Lexi — Lexical Analyzer

Jared Dyreson, Chris Nutter

California State University, Fullerton

Contents

1 Introduction

This project was intended to be a lexical analyzer for our compiler and was named “Lexi”. The goal of Lexi is to parse out the contents of a source document and generate meaningful lexemes. These lexemes were to adhere to a specific set of regular expressions to define a token.

2 How to Use

All code is compiled under the latest version of “clang” and “g++”.

1. Use on a Unix-based operating system, preferably Linux
2. Clone the repository from here
3. Enter the directory named “Lexi”
4. Run 'make test' to conduct all the tests without having to present command line arguments

3 Design — Regular Expression

The way Lexi parses each line and determines the identifier type is through the use of regular expressions. Being able to determine the identifier is crucial in defining the token's contents. Lexi after processing the file and creating a vector of strings that parses line by line which is then fed through a function that reads each character and determines one of the each lexeme types.

3.1 Comments

These are any sequence of characters enclosed by two exclamation marks, and can be embedded in a line with other code. Multiple comments in a line are supported.

```
(!.*!)
```

3.2 Identifier

These are any sequence of characters and numbers starting with an alphanumeric. It should also be known that we must compare any identifier matched to a list of reserved words. These reserved words are not to be used as variable names as the are used for data types, control-flow operators, and other key-defining words for the language. This is only supported in this iteration because *Flex* and *Bison* do this quite well.

```
([a-zA-Z]+(\d*)?)
```

3.3 Numbers

These are any integer like number, such as floating point and ordinary base 10 integers. Lexi also supports signed/unsigned numbers.

```
(?:\b)([-+]?\d*.?\d+)?(?=\b)
```

3.4 Operators

These operators should be broken down into two distinct categories:

- Unary
- Binary

Where the unary operator only takes in one argument and returns a value. The binary operator takes in two arguments and returns one value.

3.4.1 Binary — Comparators

```
(>|<|>=|<=|&&|||)
```

The operators supported here are as follows:

- Less than
- Less than or equal to
- Greater than
- Greater than or equal to
- Logical AND, OR

3.4.2 Binary — Operands

```
(+|-|*|/|=|>|<|>=|<=|&|||%|^)
```

The operators supported here are as follows:

- Addition
- Subtraction
- Multiplication
- Division
- Modulo
- Bitwise AND, OR, XOR

There is a subset of these binary operators that are exclusively used as comparators and deserve to be in their own separate category.

3.4.3 Unary

```
(^!$)
```

The only operator here is the invert/not operator, which will take in boolean value and return the inverse of it. For example, if the value is set to true and you apply the not operator, you will get false.

3.5 Separators

These symbols will typically enclose an expression that needs either be evaluated, contains string literals or separate multiple values of the same type. Here, it is easier to note the LHS (left hand side) and RHS (right hand side) variants of each separator, as they are important to distinguish in the beginning.

3.5.1 Brackets

```
({)
```

Figure 1: LHS Bracket

```
(})
```

Figure 2: RHS Bracket

3.5.2 Comma

```
(\,)
```

Figure 3: Comma Character

Identifiers are comma separated for things like instantiation for multiple variables of the same time.

3.5.3 Parenthesis

```
(()
```

Figure 4: LHS Parenthesis

```
())
```

Figure 5: RHS Parenthesis

3.5.4 String Literals

```
(\"|\’)
```

These are expressions should contain alphanumeric characters enclosed by either single or double quotes. Single quotes should be used for single char like data member. Double quotes are string literals.

3.6 Terminators

These symbols signal the end of an expression.

```
(\;|\$)
```