Introduction to language theory and compiling
Project - Part 1

Antoine Passemiers
Alexis Reynouard

October 8, 2017

Contents

1. Assumptions

This part of the project consists in designing and implementing a lexical analyzer for the Imp programming language, from a given list of reserved keywords and a grammar for the language.

- As suggested in the project brief, numbers are only made up of sequences of digits, thus can only be integers. Decimal numbers are not supposed to be recognized by the lexer. Negative integers are handled using the rule 16 from the Imp grammar: a negative number is composed of the unary operator "-" followed by a positive number. Here, the lexer must only be capable of tokenizing both the unary operator "-" and positive numbers, and thus not able to infer the existence of a negative number. This will be implemented in the next parts of the project.
- Varnames are sequences of letters and digits, and necessarily start with a letter. Also they are case sensitive. Therefore, they can not contain underscores.
- Since no lexical units have been provided for booleans nor for strings, the lexer does not handle them. In any case, booleans could be regarded as simple integers, and possibly constants could be defined in the Imp environment to support them.
  For example: $TRUE := 1; FALSE := 0$
- We did not relax the assumption on token delimiters: two tokens may or may not be separated by a space/tabulation/new line. This allows tokens to be contiguous, for instance:
  $FORTYTWO := 42; C = FORTYTWO$

  The lexical units "42" and ";" are detected as two distinct tokens. However, this behavior can induce ambiguous situations such as $2B$, which is the number 2 followed by an identifier B. This will most likely be syntactically be not appropriate, but that is not the lexer's job to tell whether it is syntactically correct or not (this is one of the parser's jobs). As a result, the lexer does not raise any exception when being in this kind of situations.
- Non-ascii characters are not allowed, except in comment sections.
- Because the provided .out files do not contain EOS tokens, we decided not to insert them yet.

2. Regular expressions

2.1 Important regular expressions

Name	**Unix-style notation**	**Description**
CommentBegin	\(*	Start symbol before a comment
CommentContent	(*[^\)]|[^*])**\)	Well-formed comment: Match any charactered until first "*)". Nested comments are forbidden.
Space	[\n\r\t]*	Token delimiter: indentation, space or new line
AlphaUpperCase	[A-Z]	A single uppercase letter
AlphaLowerCase	[a-z]	A single lowercase letter
Alpha	{AlphaUpperCase}|{AlphaLowerCase}	A single alphabetical character
Digit	[0-9]	A single digit
AlphaNumeric	{Alpha}|{Digit}	A digit or an alphabetical character
Number	{Digit}+	A sequence of digits
Identifier	{Alpha}{AlphaNumeric}*	Sequence of characters that does not start with a digit

2.2 Trivial regular expressions

begin, end, ;, :=, (,), −, +, ∗, /, if, then, endif, else, not, and, or, =, >=, >, <=, <, <>, while, do , done, for, from, by, to, print, read

3. Implementation

3.1 Lexical analyzer

3.1.1 Lexical unit matching

Besides the JFlex-default state $YYINITIAL$, we defined a new state $COMMENT$ which is active when the opening comment symbol ($CommentBegin$) is encountered. The system goes back to $YYINITIAL$ state once a comment content is matched, according to the regex $CommentContent$. All the other regular expressions can be matched only in the $YYINITIAL$ state, because comment sections can also contain reserved keywords, but the latter must not be considered as such inside comments.

The main problem is that *reserved keywords* can be matched with the varname regular expression as well. As a result, more than one token may be recognized at once, and JFlex uses two rules of preference to decide which one to return. JFlex selects first the longest match, and then chooses the first matched string among the longest matches. Thus, a common solution to keep the keywords ”reserved” is to give priority to the keyword regular expressions (all but varname) by putting them ahead in the source code. The identifier regular expression has been added at the end.

Except for comment contents, each new matched regular expression induces the instanciation of a new *Symbol* object. The matched string is stored as the *value* attribute, the *line* attribute is set to the current line number and the *column* attribute is set to the current column number. The *line* attribute is used by the symbol table for handling varnames (this will be explained in the next section).

We defined two custom exceptions, *BadTerminalException* and *BadTerminalContextException*. The first one occurs when the current string $yytext$ doesn't match any of our defined regular expressions. In other words, a *BadTerminalException* is thrown when the regular expression [ˆ] is matched. In addition, a *BadTerminalContextException* is thrown when a closing comment symbol is matched inside a non-comment section. Because the lexer is not supposed to throw exceptions, we implemented a Main Java class that wraps the execution of the JFlex-generated class (called *LexicalAnalyzer.java*) and catches all the potential exceptions. When an exception is catched, a dedicated message is simply displayed in the standard output.

3.1.2 Symbol table management

We used a *LinkedHashMap* from the java standard library to store the different symbols. We chose this kind of data structure rather than a simple *HashMap* because it is more convenient to keep track of the order of inserted symbols. Indeed, symbols are supposed to be displayed in order of appearance.

Each time a lexical unit is matched, its value is hashed and compared to the indexes present in the linked hashmap. The unit is added to the linked hashmap if and only if its hash value is not already present. At the end of the program, the symbol table is shown by simply iterating over the hashmap's keyset. The first column of the table contains tokens (unit names) and the second column contains, for each token, the number of the line where the token has been encountered for the first time during the program execution. This number is obtained by evaluating the variable *yyline* at the moment when the current token is matched.

3.2 Testing

Eight test .imp files have been written, as well as eight corresponding .out files, to test all the features of our lexer. In addition, a bash script has been written to automatically apply the following procedure:

- Generate the Java lexer class using JFlex
- Compile the latter using the javac command
- Generate the javadoc
- Generate an executable jar from the .class files
- Execute the lexer on the example .imp file and display the result
- For each of the test.imp files, run the lexer on the .imp and compare the output file with the corresponding predefined .out file. The comparison is done with vimdiff and everything must match exactly.

Here are short descriptions of our eight pairs of test files:

1. Example files provided with the project brief. Checks that the given the .imp file, the lexer generates a .out that is identical to the provided one.
2. Same .imp file with an additional comment. The output file should remain the same.
3. Empty code: a *begin* token followed by a *end* token. The symbol table should be empty.
4. Small code with a non-ascii comment. The lexer should **not** throw a *BadTerminalException*.
5. Comments, plus a single ”*)” symbol. As explained it should lead to a *BadTerminalContextException*.
6. Tokens without separators between them. As explained before, ”2a” should be tokenized as ”2” followed by ”a”.
7. Similar to the example Imp code but using negative numbers.
8. Checks that non-ascii characters are actually not supported by our lexer. A *BadTerminalException* message is expected in the output file.

4. Nested comments - Bonus

Lexing should be done with regular expressions, but nested comments can not be processed by regular expressions, because they don't have a context-free grammar. More specifically, it is impossible to know at compile time the number of states required to handle nested comments. For any fixed number of states, one will always be able to provide a comment with a level of nesting such that it cannot be recognized by the lexer. This is due to the *pumping lemma for regular languages*. More specifically, when the closing comment symbol "*)" is matched, an automaton would not be able to remember the number of "(*" symbols already matched. For any number of matched opening comment symbols the current state would be the same: this is the intuition behind the *pigeonhole principle*. As a result, we need to keep track of this number.

Here we propose two solutions: implement a recursive parser (very complicated to compile, plus the first part of the project consists only in the implementation of a lexer), or use counters (Java can do that because it is Turing-complete).

- The first solution we propose is to use a stack and call a lex scanner recursively. Nested comments will still be recognized since they consist in both a left recursive grammar and a right recursive grammar. The problem is that the stack size must be known at compile time, which makes the problem similar to the one described earlier. Once again, we are not allowed to do so.
- The second solution is more general and also requires a memory space that is theoretically infinite, since we are dealing with Turing machines. The procedure is described as follows:
  - Initialize a counter c to 0
  - Check the current symbol. If it's a "(*", increment c. If it's a "*)", decrement c. Do nothing in other cases.
  - Reject if c becomes strictly negative.
  - Parse the next symbol.

JFlex already offers a fancy way to integrate custom Java methods in the automaton states. Our preference would be to benefit from it and manipulate counters in the Java part of the code. This would be easier to design and to maintain.

Note: We are aware that it is preferable (when possible) to use automaton-based implementations rather than calling Java code everywhere, for efficiency purposes. Indeed JFlex is designed to optimize automata through minimization.