Automaten en Formele Talen Assignment 1
Lexical Analysis

Deadline: 23:59, Friday, April 20, 2018

Inge Bethke

Created by Daan de Graaf

Last changes on: April 3, 2018 (Bas van den Heuvel)

Background

The phenomena found in the Automata and Formal Languages (AFL) field are broadly spread across computer science. To illustrate some of the capabilities of this phenomena, we compile and finally execute a simple source code using the automata and formal languages you will find in the course. The source code exists of one or multiple simple arithmetic expressions. In this assignment we start with the lexical analysis of the input source and try to perform a syntactical analysis as well.

Simple arithmetic expression

The arithmetic expressions that are used have an initialising form: `var = 3 + 3`. This means that a variable followed by an equal sign followed by an expression is obligatory. This becomes useful when storing and reusing variables for assignment 3.
The tokens that can occur are illustrated with a Finite Automaton (FA) in Figure 3 in Appendix A.

Lexical Analysis (`lexer.py`)

The lexical analysis of the source code, also known as tokenizing, creates a token for a sequence of characters. The `create_lexer()` function creates the automaton shown in Appendix A and returns it.

Syntax Analysis (`parser.py`)

The syntax analysis takes the tokens from the lexer and performs a check on the structure of the code. First we try using the automaton shown in Figure 4. If the tokens are accepted by the automaton, the syntax of the code is right.

Assignment 1 (100 points)

Use Python 3 to test your code, and try to follow the **PEP8 style guide**.

Assignment 1.1 (55 points)

For implementing the lexer, complement the `lexer(M, source)` function in the `lexer.py` file. It takes the source code and should return a list of tuples. Your job is to create that list of tuples, containing the token and its identifier (the name of the state). As an example, see Figure 1. By checking possible transitions between the states, you will be able to find an identifier for each token. If all is well, the program creates a `.lex` file you can use as input for the parser.

ID	**EQU**	**INT**	**ADD**	**LBT**	**INT**	**MUL**	**INT**	**RBT**	**EOF**
Var	=	*34*	+	(	*5*	*	*56*	)	\n

Figure 1: Output of the lexer

Assignment 1.2 (45 points)

(30 points) With the use of the output from the lexing process, you are able to parse the source and check its syntax. Your job is to complement the `create_parser()` and `parser(M, source)` functions. Use the automaton from Figure 4 to recreate the automaton in the `create_parser()` function. The `parser(M, source)` function has to be completed such that it returns `True` for a correct syntax and `False` for an incorrect syntax. Use the parser from `create_parser()` to move through the automaton.
(8 points) Also try to create an error handler with an arrow (^) pointing to the error in the syntax, and an error message based on the state the error occurred in (see Figure 2).

```
'test = ( 3 - + 3 ) \n'
              ^
SyntaxError: Expected INT or ID
```

Figure 2: Example error message

(7 points) If all done correctly, explain why the approach with a DFA does not work for matching brackets. Write your answer in a comment at the top of your `parser.py`.

Submission

Put your versions of `lexer.py` and `parser.py` in a zip or tar with your name and student number and upload it to Canvas **before 23:59, Friday, April 20, 2018**. *If your code does not work with Python 3, your work will not be graded and you will receive 0 points.*

A Lexical Analysis

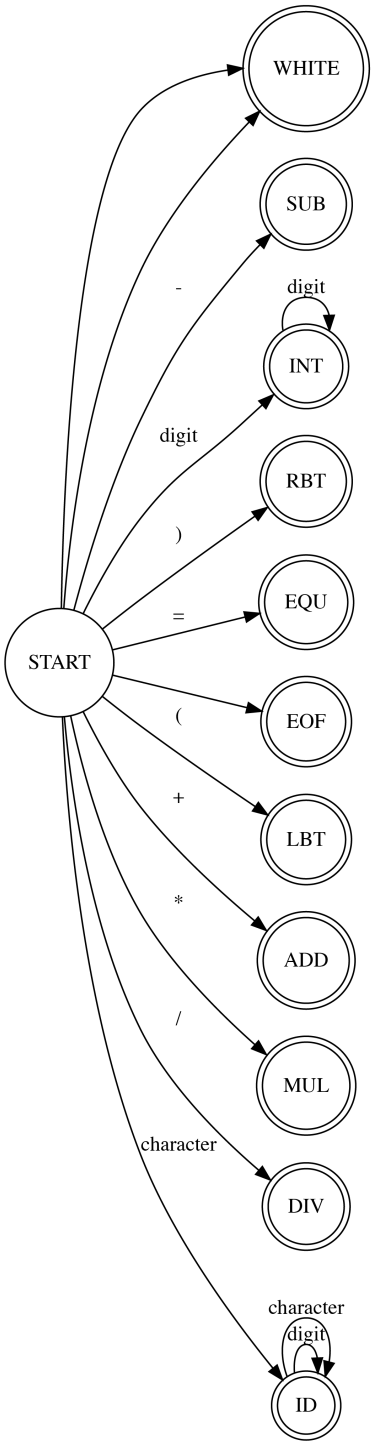

Figure 3: Finite automata for tokenizing the input source.

B Syntax Analysis

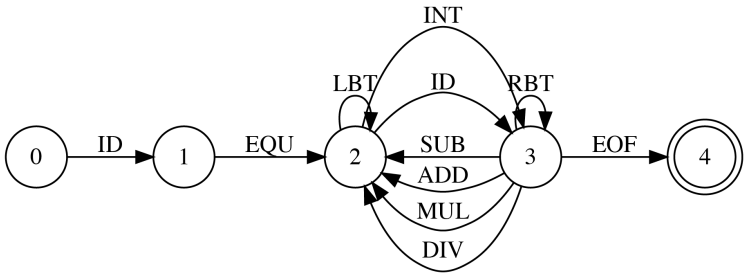

Figure 4: Finite automata for parsing the input source.