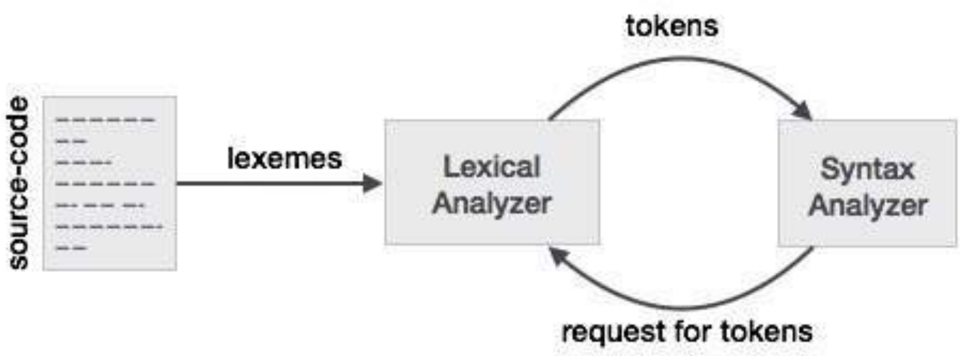
CS323 Compliers - Project 1

> 张佳晨
>
> 11713020 ZHANG, Jiachen

Introduction

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Lexical Analysis

Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

In our SPL language, major tokens are defines as follows

Arithmetic Symbols	`+`, `-`, `*`, `/`
Punctuation	`,`, `;`, `.`, `(`, `)`, `[`, `]`, `{`, `}`
Assignment	`=`
Comparison	`<`, `<=`, `==`, `>=`, `>`, `!=`
Key words	`int`, `char`, `float`, `struct`, `if`, `else`, `while`, `return`
Logical	`&&`, `||`, `!`
INT	/* integer in 32-bits (decimal or hexadecimal) */
FLOAT	/* floating point number (only dot-form) */
CHAR	/* single character (printable or hex-form) */
ID	/* identifier */

Some lexeme errors are listed below:

- undefined tokens, eg., '@', '#' as non-literal token
- illegal hex int, eg., 0x5gg
- illegal hex char, eg., '\x9', '\xt0'

1. Multi-line Comment

In order to support multiple line comment like this as below,

```
/*
comments
*/
```

`/\*.*\*/` is not work for flex on multiple Lines.

So I search on the intenet and finally chose to use flex `<STATE>` to solve this issue.

```
"/*"              { BEGIN(C_COMMENT); }
<C_COMMENT>"*/"   { BEGIN(INITIAL); }
<C_COMMENT>.      { }
<C_COMMENT>\n     { yycolno = 1; }
```

And define `%x C_COMMENT` in previous section.

2. Single-line Comment

Single line comment is quite easy, so I just implement it using regular expression below

```
"//".*[\n\r]
```

3. Character Recognition - A Simple Approach

In order to recognize character between `'\x0'` and `'\xff'`, and other printable characters, I've already defined the regular expression below,

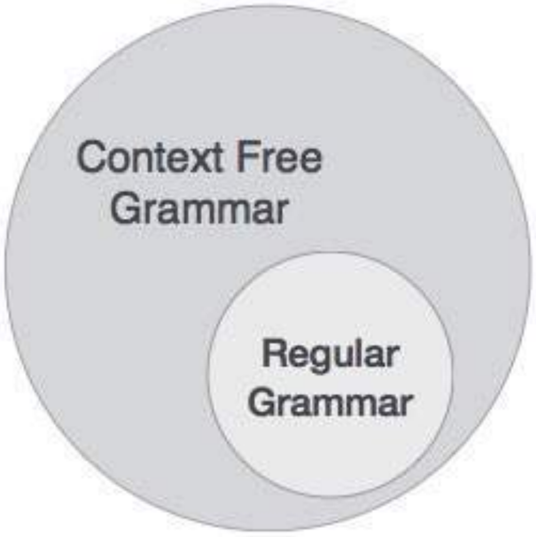
```
('[ -~]')|('\\[xX][0-9a-fA-F]{1,2}')
```

How ever, for other format-invalid character-like token, such as `'\xZ'`, it's tedious to list all the possible expression, so I define `wrong_char ('.+')` as recognizer, which is put after `char ('[ -~]')|('\\[xX][0-9a-fA-F]{1,2}')` to fix it.

Syntax Analysis

Syntax analysis or parsing is the second phase of a compiler. I've learned the basic concepts used in the construction of a parser.

The lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.



1. Implementation

The Implementation of Syntax Analyses is not hard. For most of the CFG, just list it.

But the Context-Dependent Precedence is hard to work. This sounds outlandish at first, but it is really very common. For example, a minus sign typically has a very high precedence as a unary operator, and a somewhat lower precedence (lower than multiplication) as a binary operator. I've solve it with the help of link

Reference

1. Compiler Design - Lexical Analysis
2. Bison-Context-Dependent Precedence