_j^{j \in \mathbb{N}} \quad \Gamma \vdash par_i^{i \in \mathbb{N}} : \tau_i}{\Gamma \vdash proc(par_i^{i \in \mathbb{N}}) : \tau_j^{j \in \mathbb{N}}}$$

- While statement

$$\frac{\Gamma \vdash cnd\_expr : \boldsymbol{boolean} \quad \Gamma \vdash while\_stmt_{i}^{i \in \mathbb{N} \setminus \{0\}} \quad \Gamma \vdash do\_stmt_{j}^{j \in \mathbb{N}}}{\Gamma \vdash \boldsymbol{while}\ (while\_stmt_{i}^{i \in \mathbb{N} \setminus \{0\}})\ \boldsymbol{do}\ (do\_stmt_{j}^{j \in \mathbb{N}})\ \boldsymbol{od}}$$

- Assign statement

$$\frac{(x_{i}^{i \in \mathbb{N} \setminus \{0\}} : \tau_{i}) \in \Gamma \quad \Gamma \vdash (expr_{j}^{j \in \mathbb{N} \setminus \{0\}}) \to \tau_{i}^{i \in \mathbb{N} \setminus \{0\}}}{\Gamma \vdash x_{i}^{i \in \mathbb{N} \setminus \{0\}} = expr_{j}^{j \in \mathbb{N} \setminus \{0\}}}$$

- If statement

$$\frac{\Gamma \vdash cnd\_expr : \boldsymbol{boolean} \quad \Gamma \vdash then\_stmt_{x}^{x \in \mathbb{N}} \quad \Gamma \vdash elif\_stmt_{y}^{y \in \mathbb{N}} \quad \Gamma \vdash else\_stmt_{z}^{z \in \mathbb{N}}}{\Gamma \vdash \boldsymbol{if}\ cnd\_expr\ \boldsymbol{then}\ (then\_stmt_{x}^{x \in \mathbb{N}})\ \boldsymbol{elif}\ (elif\_smt_{y}^{y \in \mathbb{N}})\ \boldsymbol{else}\ (else\_stmt_{z}^{z \in \mathbb{N}})\ \boldsymbol{fi}}$$

Translating to the C language

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.

Multiple return types

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

```
proc multAddDiff()int, int, int :
    int primo, secondo, mul, add, diff;

    write("Inserire il primo argomento:\n");
    readln(primo);
    write("Inserire il secondo argomento:\n");
    readln(secondo);
    mul, add, diff := primo*secondo, primo + secondo, primo - secondo;
    -> mul, add, diff
corp;
```

When translating this snippet, this is what gets generated:

```c
typedef struct function_struct_t2cmultAddDiff
{
    int p_0;
    int p_1;
    int p_2;
} function_struct_t2cmultAddDiff;

function_struct_t2cmultAddDiff t2cmultAddDiff()
{
    int t2cprimo, t2csecondo, t2cmul, t2cadd, t2cdiff;
    printf("%s", "Inserire il primo argomento:\n");
    t2cprimo = string_to_int(readln());
    printf("%s", "Inserire il secondo argomento:\n");
    t2csecondo = string_to_int(readln());
    t2cmul = t2cprimo * t2csecondo;
    t2cadd = t2cprimo + t2csecondo;
    t2cdiff = t2cprimo - t2csecondo;
    function_struct_t2cmultAddDiff
        function_struct_t2cmultAddDiff396874f9a95149b6be1614ee5e99fed5;
    function_struct_t2cmultAddDiff396874f9a95149b6be1614ee5e99fed5.p_0 = t2cmul;
    function_struct_t2cmultAddDiff396874f9a95149b6be1614ee5e99fed5.p_1 = t2cadd;
    function_struct_t2cmultAddDiff396874f9a95149b6be1614ee5e99fed5.p_2 = t2cdiff;
    return function_struct_t2cmultAddDiff396874f9a95149b6be1614ee5e99fed5;
}
```

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:

```
    int __t2c__a, __t2c__b, __t2c__c;
    function_struct___t2c__multAddDiff
        multAddDiffd3eae84310004a499bdcd38c5f9e2073 = __t2c__multAddDiff();
    __t2c__a = multAddDiffd3eae84310004a499bdcd38c5f9e2073.p_0;
    __t2c__b = multAddDiffd3eae84310004a499bdcd38c5f9e2073.p_1;
    __t2c__c = multAddDiffd3eae84310004a499bdcd38c5f9e2073.p_2;
```

Memory allocation for strings

Toy doesn't requier 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, a tiny C library was defined. It defines the funcion `readln`, named after the Toy function, 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**. The ones that shouldn't are function parameters and those that get returned to the calling function. Although it seems easy to discriminate between these, there are cases in which this operation is impracticable with a single visit of the Syntactic Tree (i.e. nesting callProcedure statements in a return statement).

Strings spanning on multiple lines

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

```
//This doesn't work
char *str ="C doesn't like
enjambment";
```

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

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

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

```
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:

```
    int __t2c__a_boolean=1;
    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.