CS-1319 - Monsoon 2023 - Assignment 2

Team 9: Saawan Ka Paani

October 1, 2023

1 Introduction

The nanoC lexer is designed to tokenize and recognize elements of the nanoC programming language. It utilizes the Flex tool to define regular expressions for identifying keywords, identifiers, constants, string literals, punctuators, and comments. In this document, we will explain the working of the lexer and the design choices made to achieve its functionality.

2 Lexer Components

2.1 Regular Expression Definitions

The lexer defines regular expressions for various elements, such as keywords, identifiers, constants, string literals, punctuators, comments, and whitespace. Each regular expression is enclosed in double quotes and defines a specific pattern for tokenization.

```
CHAR "char"
ELSE "else"
FOR "for"
IF "if"
INT "int"
RET "return"
VOID "void"
IDENTIFIER [a-zA-Z][a-zA-Z0-9]*
INTEGER_CONSTANT (\+|-)?[1-9][0-9]*|(0)[0-9]*?
CHARACTER_CONSTANT '([^'\\]|\\[\\'\"?abfnrtv])'
STRING_LITERAL "["^"\\\n][\n]*(\\0)?([^"\\\n]*[\n]*)?["]
PUNCTUATOR (\[|\]|\(|\)|\{|\}|->|&|\*|\+|-|\/|%|!|\?|<|>|<=|>=|==|!=|&&|\|\||=|:|;|,)
MULTI_LINE_COMM (\/\*)[^(\*\/)]*(\*\/)
SINGLE_LINE_COMM \/\/[^\n]*
WS [ \t\n]
```

Listing 1: Regular Expression Definitions

2.2 Rules and Actions

The lexer defines rules and corresponding actions to execute when a token is recognized. Each rule consists of a regular expression pattern and an associated action. When a token matches a pattern, the lexer executes the action.

```
{CHAR}                { printf("<KEYWORD, char>"); /* Keyword Rule */}
{ELSE}                { printf("<KEYWORD, else>"); /* Keyword Rule */}
{FOR}                 { printf("<KEYWORD, for>"); /* Keyword Rule */}
{IF}                  { printf("<KEYWORD, if>"); /* Keyword Rule */}
{INT}                 { printf("<KEYWORD, int>"); /* Keyword Rule */}
{RET}                 { printf("<KEYWORD, return>"); /* Keyword Rule */}
{VOID}                { printf("<KEYWORD, void>"); /* Keyword Rule */}
{IDENTIFIER}          { printf("<IDENTIFIER, \%s>", yytext); /* Identifier Rule */ }
{INTEGER_CONSTANT}    { printf("<INTEGER CONSTANT, \%s>", yytext); /* Literal Rule */ }
{CHARACTER_CONSTANT}  { printf("<CHARACTER CONSTANT, \%s>", yytext); /* Literal Rule */ }
```

```
{STRING_LITERAL}      { printf("<STRING LITERAL, \%s>", yytext); /* Literal Rule */ }
{PUNCTUATOR}          { printf("<PUNCTUATOR, \%s>", yytext); /* Statement Rule */ }
{MULTI_LINE_COMM}     { printf("<MULTI LINE COMMENT, \%s>", yytext); /* Multiple Comment
    Rule */ }
{SINGLE_LINE_COMM}    { printf("<SINGLE LINE COMMENT, \%s>", yytext); /* Single Comment Rule
     */ }
{WS}                  { /* Ignore white-space */ }
.                     { printf("<INVALID TOKEN>"); exit(1); }
```

Listing 2: Rules and Actions

3 Lexer Working

The working of the lexer is such that it applies the defined regular expressions in the order they are listed. It scans the input source code in the order of character by character and matches the characters against the regular expressions defined. When a match is found, the associated action is executed.

For example, when the lexer encounters the keyword "int", it matches the regular expression for keywords, executes the associated action, and prints `<KEYWORD, int>`. Similarly, it recognizes identifiers, constants, string literals, punctuators, and comments based on the specified patterns.

In the nanoC compiler Whitespace characters are ignored by the lexer, and invalid tokens trigger an error message `<INVALID TOKEN>` which results in the immediate exit from the program.
For instance if `p1 = $` is the input, the output is `<IDENTIFIER, p1><PUNCTUATOR,=><INVALID TOKEN>`. After this the program terminates $ sign is not recognised in our regular expression definition.

4 Design Choices

Several design choices were made in the lexer's design:

- Each regular expression is crafted to match specific elements of the nanoC language defined for us ensuring accurate tokenization.
- Tokens are labeled with their types, such as KEYWORD, IDENTIFIER, CONSTANT, STRING LITERALS, PUNCTUATORS to provide meaningful information.
- Whitespace is ignored as per the description of the nano C.
- Invalid tokens trigger an error message, helping to identify issues in the source code by terminating immediately.

5 Conclusion

The nanoC lexer is designed to accurately tokenize nanoC source code, providing essential information about the code's structure and elements. Its use of regular expressions and well-defined rules and actions allows it to effectively recognize and label tokens in nanoC programs.