Toy language Compiler

A simple Parser 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.

The patterns

//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.

The grammar

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.

Translating to the C language