INFO-F403
INTRODUCTION TO LANGUAGE THEORY AND COMPILING

PROYECT PART 1: LEXICAL ANALYZER

Niklaus Geisser
Joan Gerard

Table of Contents

1 States

The lexical analyzer has 3 states, besides YYINITAL, defined as follows:

1.1. Begin Program State

This state, defined as “BEGINPROGRAMSTATE”, handles the strings after of what is matched in “BEGINPROG” regular expression. The following rules are declared:

- `{ProgramName}` Used to add the PROGNAME Lexical Unit.
- `{NotProgramName}` Used to show errors for invalid program names.

Any other scanned character will make the state to jump back into the YYINITAL state.
This state was defined in order to separate concerns and avoid the scanner to mix the program name up with variable names.

1.2. Long Comment State

This state, defined as “LONGCOMMENTSTATE”, handles the multiline comments (/**/). Since all comments must be ignored the tokenizer process will be deactivated once it reads an open-comment string and it will be reactivated once it reads a close-comment string.
The following rules will be declared:

- `{LongCommentEnd}` Jumps to YYINITIAL state once the comment ends.
- `{Endline}` Do nothing.

1.3. Short Comment State

This state, defined as “SHORTCOMMENTSTATE”, handles in-line comments (//). Same as long comment state, the tokenizer process will be deactivated once it reads a double slash and it will be reactivated once it reads an end of line. The following rules will be declared:

- `{EndLine}` Jumps to YYINITIAL state once the comment ends.

2. Regular expressions (RE)

In order to generate the tokens some regular expressions are used throughout the process, in this section the most important ones will be mentioned and explained.

The lexical analyzer uses RE to match some strings that need to be tokenized and some strings that do not match the language rules. For instance:

- *VarName*: matches variable names.

- *NotVarName*: matches everything that is not a varname.
- *Number*: matches a number.
- *NotNumber*: matches invalid numbers.
- *ProgamName*: matches a valid program name.
- *NotProgramName*: matches invalid program names.

If *NotVarName* rule did not exist, the lexical analyzer would match all the valid subsets of characters with *VarName* rule.

For instance, let's suppose that there exists a string `VARIABLES not_a_Variable` somewhere in the file and the NotVarName rule does not exist, the lexical analyzer will tokenize:

```
token: VARIABLES      lexical unit: VARIABLES
token: not       lexical unit: VARNAME
token: a         lexical unit: VARNAME
token: Variable  lexical unit: VARNAME
```

However, adding this rule to the Lexical Analyzer will tokenize it as:

```
token: VARIABLES      lexical unit: VARIABLES
```

Note that the string `not_a_Variable` matched with the `NotVarName` rule and was not printed nor added to the token list. This decision was taken because at this point we know that something is already wrong so in this case the compiling process will not continue and the user can be notified immediately with an error message[1]. If this rule was not created, the error would be triggered in the static check process[2] anyway. Same for *NotNumber* and *NotProgramName* rules.

2.1. Variable name (`{VarName}`)

A variable name is a string that is constrained by the following rules:

1. It starts with a lowercase letter.
2. The lowercase letter can be followed by other lowercase letters or digits.

The RE that matches these rules:

```
[a-z][a-z0-9]*
```

The green part matches first rule. The blue part matches the second rule.

[1] See Error Message section.
[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi Jeffrey D. Ullman. 1986. Compilers Principles, Techniques and Tools. Pearson, Boston. pp. 97

Example:

Matched string	**Not matched string**
validvar counter2nd counteruntil100	Variablei variableName var_name

2.2. Not a variable name (`{NotVarName}`)

A string is not a variable name when one of the following rules matches:

1. The string has at least one special character.
2. The string has at least one uppercase letter.
3. The string starts with one or more digits.

These rules are defined as follows:

```
1. MixedSpecialChar = {AnyChar}{SpecialChars}+{AnyChar}
2. AtLeastOneUppercase = {AnyChar}[A-Z]+{AnyChar}
3. DigitsAtThebegining = [0-9]+{AnyChar}
```

Where `{AnyChar}` represents a combination of letters with numbers and `{SpecialChars}` is a list of some special characters defined as follows:

- `SpecialChars = [_\!@#\$%\^&\.\?\|\/\{\}\[\]\`\~\"\'\;]`
- `AnyChar = [0-9a-zA-Z]*`

Finally, a string is not a valid variable name when one of the following rules matches:

```
NotVarName = {MixedSpecialChar} | {AtLeastOneUppercase} |
{DigitsAtThebegining}
```

Example:

Matched string	**Rule**
var_name	`MixedSpecialChar`
validname?	`MixedSpecialChar`
invaliDName	`AtLeastOneUppercase`
9variable	`DigitsAtThebegining`

2.3. Number (`{Number}`)

A string that is a number matches one of the following rules:

1. It has one digit from 0 to 9.
2. It has more than one digit starting always with a number different than 0.

The RE that matches this description is:

```
[0-9]|([1-9][0-9]*)
```

The green part matches first rule. The blue part matches the second rule.

Example:

Matched string	**Not matched string**
0 9 145	09 0009

2.4. Not a number (`{NotNumber}`)

A string that is not a number matches all of the following rules:

1. It starts with zero.
2. It is followed by at least one number.

The RE that matches with the above description is:

```
0+[0-9]+
```

The green part matches first rule. The blue part matches the second rule.

Example: 0009, 0123, 010

2.5. Program Name (`{ProgamName}`)

A string that is a valid Program Name matches all of the following rules concatenated:

1. It starts with an uppercase letter.
2. It could be followed by lowercase or uppercase letter as well as digits.
3. The string needs to have at least one lowercase letter.

The regular expression that matches this description is:

```
[A-Z][0-9a-zA-Z]*[a-z]+[0-9a-zA-Z]*
```

The green part matches the first rule. The blue part matches the second rule. The yellow part matches the third rule.

This regular expression is represented as:

`[A-Z]{AnyChar}[a-z]+{AnyChar}`

Where `{AnyChar}` matches a combination of letters with numbers

Example:

Matched string	**Not matched string**
SumTwoNumbers AllGrammar PairRecognizer Factorial	SUMTWONUMBERS aLLgRamar pairRecognizer FACTORIAL

2.6. Not a Program Name (`{NotProgramName}`)

A string that is not a program name matches one of the following rules:

1. It starts with a lowercase letter and it is followed by at least one letter or number.
2. Alternately to the rules above, the string will match if it has any character followed by one or more special characters followed by any character.
3. All uppercase letters with numbers.

The regular expression that matches this description is:

`[a-z]([A-Za-z0-9]+)|{MixedSpecialChar}|[A-Z0-9]+`

The blue part matches the first rule. The yellow part matches the second rule. The green part matches the third rule.

Where `{MixedSpecialChar}` matches a string that contains at least one special character.

Example:

Matched string	**Rule**
program_name	Second rule
programName?	Second rule
pRogramName	First rule
PROGRAMNAME	Third rule

3. Error Messages

When there is a string that matches with `{NotNumber}`, `{NotProgamName}`, `{NotVarName}` rules, a message output will be displayed after the identifiers list displaying certain information related to the error: the type of error, the line, the column and the value of the string that threw the error.

As an example, consider that the following .sf file is passed to the lexical analyzer:

```
BEGINPROG FACTORIAL

VARIABLES WRONG_variable, n
n := 0123

ENDPROG€
```

We know that:

- `FACTORIAL` will match with `{NotProgamName}` because it is all uppercase.
- `WRONG_variable` will match with `{NotVarName}` because it is not lowercase only.
- `0123` will match with `{NotNumber}` because it starts with zero.
- € (note `ENDPROG€`) will match with “one char” rule (.) because it was not matched with any other rule.

Consequently these strings will not be tokenized nor displayed in the list of tokens and the following list of errors will be displayed at the end of the output:

```
-------- Errors ---------------
SYNTAX ERROR PROGNAME: FACTORIAL is not well defined. Please check
line: 1 column: 10
SYNTAX ERROR VARNAME: WRONG_variable is not well defined. Please check
line: 3 column: 10
SYNTAX ERROR NUMBER: 0123 is not well defined. Please check line: 4
column: 5
SYNTAX ERROR NOT RECOGNIZED CHAR: € not recognized. Please check line:
6 column: 7
```

Each of them corresponds to an inconsistency between the definition of the language and the program itself.

At this point the Lexical Analyzer already knows that the program has some errors and it can stop the compiling process and notify the user. Nevertheless, this is the only information that the lexical analyzer knows.

4. Bonus Question

In this application, the long comment state deactivates the tokenizer process until it finds the close-comment string (*/) and activates it again but this could have been done with a regular expression that matches: `"/*"[A-Za-z0-9]*"*/"` since nested comments are forbidden for this kind of language. Nevertheless, the language that describes nested comments is not regular because it is necessary to remember how many open-comment strings were already read and no DFA can recognize this language, so by definition no regular expression can match it.[3]

This technical issue can be resolved using states in JFlex in combination with some Java code. The idea is to have a counter variable that is initialized in 0, once the scanner recognizes the open-comment string (/*) it increments the counter and enters into a state. In this state three options are given:

1. Read an open-comment string will increment the counter.
2. Read a closed-comment string will decrease the counter.
3. Read any other character will not affect the counter value.

Each time that a closed-comment is read the java code will check if it arrived to 0. If this is the case, it will jump again into the `YYINITIAL` state. If it is not the case it will stay in this state until the process finishes.

In the case that the scanner process finishes, and the counter is greater than 0, means that there were more open-comment strings. Otherwise, if the counter is less than 0 there were more closed-comment strings than open-comment ones. A stack could have been used also with this purpose, but the idea of a counter is simpler.

5. Description of Test files

File Name	**Description**
counter.sf	Given a number n, read n times a number and sum up.
grater_than.sf	Returns the greater value of two numbers, if the values are the same it returns -1
sum_two_numbers.sf	It contains invalid rules like the program name has special characters, some variables starts with numbers, etc.
all_gramar.sf	It contains all the keywords defined in the language.

[3] The formal proof of this statement was omitted but basically it is the same as the one studied during classes where L = {a^n b^n | n > 0}

6. Regular Expression to ε-NFA

Kleene's theorem implies that all finite automata recognize regular languages and all regular languages are recognized by a finite automaton. The `{Number}` rule with the RE `([0-9]|([1-9][0-9]*))` has been chosen in order to illustrate such theorem in the present project.