UNIVERSITÉ LIBRE DE BRUXELLES

DÉPARTMENT D'INFORMATIQUE

INFO-F403 - INTRODUCTION TO LANGUAGE THEORY AND COMPILING

Project Report – Part 1

Author:
Hakim BOULAHYA

Professor:
Gilles GEERAERTS

November 23, 2017

Contents

1 Extended Regular Expressions (ERE)

This section describes the extended regular expressions used by the lexer. The macros section describes simple ERE that are used to write the advanced ERE.

1.1 Macros

1. `AlphaUpperCase = [A-Z]` : Match all uppercase letters
2. `AlphaLowerCase = [a-z]` : Match all lowercase letters
3. `Alpha = {AlphaUpperCase}|{AlphaLowerCase}` : Match alphabet case-sensitive characters
4. `Numeric = [0-9]` : Match digit characters
5. `AlphaNumeric = {Alpha}|{Numeric}` : Match alphabetic **OR** digit characters

1.2 Macros - Lexical Units

1. `Number = ([1-9]Numeric*)|0` : Match all numbers that **MUST** start with a non-zero digit and follow by 0 or more digits (first RE), **OR** match only zero digit (second RE). This RE correspond to the [Number] lexical unit.
2. `VarName = AlphaAlphaNumeric*` Match identifiers. Identifiers **MUST** starts with a alphabetic character and **CAN** be followed by 0 or more digits **OR** alphabetic characters. Correspond to the [VarName] lexical unit.
3. `Blank = \s+` Match 1 or more *blank* character: spaces, new lines and/or tabulations.

1.3 Keywords

All the following keywords are matching *themself* directly *i.e.* return the corresponding symbol with the corresponding lexical unit (see section 2.2):

;	:=	(	)	+
-	*	/	if	then
endif	else	while	for	from
by	to	do	done	note
and	or	=	>=	>
<=	<	<>	print	read
begin	end			

2 Implementation

2.1 Code structure

Files They are two source files for the compilator: `LexicalAnalyzer.flex`, contains the lexer ERE that analyse the input sources and `ImpCompilo.java` that contains the Java source code, used

in the Regular Expressions matching actions. For details about each method of `ImpCompilo.java`, read the JavaDoc at **doc/javadoc/ImpCompilo.html**.

Code flexibility To allow a better code flexibility, the JFlex generated class will `extends ImpCompilo` class. This allow to develop the compilator directly from a Java source class, instead of putting the ERE and the logical code inside the .flex file. The generated class is named `Main.java`, which is used to run the scanner.

2.2 Keywords

One state is enough to return the IMP language tokens. The state YYINITIAL (the default inclusive state) is used to match all the keywords listed in section 1.3. For each keyword the action is running the method `ImpCompilo.symbol(LexicalUnit)`. This method runs the following instruction:

1. if LexicalUnit is VarName: add it to `identifiers` dictionnary
2. Add the symbol to list `symbols`
3. Output the symbol using the provided `toString` method.

The instruction that add the symbol to the list `symbols` will be relevant for the part 2 of the project.

Example For the keyword `if`, the corresping ERE line in YYINITIAL is:

```
"if"            {return symbol(LexicalUnit.IF);}
```

[VarName] and **[Number]** lexical units are also handled in YYINITIAL. The only difference is that the right-side of the line code is not a keyword but the correspding ERE defined in section 1.2.

2.3 Blank characters

Blank characters (space, new line and tab character) are ignored by the scanner. It is done by matching space *i.e.* the ERE `\s+`, which matches 1 or more blank character, to an empty action. We have in YYINITIAL:

```
{Blank}         {}
```

2.4 Unknown token

If some characters don't match any of the ERE in YYINITIAL, the scanner will output:

```
Unknown token: '<text>'
```

Note that this doesn't happen in the COMMENT state (see section 2.5), since everything is ignored.

2.5 Comments

Comments can only be represented by an enclosed text: `(* Comments content *)`.
Therefore the solution provided to ignore comments is the following:

- In YYINITIAL when a comment is opened, i.e. matches `(*`, change state to COMMENT
- In COMMENT, ignore all characters (including blank characters) except the ending comment characters `*)`. If these characters are matched change state to YYINITIAL.

This process will ignore all characters enclosed in comments.

2.5.1 Forbidden behaviours

There are two *forbidden* behaviours of comment detection that can be highlighted:

1. A comment is opened but not closed.
2. A closed comment is detected, but no opening precede it.

The behaviour number 1 will ignore every characters until the EOF, since the state will change to COMMENT, and never revert to YYINITIAL.

The behaviour number 2 is in state YYINITIAL, which means that the *closed* comment characters `*)` will match the TIMES and LPAREN `LexicalUnit`, because the closed comment characters are only defined in COMMENT state.

With our implementation the nested comments, that are *forbidden*, should raise a syntax error, when detected in the parser. Because with `(* (* *) *)`, the second comment opening will be ignored in the COMMENT state, the first closed comment characters will close the first comment opening. The last closed comment characters is left as if we were in behaviour number 2. Note that this behaviour is the one implemented in the Java language.

2.5.2 Nested comments (Bonus)

If we need to handle nested comments, an easy implementation is to use a counter that will count how many comments are opened, therefore how many comments still need to be closed. The process is the following:

1. in YYINITIAL, if comment open: init counter to 1 and change state to COMMENT.
2. in COMMENT, if comment open: increment the counter.
3. in COMMENT, if comment close: decrement the counter, and if the counter is equal to 0, change state to YYINITIAL.

Here is an implementation of the proposed solution (in pseudocode):

```
// Lexer
<YYINITIAL> {
    ...
    "(*"    {initCommentState();}
    ...
}

<COMMENT> {
    "(*"    {openComment();}
    "*)"    {closeComment();}
    .|{Blank}   {} // Ignore everything else
}

// Code

initCommentState(){
    commentCounter = 1;
    changeState(COMMENT);
}

openComment(){
    commentCounter++;
}

closeComment(){
    commentCounter--;
    if (commentCounter == 0) // If all closed, scan
        changeState(YYINITIAL);
}
```

For the example given in section 2.5.1, if we use the solution proposed above, no error will be detected, and the nested comments will be ignored.

3 How To

3.1 Files

The executable jar is **dist/imp-compilo.jar**. The source files are in **more/src**. It contains the following files:

- **LexicalUnit.java**: provided class.
- **Symbol.java**: provided class.
- **ImpCompilo.java**: the class that the lexer extends, containing the source code used in the lexer.
- **LexicalAnalyzer.jflex**: the lexical analyzer that generates the **Main.java** class.

All the imp source files, used to test the scanner, are in **test** directory.

3.2 Run and JavaDoc

Run Execute the following command to run the scanner:

```
java -jar dist/imp-compilo.jar <input>
```

Doc The JavaDoc is available at **doc/javadoc/ImpCompilo.html**.