Introduction to language theory and compiling
Project - Part 1

Antoine Passemiers
Alexis Reynouard

October 8, 2017

Contents

1 Regular expressions **1**

1.1 Important regular expressions 1

1.2 Trivial regular expressions 1

2 Implementation **2**

2.1 Lexical unit matching 2

2.2 Symbol table management 2

3 Nested comments - Bonus **3**

1. Regular expressions

1.1 Important regular expressions

CommentBegin	\(*	Start symbol before a comment
CommentContent	(*[ˆ\)]|[ˆ*])**\)	Welll-formed comment, containing neither a ”*” nor a ”)”. Nested comments are forbidden.
AlphaUpperCase	[A-Z]	A single uppercase alphabetical character
AlphaLowerCase	[a-z]	A single lowercase alphabetical character
Alpha	{AlphaUpperCase}|{AlphaLowerCase}	A single alphabetical character
Digit	[0-9]	A single digit
AlphaNumeric	{Alpha}|{Numeric}	A digit or an alphabetical character
Sign	[+-]	Either the symbol ”+” or ”-”
Integer	{Sign}?(([1-9][0-9]*)|0)	Sequence of digits that cannot begin with a ”0”
Decimal	\.[0-9]*	Decimal in scientific notation, with only digits after the decimal point
Exponent	[eE]{Integer}	Exponent in scientific notation
Real	{Integer}{Decimal}?{Exponent}?	Real number in scientific notation
Identifier	{Alpha}{AlphaNumeric}*	Sequence of characters that does not start with a digit

1.2 Trivial regular expressions

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

2. Implementation

2.1 Lexical unit matching

TODO: On a deux etats, a permet de faire les commentaires, dans l'etat initial, on parse avec toutes les regex ecrites au dessus. Et blabla...

TODO: isAtEOF ? Pas oublier d'enlever les deux TODO dans le fichier flex

2.2 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. 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*.

Here we propose two solutions: implement a recursive parser (very complicated to compile), or use counters (Java can do that because it is Turing-complete).

- The first solution we propose is to use a stack by calling the 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.
- 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.

TODO: Rajouter ce que tu as dire et s'assurer que je ne dis pas que des conneries