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 an arithmetic expression 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 by itself. This will be implemented in the next parts of the project.
- Variable identifiers (varnames) are sequences of letters and digits, and necessarily start with a letter. Therefore, they can not contain underscores. Also they are case sensitive.
- 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 blank character. This allows tokens to be contiguous, *i.e.* two tokens need to be separated by a blank character only if the first one ends with a letter and the last one starts with a letter.
  For instance, one can write:
  ```
  begin read(a);read(b);while b<>0do c:=b;
  while a>=b do a:=a-b done;b:=a;a:=c done;print(a) end
  ```
- 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 Notable regular expressions

Name	Regular expression	Description
CommentBegin	`\(\*`	Opening comment symbol
CommentContent	`(\*[^\)]|[^*])*\*\)`	Well-formed comment: Match any sequence of characteres up to and including the first "`*)`". (Nested comments are forbidden.)
Space	`[ \n\r\t\f]*`	Blank spaces: only space, new line, vertical/horizontal tabulations are taken into account
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`, `\*\)`.

The last regular expression match an "end of comment" token. This should never happen as this token is already considered as part of the comment. We add this one to raise an exception when this occurs outside any comment. Another option would be to consider two lexical units: `TIMES` and `RPAREN`. Since this sequence does not make sense according to the Imp grammar, we choose to already raise an exception in order to detected nested comment more quickly.

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 after the others.

Except for comment contents, each new matched regular expression induces the instantiation 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 `[^]` (whose match anything but has the lowest priority) 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 instead of any other 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

Nine test .imp files have been written, as well as nine 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
- 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 comparisons stop when one output does not correspond to the expected output and the differences are displayed.

  The amount and type of blank characters is ignored when comparing outputs, but not the presence or absence of a space.

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 and without the character of end of line at the end of the file. The output file should remain the same.
3. Example .imp file provided with the project, but without any unnecessary blanks. The output file should remain the same.
4. Empty code: a *begin* token followed by a *end* token. The symbol table should be empty.
5. Small code with a non-ASCII comment. The lexer should **not** throw a *BadTerminalException*.
6. Comments, plus a single "*)" symbol. As explained it should lead to a *BadTerminalContextException*.
7. Tokens without separators between them. As explained before, "2a" should be tokenized as "2" followed by "a".
8. Similar to the example Imp code but using negative numbers.
9. 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

The role of the lexer is to split the whole .imp file into a list of tokens. For simplicity, we can say that this is done by matching each part of the file against regular expressions. This is to say that, for every file part, we are asking if it is a word that belong to a regular language. The regular language it belongs to give us the kind of token it is.

We want comments to be deleted by the lexer so that they are not visible to the parser. Thus, comments are words of a regular language that can be described informally this way: they start with an "`(*`", contain any character and end with *the first* "`*)`". This is the rule we used in the implementation. We need to "stop" the comment at the first "`*)`" to avoid to consider as part of the comment all the characters between the first "`(*`" and the last "`*)`".

Lexing should be done with regular expressions, but nested comments can not be processed by regular expressions, because they do not belong to regular languages. 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*. Another way to say this is: when the closing comment symbol "`*)`" is matched, a **finite** 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.

We can think of two solutions: implement a recursive lexer (very complicated to compile), or use counters (Java can do that because it is Turing-complete). In fact these two solutions are two different point of views on the way to keep track of the "comment level" the lexer is in. This is because the nested comments belongs to the class of context-free languages (recognizable by pushdown automata).

- 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 second solution may be described by a procedure 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 for efficiency purposes (when possible) to use automaton-based implementations rather than calling Java code everywhere. Indeed JFlex is designed to optimize automata through minimization.