Creating a Language for **EXT**Lex

Summary

1. Introduction

This guide will help you create the files necessary for you to perform lexical analysis of a custom programming language in **EXT**Lex. The files necessary for **EXT**Lex to run are:

- The language's **automaton file.**
- The language's **reserved words file.**
- A **code file** to be analysed.

Figure 1 illustrates how **EXT**Lex uses those files in order to identify tokens and lexical errors in comparison to applications that compile lexical analysers:

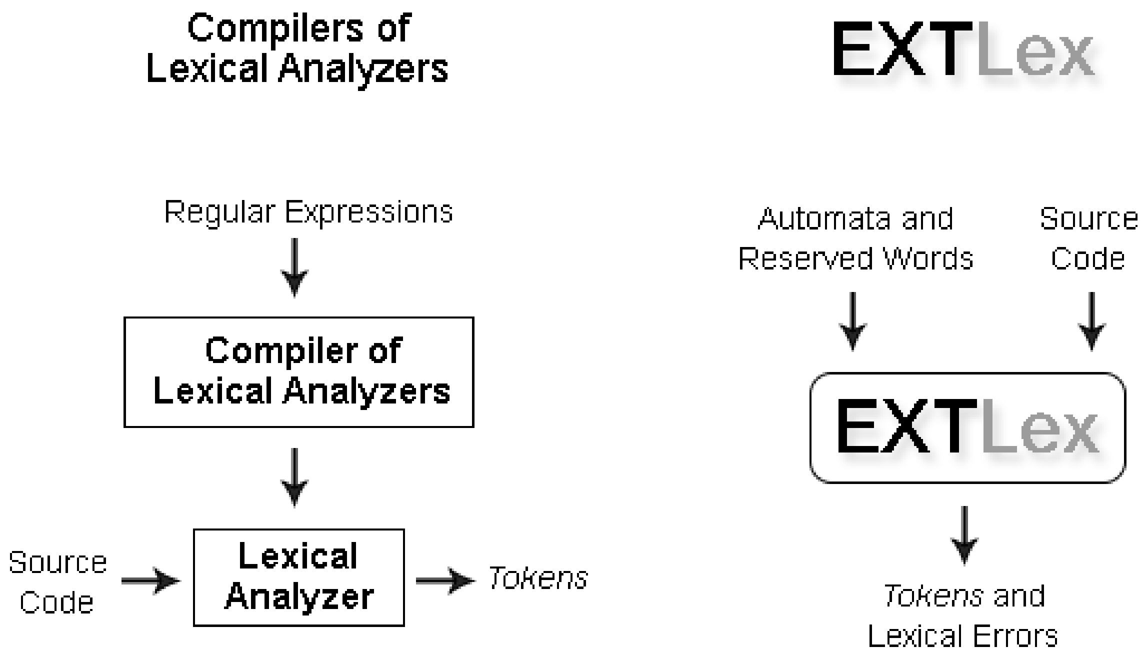
Compilers of Lexical Analyzers
Regular Expressions
Compiler of Lexical Analyzers
Source Code
Lexical Analyzer
Tokens
EXTLex
Automata and Reserved Words
Source Code
EXTLex
Tokens and Lexical Errors

Figure 1 – Comparison between **EXT**Lex and compilers of lexical analysers

The code file, as its name suggests, must have the code which you intend to analyse. The automaton file and the reserved words file are the ones that describe your programming language. Each of them has a distinct format, which is described in detail in the following chapters.

2. Creating the Automaton File

2.1. Understanding Automatons in Lexical Analysis

The **automaton file** is the one that must contain the automaton which describes the lexical structure of your programming language. Suppose that we want to build a lexical analyser for a programming language which has the following tokens:

Token	Regular Expression
<integer>	[0-9]+
<float>	[0-9]+.[0-9]+
<char>	\’[A-z0-9]\’
<string>	\”[A-z0-9]+\”
<comment>	#[A-z 0-9]#
<operator>	<(=) | >(=) | [~=]=
<identifier>	[A-z][A-z0-9]*
Reserved Characters	; | (|) | * | + | - | / | ,

Table 1 – Tokens of a programming language

One can notice that the tokens in Table 1 are described as regular expressions. Those regular expressions can be transformed in an automaton. Consider, for an example, the automaton illustrated in Figure 2, which represents the **<integer>** and **<float>** tokens:

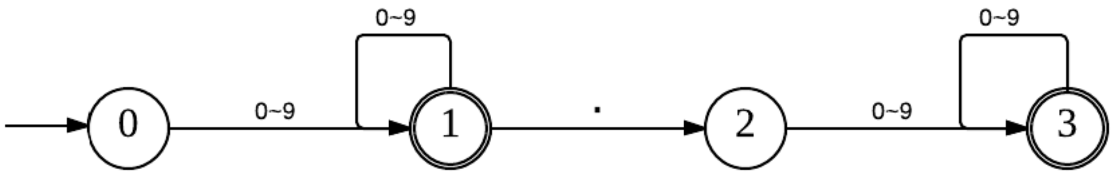

Figure 2 – Automaton that represents the <integer> and <float> tokens

Consider a lexical analyser that reads one character at a time from a code, and then updates its current analysis state according to this automaton. In this scenario, the meaning of each state in the automaton of Figure 2 is:

- **State 0:** Initial state. That is the lexical analyser’s state before reading any characters from a code file.

- **State 1:** Final state for the identification of am <integer> token. This is the state to which the lexical analyser is positioned after reading an initial number character between 0 and 9. Afterwards, the lexical analysis continues in State 1 after reading whichever amount of number characters. If no more characters are found after the sequence of number characters, the lexical analysis is concluded, and a well formed <integer> token is identified. Some examples of integers which could be identified in a code through this automaton are “1”, “123456” and “99999999999”.

- **State 2:** Error state. After reading an arbitrary sequence of number characters from 0 to 9, the lexical analyser is positioned in State 2 if a “.” (dot) character is found next. This indicates that the lexeme being read from the code file is not an <integer>, but a <float>, since it has a dot after a sequence of number characters. If no number characters are found after the dot, then the lexical analysis stops at State 2, which is not a final state. This characterizes an error, since it was expected that at least one number character would be present after the dot in order for a <float> token to be successfully formed.

- **State 3:** Final state for the identification of a <float> token. The lexical analyser is positioned in State 3 if at least one number character is read from the code after the “.” (dot). Once no more number characters from 0 to 9 are found in the code, the lexical analysis is then concluded, and a well formed <float> token is identified. Some examples of well-formed <float> tokens are: “1.0”, “123.456” and “9999999.99999999”.

An automaton that describes all tokens in Table 1 can also be constructed. Figure 3, which is in the next page, shows how such automaton can be structured.

Any programming language can be represented by such automatons. Sophisticated languages may require many complex resources and mechanisms for their analysis to be possible, but even their foundational lexical structures can be described by an automaton. **EXT**Lex provides a syntax in order for the user to represent such automatons, and hence allow for **EXT**Lex to analyse any programming language conceived by the user. This syntax is described in the following section.

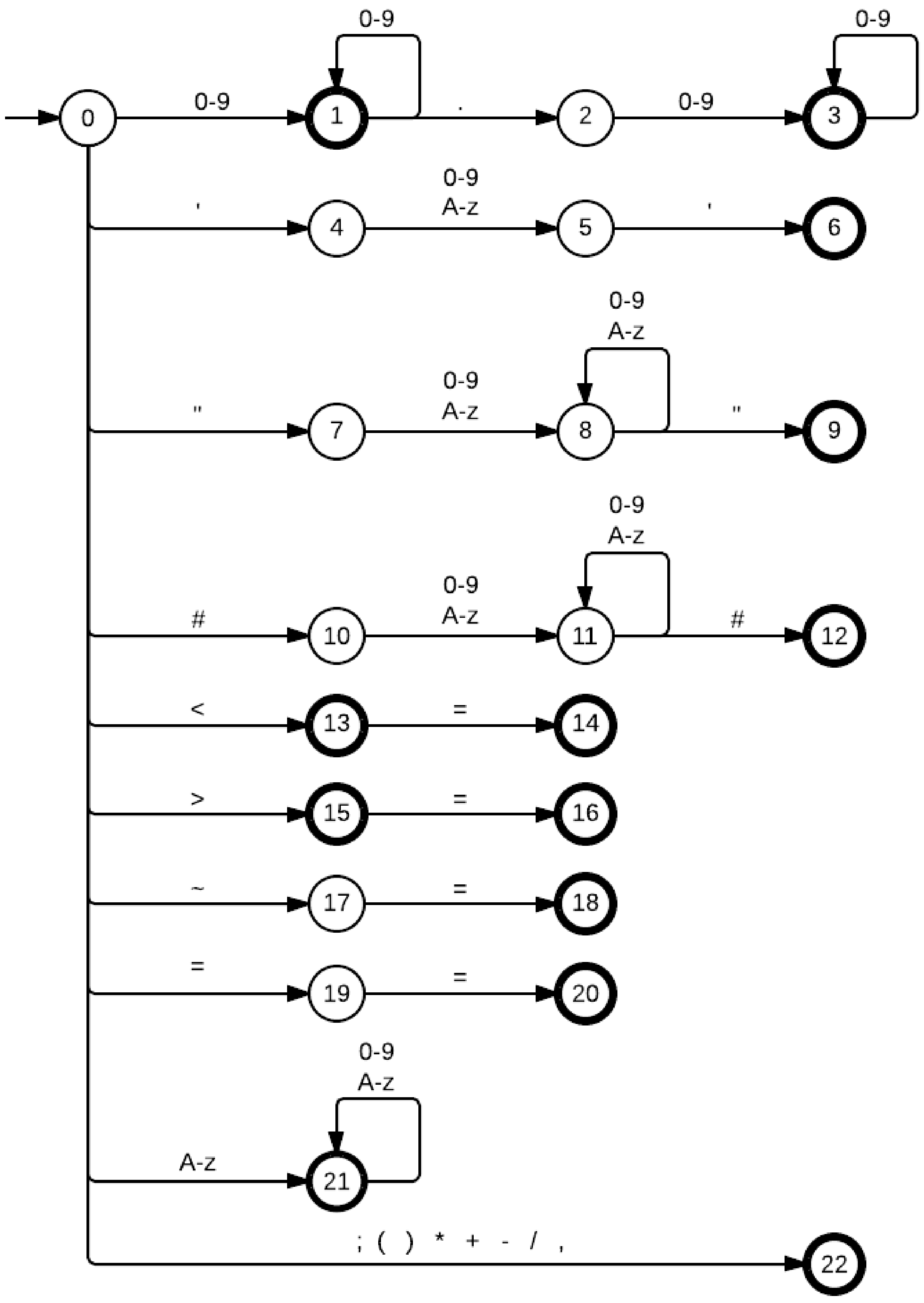

Figure 3 – Automaton that recognizes all tokens in Table 1

2.2. The **EXT**Lex Syntax

The automaton file that describes the lexical structure of a programming language must respect the syntax described in this tutorial, otherwise, **EXT**Lex will not be capable of identifying tokens properly.

In the automaton file must be found all states of the automaton which describe a given programming language. Each state must be described in the automaton file in the following manner:

Description	File Data
Name of the state.	01: <name>
Resulting token or error message of the state.	02: <token_name_or_error_message>
Lexical action of the state.	03: <lexical_action>
Transition: ASCII code and name of resulting state.	04: <ASCII> <state_name>
Transition: ASCII code range and name of resulting state.	05: <ASCII>~<ASCII> <state_name>
Empty line indicating end of state's description	06:

Table 2 – Structure of an automaton state in the automaton file of **EXT**Lex

As described in Table 2, each state must be separated by an empty line. The last line of the automaton file must also be empty. The following subsections clarify on the details of the **EXT**Lex automaton file syntax.

2.2.1. Initial State

The initial state of the automata will be the first one found in the automaton file.

2.2.2. Lexical Actions

The "lexical action" of each state represents the action which must be taken by **EXT**Lex in case the lexical analysis is finished in a given state. **EXT**Lex recognizes four lexical actions: LEXEME, ERROR, NEWLINE or NULL. Below are the descriptions of each lexical action:

- **LEXEME:** Should be used only in final states that represent a well-formed token. When the lexical analysis is finished in a state marked by the "LEXEME" lexical action, **EXT**Lex will store the <resulting_token> value of the state along with the lexeme identified, and position the current lexical analysis state on the automaton's initial state. If the state is characterized by the "LEXEME" lexical action, then the second line of the state's description in the file should contain the name of the resulting well-formed token. Some examples of token names are "<integer>" and "<float>".

- **ERROR:** Should be used only in non-final states that represent a malformed token. If a state's lexical action is "ERROR", then the second line of the state's description in the automaton file must contain not a token name, but an error message. The error message can be of any size and should describe the type of error assigned to the state, such as "Malformed float number" or "String with missing closing quote". When the lexical analysis is finished in a state marked by the "ERROR" lexical action, **EXT**Lex will store the resulting error message along with the line and character indicators of where the error can be found in the code and position the current lexical analysis state on the automaton's initial state.

- **NEWLINE:** Should be used in states which result from the reading of a line breaking character, such as "\n". When the lexical analysis is finished in a state marked by the "NEWLINE" lexical action, **EXT**Lex will discard the character, increment the line counter and position the current lexical analysis state on the automaton's initial state. The transitions and the token name/error message assigned for the state in the automaton file will be ignored.

- **NULL:** Should be used in states which result from the reading of a meaningless character or empty space, such as "\s" and "\t". When the lexical analysis is finished in a state marked by the "NULL" lexical action, **EXT**Lex will empty the lexeme buffer and set the current lexical analysis state on the automaton's initial state.

2.2.3. Transitions

As shown in Table 2, transitions can be structured in two ways:

- <ASCII> <state_name>

- <ASCII>~<ASCII> <state_name>

The <ASCII> element represents the integer number that describes a given character in the ASCII table. The character "a", for an example, is identified by the integer number 97. The website www.asciitable.com contains the integer numbers that represent every character in the ASCII table. As of now, **EXT**Lex does not recognize characters that cannot be found in the ASCII table.

The **"<ASCII> <state_name>"** transition expression has the following meaning: update the lexical analysis state to <state_name> if the character <ASCII> is read. The transition **"97 <state_1>"**, for an example, means that if the character "a" (**97** in ASCII) is read, the lexical analysis state will be updated to **<state_1>**.

The **“<ASCII>~<ASCII> <state_name>”** transition expression represents a range of ASCII characters, and has the following meaning: update the lexical analysis state to <state_name> if any character in the range <ASCII>~<ASCII> is read. The transition **“97~122 <state_1>”**, for an example, means that if a character between “a” (**97** in ASCII) and “z” (**122** in ASCII) is read, the lexical analysis state will be updated to **<state_1>**.

2.2.4. Examples

Consider the automaton of Figure 2. An equivalent representation of such automaton in **EXT**Lex syntax is described in Table 3.

Line	**Content in EXTLex Syntax**
01:	0
02:	No characters read.
03:	ERROR
04:	48~57 1
05:	
06:	1
07:	<integer>
08:	LEXEME
09:	48~57 1
10:	46 2
11:	
12:	2
13:	Incomplete float number.
14:	ERROR
15:	48~57 3
16:	
17:	3
18:	<float>
19:	LEXEME
20:	48~57 3
21:	

Table 3 – Example of automaton in **EXT**Lex syntax

An example of a larger automaton can be found in the file **“automata.txt”**, which can be found at the folder **“EXTLex/example/”** of this package. It contains states with all types of lexical actions.

3. Creating the Reserved Words File

3.1. Understanding Reserved Words in Lexical Analysis

Almost every programming language has reserved words. They are words which can be used in very specific contexts when coding in a given programming language, and usually comprise words associated with resources such as repetition loops, conditional expressions and etc.

Some examples of words which are reserved in many distinct languages are **“for”**, **“while”**, **“if”** and **“else”**. When a word is reserved, it cannot be used to name variables, procedures and functions. In technical terms, they cannot be **identifiers**.

3.2. The **EXT**Lex Syntax

The syntax for writing the reserved words file for **EXT**Lex is very simple. The file must contain all reserved words of a given programming language, one per line. Consider a programming language that has the reserved words “for”, “while” and “int”. The reserved words file in **EXT**Lex syntax to represent such programming language should be structured as described in Table 4.

Line	Description	File Entries
01:	First reserved word.	for
02:	Second reserved word.	while
03:	Third reserved word.	int
04:	Empty line indicating end of file.	

Table 4 - Example of reserved word list in **EXT**Lex syntax

When **EXT**Lex successfully finds a well-formed token, it tries to find the lexeme of the token in the reserved words list. If the lexeme is a reserved word, then, instead of storing the token along with its lexeme in the list of tokens found, it replaces the token name with the lexeme itself.

Suppose that a code written in the same programming language described by Table 4 has a variable named “while”. Now suppose that the token which represents variables in such programming language is named “<variable>”. When **EXT**Lex finds the variable of name “while”, it will search for it in the list of reserved words before storing the pair **“<variable> while”** in the list of tokens found. As so happens, **“while”** will be found to be a reserved word, and instead of storing the pair **“<variable> while”**, **EXT**Lex will store the pair **“while while”** in the list of tokens found.

4. Final Remarks

This tutorial clarifies on how to represent the lexical structure of a programming language so that **EXT**Lex can perform lexical analysis adequately. **EXT**Lex needs two files to be able to analyse codes written in a given programming language: the **automaton file** and the **reserved words file**. The steps of creating each of those files are described in detail throughout Chapters 1, 2 and 3.

For guidance as to how to run **EXT**Lex or incorporate it into your own Java lexical analyser, please refer to the **README.txt** file included in this package. If you use any of the resources from this package to build a tool or publish an academic article, please cite the following paper:

Gustavo Henrique Paetzold and Eler Elisandro Schemberger. **EXTLex: Um Analisador Léxico Configurável.** V EPAC - Encontro Paranaense de Computação. 2013.