LW2 | LW3

DA53: Compilers and Language Theory

stephane.galland@utbm.fr

Lexical Analysis and Syntax Analysis

00 | Goal

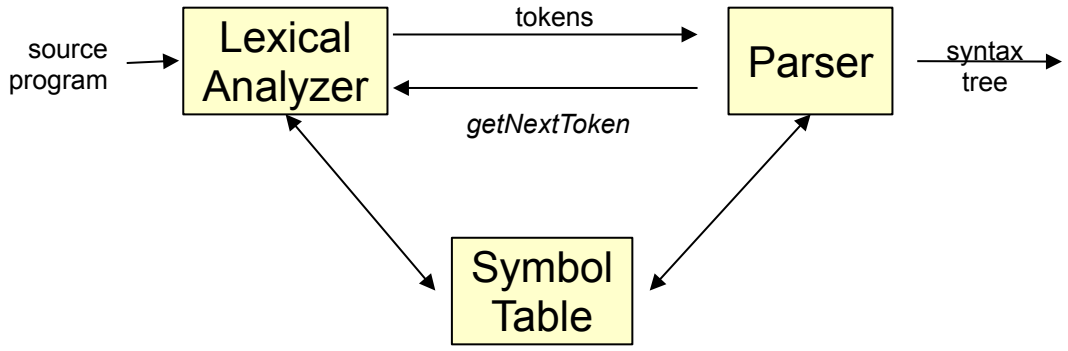
The goal of this on-computer tutorial session is to write a small interpreter for the Tiny Basic language.

The development must be done in Java.

JavaCC (https://javacc.github.io/javacc/) should be used. This library is also available on Teams.

You must download and complete a skeleton of Java project, downloadable from Teams.

Remember that the lexical analyzer takes a source program, and scans it, and replies the recognized tokens when they are requested by the parser. The parser generates a syntax tree that can be executed (interpretation mode).



During this on-computer tutorial, you will :

1. Define the grammar and the Syntax-Directed Definition (SDD) of the Tiny Basic language.
2. Write a symbol table
3. Write the lexer
4. Write the interpretation rules

01 | Language Definition

The language to implement in this labwork is a part of the Tiny Basic dialect of the BASIC language.

01.1 Statements

The language is composed of statements, one per line. If this labwork we do not recognize multiple statements on the same line. A line must follow one of the following formats :

- statement
- number statement

where number is the number of the line, and statement is the instruction of the language.

The supported statements are :

- **PRINT** expression
  - Print the given expression on the standard output
- **IF** expression relational_operator expression **THEN** statement
  - The statement is executed only if the given relational operator on the two given expressions replies true.
  - The relational operators are :
    - = equality
    - < less than
    - > greater than
    - <= less or equal
    - >= greater or equal
    - <>, >< not equal
- **GOTO** expression
  - Run the statement at the line defined by the expression.
- **INPUT**variable [**,** variable ...]
  - Read variable values from the standard input .
- **LET** variable **=** expression
  - Assign the value of the expression to the variable.
- **GOSUB** expression

    - Run the statement at the line defined by the expression, but in opposite than GOTO, remember the line of the GOSUB and run the statement just after that line when the statement RETURN is encountered.

- **RETURN**
    - Run the statement at the line following the last GOSUB.
- **END**
    - Stop the program.
- **REM** text
    - Insert a comment in the source code.

01.2 Expressions

Many of these statement types involve the use of expressions. An expression is the combination of one or more *numbers*, *strings* or *variables*, joined by *operators*, and possibly grouped by *parentheses*.

There are four operators: +, -, *, /

Note that the operators + and – have two versions : the binary (a+b) and the unary (+a).

A string of characters is enclosed by the quote characters (").

01.3 Variable Definition and Scope

The variables do not need to be declared. The scope of all the variables is global. So that a variable, when it is used, *could be undefined*.

All the variables are of numerical type or of string type, depending on the type of the expression assigned to the variable.

01.4 Example

Let the following Tiny Basic program :

```
5 LET S = 0
10 INPUT NUM
15 INPUT V
20 LET N = NUM
30 IF N <= 0 THEN GOTO 99
40 LET S = S + V
50 LET N = N - 1
70 GOTO 30
99 PRINT S/NUM
100 END
```

02 | JavaCC

JavaCC is a Java library and a collection of CLI tools that is generating a lexer and a parser written in Java.

02.1 Input file

The input file of JavaCC is a file with extension « .jj », named here «tinybasic.jj ».

The input file must start with the Java definition of the Parser (named TinyBasicParser). You may put in this part all the functions and attributes that are invoked in the grammar rules.

After the Java definition, you must put the lexer rules and the grammar (BNF) rules.

```
options {
  IGNORE_CASE = true;
  STATIC = false;
}

PARSER_BEGIN(TinyBasicParser)

package fr.utbm.gi.lo46.tp2.parser;

import java.util.SortedMap;
import java.util.Map;
import java.util.List;
import java.util.TreeMap;
import java.util.ArrayList;

import fr.utbm.gi.lo46.tp2.context.*;
import fr.utbm.gi.lo46.tp2.symbol.*;
import fr.utbm.gi.lo46.tp2.syntaxtree.*;

public class TinyBasicParser {

     private int basicLineNumber = 1;
     private final SymbolTable symbolTable = new SymbolTable();

     /** Replies the symbol table used by the parser.
      * @return the symbol table.
      */
     public SymbolTable getSymbolTable() {
           return this.symbolTable;
     }

}

PARSER_END(TinyBasicParser)
```

02.2 Lexer rule syntax

- Each lexer rule defines the lexeme that is recognized by the lexer. The recognized lexemes are named tokens.
- Different types of lexer rules are available : SKIP, TOKEN, …

- The best practice in JavaCC is to define tokens only when they are associated to complex regular expressions, ie. then an regular expression operator is used. For example, the lexeme « <= » may not be defined in a token rule. Rather is should be directed typed in the BNF rules.

- The SKIP rule tells to the lexer to ignore several sequences of characters :

```
SKIP :
{
      " "
   |  "\t"
   |  "\r"
}
```

- The TOKEN rule tells to the lexer the token to reply to the syntax analyzer :

```
TOKEN :
{
      < CR: "\n" >
}
```

02.3 Parser rule syntax

- The grammar (BNF) rules defines the syntactic rules of the language.

- Each rule is defined like a kind of Java function (with a similar but different syntax) :

```
void ruleName() :
{
}
{
}
```

- The prototype of the rule has the same syntax as the prototype of a Java function. Return type and parameters are possible. Indeed, a rule could return a value or take values as parameters when it is invoked.

- The first block after the column character must contains any declaration of Java variable.

- The second block after the column character must contains the BNF rules (separated by the « | » character).

- The BNF rule is a sequence of one of :

    1. a token
        - the value of the token could be retreived with an assign symbol (see example below).
    2. a BNF rule invokation
        - it is a call to one of the BNF-rule functions, using the Java syntax.
        - the value returned by the BNF-rule could be retreived with an assign symbol (see example below).
        - Parameters could be passed to the BNF-rules function.
    3. a Java block

The following example is rule that is recognizing a parenthezed numerical expression, a number literal, or an identifier.

```
private Number factor() :
{
   Number value;
   Token t;
}
{
     "(" value = expression() ")"
    { return value; }
   | t = <NUMBER>
    { return NumberUtil.parse(t.image); }
   | t = <IDENTIFIER>
    { this.symbolTable.add(t.image,t.beginLine);
       return this.symbolTable.get(t.image);
     }
}
```

02.4 Run JavaCC

- On the CLI, type :

  $> cd src/fr/utm/gi/lo46/tp2/parser

  $> javacc tinybasic.jj

02 | Working Steps

02.1 Skeleton

- Download the skeleton, and install it into your Eclipse.
- The skeleton contains : a symbol table (SymbolTable), a context of interpretation (ExecutionContext), an interpreter (Interpreter, and SyntaxTreeInterpreter), the main function (TinyBasicInterpreter), and several abstract classes that correspond to the tree nodes of the syntax tree.
- Takes a look on the code.

02.2 JavaCC Scanner and Lexer Definition

- Write the JavaCC input file that is corresponding to the Lexer (the token).

02.3 Complete the JavaCC Definition

- Write the grammar inside your JavaCC input file (the BNF rules with the Java blocks empty).

02.4 Parse Tree Node

- Create classes for all the nodes that are required in a parse tree dedicated to the Tiny Basic language.
- Each node of the parse tree must corresponds to an element of the language.
- Each statement node should be linked to child nodes that are corresponding to the parameters of the statement.

02.5 Write the SDD in JavaCC

- Update your JavaCC input file with the SDD rules that permit to :
  - check the types
  - generate the parse tree for the Tiny Basic program.

02.6 Interpretation of the Parse Tree

- Add in each node of your parse tree, the function **run(...)** and **evaluate(...)** which may be invoked to run the parse tree as in an interpreter, and to compute the value of an expression.

02.7 Extension of the Language

- Update your compiler implementation to support the new statements :
  - Statement : **for id = num to num step num statements next id**
  - Arrays as **id (expression)**
  - The new keyword that is replying the size of an array : **LEN (id)**
  - Statement : **while (expression relop expression) do statements wend**