Introduction to language theory and compiling
Project - Part 1

Université Libre de Bruxelles

Hakim Boulahya

October 28, 2017

Contents

1 Definition

1.1 Emphasizing

- MUST
- OR
- CAN

1.2 Notation and defintions

- case-senstive characters
- alpha numeric
- ERE: Extended Regular Expression

2 Extended Regular Expressions

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

2.1 Shortcuts

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

2.2 Advanced

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+`

2.3 Keyword

All the following keyword are matching *themself* directly:

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

3 Implementation

3.1 Code structure

Files Our implementation is contained in two files: `LexicalAnalyzer.flex`, contains the ERE that analyse the input sources and `ImpCompilo.java` that contains the Java source code, used in the regular Expressions matching actions.

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 should be used to run the scanner.

3.2 Keywords

This section describe the states**?** implemented in our lexical analyzer and the corresponding ERE used.

One state is enough (necessary) to return the IMP language tokens. The state **YYINITIAL** (the default inclusive state) is used to match all the keywords listed in section 2.3. 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 `symbols`
3. Output the symbol using the provided `toString` method.

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 2.2.

3.3 Blank characters

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

```
{Blank}          {}
```

3.4 Unknown token

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

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

Note that this does not append in the COMMENT state (section 3.5), since everything is ignored.

3.5 Comments

Comments can only be represented by an enclosed text: `(* Comments content *)`.

Therefore the solution provide to ignore the comments is the following:

- In YYINITIAL and 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 in enclosed in comments.

3.5.1 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.

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, will 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.

3.5.2 Nested comments (Bonus)

Use counter ?

4 All in yyinitial vs states for different grammars

Changes between states are diffuclt to detect Unknown tokens it increase diffulcty of impl better to use one state always makes no sense to use multi state since the grammar is checked during the scanner, therefore the grammar should be impl in part 2.