INDEX

Table of Contents

1. Experiment name: Write a simple lex specification to recognize the following verb, is, am, are, were, Write a lex program to recognize different types of operator.

Theory:

Lexical analysis is the first phase of the compiler design process. The main objective of this phase is to convert the input program into a sequence of tokens that can be processed by the parser. Lexical analysis is implemented using a scanner or a lexer, which identifies the basic building blocks of the input program. The scanner reads the input program character by character, matches them with a pattern, and generates a corresponding token.

Scanner:

In this experiment, we need to write a scanner to recognize the verbs "is," "am," "are," and "were," and a lex program to recognize different types of operators. The scanner reads the input program character by character and matches them with the patterns defined in the lex program. If a match is found, the corresponding token is generated.

Interaction with the parser:

The output generated by the scanner is used as an input to the parser. The parser uses the tokens generated by the scanner to build a parse tree. The parse tree represents the syntactic structure of the input program.

Lex:

The lex program for recognizing the verbs "is," "am," "are," and "were" is as follows:

```
%{
#include <stdio.h>
%}
%%
is      printf("IS\n");
am      printf("AM\n");
are     printf("ARE\n");
were    printf("WERE\n");
%%

int main(){
    yylex();
    return 0;
}
```

The lex program for recognizing different types of operators is as follows:

```
%{
#include <stdio.h>
%}

%%
[+-*/]  printf("OPERATOR: %c\n",yytext[0]);
=       printf("OPERATOR: =\n");
==      printf("OPERATOR: ==\n");
!=      printf("OPERATOR: !=\n");
<=      printf("OPERATOR: <=\n");
>=      printf("OPERATOR: >=\n");
<       printf("OPERATOR: <\n");
>       printf("OPERATOR: >\n");
%%

int main(){
    yylex();
    return 0;
}
```

If an input program contains any of the verbs "is," "am," "are," or "were," the corresponding token (IS, AM, ARE, or WERE) is generated. Similarly, if the input program contains any operator (+, -, *, /, =, ==, !=, <=, >=, <, >), the corresponding token is generated.

Implementation environment and hardware information:

This experiment can be implemented on any system with a C compiler and the Flex tool installed.

Result:

Input Program:

```
The cats are sleeping.

x = 10;

y = x * 2;
```

Output:

```
ARE

OPERATOR: =

OPERATOR: *

OPERATOR: ;

OPERATOR: =

OPERATOR: ;
```

Discussion:

In this experiment, we learned how to write a lex program to recognize different types of tokens. The scanner reads the input program character by character and generates corresponding tokens. The output generated by the scanner is used as an input to the parser, which builds a parse tree. The parse tree represents the syntactic structure of the input program. The lex program can be extended to recognize other types of tokens, such as keywords, identifiers, and literals. The Flex tool provides a simple and efficient way to generate scanners for various programming languages.

2. Experiment Name: Write a simple lex specification to recognize different keywords.

Theory:

A keyword is a reserved word in a programming language that has a predefined meaning and cannot be used as an identifier. These keywords are recognized by the compiler or interpreter and used to build the syntax of the program. In compiler design, lex is a tool used to generate lexical analyzers, which are used to identify the tokens in a program.

Scanner:

The scanner, also known as a lexical analyzer, is the first phase of a compiler. Its main function is to read the input source code and convert it into a sequence of tokens, which are passed on to the next phase, the parser. The scanner uses regular expressions to recognize keywords, identifiers, operators, and other elements of the source code.

Interaction with the Parser:

The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

Lex:

Lex is a tool used to generate lexical analyzers based on regular expressions. The specification for recognizing different keywords using Lex would include a set of declaration and transition rules. The declaration rules define the regular expressions for each keyword, while the transition rules define the action to be taken when a keyword is recognized.

An example of a declaration rule for the keyword "if" would be:

```
IF          if
```

This declaration rule defines the regular expression "if" as a keyword and assigns it the token name "IF."

An example of a transition rule for the "IF" keyword would be:

```
{IF}            { return IF; }
```

This transition rule specifies that when the scanner recognizes the "IF" keyword, it should return the token "IF."

Implementation environment and hardware information:

The implementation environment for this experiment would depend on the specific tools used. Lex can be used on various operating systems, including Windows, Linux, and macOS. The hardware requirements would depend on the size of the input source code and the complexity of the Lex specification.

Result:

Suppose we have the following input source code:

```
if (x > y) {
    z = x + y;
}
```

The Lex specification for recognizing the "if" keyword would generate the following output:

```
IF '(' ID '>' ID ')' '{' ID '=' ID '+' ID ';' '}'
```

This output represents the sequence of tokens generated by the scanner for the input source code.

Discussion:

In this experiment, we have learned how to use Lex to generate a lexical analyzer for recognizing keywords in a programming language. The specification for recognizing keywords includes a set of declaration and transition rules. The declaration rules define the regular expressions for each keyword, while the transition rules define the action to be taken when a keyword is recognized. The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

3. Experiment Name: Write a simple lex specification to recognize real numbers.

Theory:

Real numbers are decimal numbers with or without a fractional part. In programming languages, real numbers are represented using floating-point or double-precision numbers. In compiler design, Lex is a tool used to generate lexical analyzers, which are used to identify the tokens in a program.

Scanner:

The scanner, also known as a lexical analyzer, is the first phase of a compiler. Its main function is to read the input source code and convert it into a sequence of tokens, which are passed on to the next phase, the parser. The scanner uses regular expressions to recognize real numbers, identifiers, operators, and other elements of the source code.

Interaction with the Parser:

The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

Lex:

Lex is a tool used to generate lexical analyzers based on regular expressions. The specification for recognizing real numbers using Lex would include a set of declaration and transition rules. The declaration rules define the regular expressions for recognizing real numbers, while the transition rules define the action to be taken when a real number is recognized.

An example of a declaration rule for recognizing real numbers would be:

```
REAL         [0-9]*\.?[0-9]+
```

This declaration rule defines a regular expression that matches real numbers, including those with or without a fractional part. The regular expression allows for zero or more digits before the decimal point and one or more digits after the decimal point.

An example of a transition rule for recognizing real numbers would be:

```
{REAL}           { return REAL; }
```

This transition rule specifies that when the scanner recognizes a real number, it should return the token "REAL."

Implementation Environment and Hardware Information:

The implementation environment for this experiment would depend on the specific tools used. Lex can be used on various operating systems, including Windows, Linux, and macOS.

The hardware requirements would depend on the size of the input source code and the complexity of the Lex specification.

Result:

Suppose we have the following input source code:

```
x = 3.14;
```

The Lex specification for recognizing real numbers would generate the following output:

```
ID '=' REAL ';'
```

This output represents the sequence of tokens generated by the scanner for the input source code.

Discussion:

In this experiment, we have learned how to use Lex to generate a lexical analyzer for recognizing real numbers in a programming language. The specification for recognizing real numbers includes a set of declaration and transition rules. The declaration rules define the regular expressions for recognizing real numbers, while the transition rules define the action to be taken when a real number is recognized. The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

4. Experiment Name: Write a simple lex specification to recognize identifiers.

Theory:

Identifiers are user-defined names given to various programming constructs like variables, functions, classes, etc. In compiler design, Lex is a tool used to generate lexical analyzers, which are used to identify the tokens in a program.

Scanner:

The scanner, also known as a lexical analyzer, is the first phase of a compiler. Its main function is to read the input source code and convert it into a sequence of tokens, which are passed on to the next phase, the parser. The scanner uses regular expressions to recognize identifiers, real numbers, operators, and other elements of the source code.

Interaction with the Parser:

The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

Lex:

Lex is a tool used to generate lexical analyzers based on regular expressions. The specification for recognizing identifiers using Lex would include a set of declaration and transition rules. The declaration rules define the regular expressions for recognizing identifiers, while the transition rules define the action to be taken when an identifier is recognized.

An example of a declaration rule for recognizing identifiers would be:

```
ID           [a-zA-Z][a-zA-Z0-9]*
```

This declaration rule defines a regular expression that matches identifiers, including letters and numbers. The regular expression allows for at least one letter followed by zero or more letters or numbers.

An example of a transition rule for recognizing identifiers would be:

```
{ID}           { return ID; }
```

This transition rule specifies that when the scanner recognizes an identifier, it should return the token "ID."

Implementation environment and hardware information:

The implementation environment for this experiment would depend on the specific tools used. Lex can be used on various operating systems, including Windows, Linux, and macOS. The hardware requirements would depend on the size of the input source code and the complexity of the Lex specification.

Result:

Suppose we have the following input source code:

```
int sum = 0;
```

The Lex specification for recognizing identifiers would generate the following output:

```
TYPE ID '=' INT ';'
```

This output represents the sequence of tokens generated by the scanner for the input source code.

Discussion:

In this experiment, we have learned how to use Lex to generate a lexical analyzer for recognizing identifiers in a programming language. The specification for recognizing identifiers includes a set of declaration and transition rules. The declaration rules define the regular expressions for recognizing identifiers, while the transition rules define the action to be taken when an identifier is recognized. The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

5. Experiment Name: Write a simple lex specification to recognize integers.

Theory:

Integers are a common data type in programming languages, used to represent whole numbers. In compiler design, Lex is a tool used to generate lexical analyzers, which are used to identify the tokens in a program.

Scanner:

The scanner, also known as a lexical analyzer, is the first phase of a compiler. Its main function is to read the input source code and convert it into a sequence of tokens, which are passed on to the next phase, the parser. The scanner uses regular expressions to recognize integers, identifiers, operators, and other elements of the source code.

Interaction with the Parser:

The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

Lex:

Lex is a tool used to generate lexical analyzers based on regular expressions. The specification for recognizing integers using Lex would include a set of declaration and transition rules. The declaration rules define the regular expressions for recognizing integers, while the transition rules define the action to be taken when an integer is recognized.

An example of a declaration rule for recognizing integers would be:

```
DIGIT       [0-9]

INTEGER     {DIGIT}+
```

This declaration rule defines a regular expression that matches integers, including digits from 0 to 9. The regular expression allows for at least one digit followed by one or more digits.

An example of a transition rule for recognizing integers would be:

```
{INTEGER}   { return INTEGER; }
```

This transition rule specifies that when the scanner recognizes an integer, it should return the token "INTEGER."

Implementation environment and hardware information:

The implementation environment for this experiment would depend on the specific tools used. Lex can be used on various operating systems, including Windows, Linux, and macOS. The hardware requirements would depend on the size of the input source code and the complexity of the Lex specification.

Result:

Suppose we have the following input source code:

```
int x = 123;
```

The Lex specification for recognizing integers would generate the following output:

```
TYPE ID '=' INTEGER ';'
```

This output represents the sequence of tokens generated by the scanner for the input source code.

Discussion:

In this experiment, we have learned how to use Lex to generate a lexical analyzer for recognizing integers in a programming language. The specification for recognizing integers includes a set of declaration and transition rules. The declaration rules define the regular expressions for recognizing integers, while the transition rules define the action to be taken when an integer is recognized. The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

6. Experiment Name: Write a simple lex specification to recognize floats.

Theory:

Floats, also known as floating-point numbers, are a common data type in programming languages used to represent decimal numbers. In compiler design, Lex is a tool used to generate lexical analyzers, which are used to identify the tokens in a program.

Scanner:

The scanner, also known as a lexical analyzer, is the first phase of a compiler. Its main function is to read the input source code and convert it into a sequence of tokens, which are passed on to the next phase, the parser. The scanner uses regular expressions to recognize integers, floats, identifiers, operators, and other elements of the source code.

Interaction with the Parser:

The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

Lex:

Lex is a tool used to generate lexical analyzers based on regular expressions. The specification for recognizing floats using Lex would include a set of declaration and transition rules. The declaration rules define the regular expressions for recognizing floats, while the transition rules define the action to be taken when a float is recognized.

An example of a declaration rule for recognizing floats would be:

```
DIGIT       [0-9]

FLOAT       {DIGIT}+"."{DIGIT}+
```

This declaration rule defines a regular expression that matches floats, including digits from 0 to 9, and a period "." to represent the decimal point. The regular expression allows for at least one digit before and after the decimal point.

An example of a transition rule for recognizing floats would be:

```
{FLOAT}     { return FLOAT; }
```

This transition rule specifies that when the scanner recognizes a float, it should return the token "FLOAT."

Implementation environment and hardware information:

The implementation environment for this experiment would depend on the specific tools used. Lex can be used on various operating systems, including Windows, Linux, and macOS. The hardware requirements would depend on the size of the input source code and the complexity of the Lex specification.

Result:

Suppose we have the following input source code:

```
float x = 1.23;
```

The Lex specification for recognizing floats would generate the following output:

```
TYPE ID '=' FLOAT ';'
```

This output represents the sequence of tokens generated by the scanner for the input source code.

Discussion:

In this experiment, we have learned how to use Lex to generate a lexical analyzer for recognizing floats in a programming language. The specification for recognizing floats includes a set of declaration and transition rules. The declaration rules define the regular expressions for recognizing floats, while the transition rules define the action to be taken when a float is recognized. The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

7. Experiment Name: Write a simple lex specification to recognize positive and negative integers and floats.

Theory:

In programming languages, integers and floats are two common data types used to represent numerical values. An integer is a whole number without a decimal point, while a float is a number with a decimal point. In this experiment, we will write a Lex specification to recognize positive and negative integers and floats.

Scanner:

The scanner, also known as a lexical analyzer, is the first phase of a compiler. Its main function is to read the input source code and convert it into a sequence of tokens, which are passed on to the next phase, the parser. The scanner uses regular expressions to recognize integers, floats, identifiers, operators, and other elements of the source code.

Interaction with the Parser:

The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

Lex:

The Lex specification for recognizing positive and negative integers and floats includes a set of declaration and transition rules. The declaration rules define the regular expressions for recognizing positive and negative integers and floats, while the transition rules define the action to be taken when an integer or float is recognized.

An example of a declaration rule for recognizing positive and negative integers and floats would be:

```
DIGIT        [0-9]

INT          [-]?{DIGIT}+

FLOAT        [-]?{DIGIT}+"."{DIGIT}+
```

This declaration rule defines a regular expression that matches both positive and negative integers and floats. The regular expression for integers allows for an optional negative sign (-) followed by one or more digits, while the regular expression for floats allows for an optional negative sign, one or more digits before and after the decimal point, and a required decimal point.

An example of a transition rule for recognizing positive and negative integers and floats would be:

```
{INT}        { return INT; }
```

```
{FLOAT}     { return FLOAT; }
```

These transition rules specify that when the scanner recognizes an integer or a float, it should return the token "INT" or "FLOAT," respectively.

Implementation environment and hardware information:

The implementation environment for this experiment would depend on the specific tools used. Lex can be used on various operating systems, including Windows, Linux, and macOS. The hardware requirements would depend on the size of the input source code and the complexity of the Lex specification.

Result:

Suppose we have the following input source code:

```
int x = -123;

float y = 1.23;
```

The Lex specification for recognizing positive and negative integers and floats would generate the following output:

```
TYPE ID '=' INT ';'

TYPE ID '=' FLOAT ';'
```

This output represents the sequence of tokens generated by the scanner for the input source code.

Discussion:

In this experiment, we have learned how to use Lex to generate a lexical analyzer for recognizing positive and negative integers and floats in a programming language. The specification for recognizing integers and floats includes a set of declaration and transition rules. The declaration rules define the regular expressions for recognizing positive and negative integers and floats, while the transition rules define the action to be taken when an integer or float is recognized. The scanner interacts with the parser by passing a stream of tokens generated from the input source code. The parser uses these tokens to construct the syntax tree, which represents the program's structure.

8. Experiment name: Write a simple lex specification to recognize different punctuation symbols.

Theory:

Punctuation symbols are characters used to aid comprehension and provide clarity to written text. In programming, punctuation symbols are also used to indicate the structure of the code. Examples of punctuation symbols include commas, semicolons, parentheses, brackets, and curly braces.

Scanner:

A scanner is a program that reads in input text and produces a stream of tokens, which are used by the parser to construct a syntax tree. In this experiment, the scanner will read in input text and recognize different punctuation symbols, producing a stream of tokens that indicate which symbol was recognized.

Interaction with the parser:

The tokens produced by the scanner will be used by the parser to construct a syntax tree. The punctuation symbols recognized by the scanner will be used to indicate the structure of the code.

Lex:

The following lex specification can be used to recognize different punctuation symbols:

```
%{
#include <stdio.h>
%}


%%
","         { printf("COMMA\n"); }
";"         { printf("SEMICOLON\n"); }
"("         { printf("LEFT PARENTHESIS\n"); }
")"         { printf("RIGHT PARENTHESIS\n"); }
"["         { printf("LEFT BRACKET\n"); }
"]"         { printf("RIGHT BRACKET\n"); }
"{"         { printf("LEFT CURLY BRACE\n"); }
"}"         { printf("RIGHT CURLY BRACE\n"); }
%%
```

```
int main() {
    yylex();
    return 0;
}
```

The above lex specification defines different punctuation symbols as regular expressions on the left-hand side of the rules, with the corresponding action to be taken on the right-hand side enclosed in curly braces. The action in this case is simply to print out the name of the token.

Implementation environment and hardware information:

This lex specification can be implemented on any platform with a C compiler and the flex tool installed.

Result:

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        printf("%d is even\n", i);
    } else {
        printf("%d is odd\n", i);
    }
}
```

Output:

```
for LEFT PARENTHESIS int i = 0 SEMICOLON i < 10 SEMICOLON i++
RIGHT PARENTHESIS LEFT CURLY BRACE

if LEFT PARENTHESIS i % 2 == 0 RIGHT PARENTHESIS LEFT CURLY
BRACE

printf LEFT PARENTHESIS "%d is even\n" COMMA i RIGHT
PARENTHESIS SEMICOLON

RIGHT CURLY BRACE else LEFT CURLY BRACE

printf LEFT PARENTHESIS "%d is odd\n" COMMA i RIGHT
PARENTHESIS SEMICOLON

RIGHT CURLY BRACE

RIGHT CURLY BRACE
```

Discussion:

The lex specification defined above is a simple and efficient way to recognize punctuation symbols in input text. It can be easily extended to recognize additional symbols by adding more rules to the specification. The output produced by the lex scanner can be used by the parser to construct a syntax tree, allowing for the analysis and execution of the code.

9. Experiment name: Write a simple lex specification to recognize digits

Theory:

In lexical analysis, a digit is a numeric character used in the representation of numbers in various writing systems. Digits can be used in a variety of contexts, such as representing a numerical value or as part of an identifier. A lexical analyzer or scanner is responsible for identifying digits and other types of tokens in the input stream of characters. The lex tool is a popular tool for generating lexical analyzers and is widely used in compiler construction.

Scanner:

The scanner, also known as a lexer or tokenizer, reads the input text and breaks it into a stream of tokens. In this experiment, we will write a lex specification to recognize digits in the input stream.

Interaction with the parse:

The output of the scanner is a stream of tokens, which is then used by the parser to construct a parse tree. In this experiment, we will only focus on the scanner part, and the output will be a stream of tokens representing the digits in the input.

Lex:

The following is a simple lex specification to recognize digits:

```
%{
#include <stdio.h>
%}

digit [0-9]

%%

{digit}+     printf("DIGIT: %s\n", yytext);

%%

int main() {
    yylex();
    return 0;
}
```

In the above lex specification, we declare a variable `digit` that matches any digit from 0 to 9. Then, we define a rule `{digit}+` that matches one or more digits and prints out the matched text using the `printf` function. Finally, we define a `main` function that calls the `yylex` function to start scanning the input.

Implementation environment and hardware information:

The lex tool is a widely used lexical analyzer generator and is available on many platforms, including Unix-based systems and Windows. It does not require any specific hardware requirements and can be run on any modern computer.

Result:

Let's assume the input stream contains the following text:

```
12345 67890
```

When we run the above lex specification on the input, we get the following output:

```
DIGIT: 12345

DIGIT: 67890
```

Discussion:

In this experiment, we have written a simple lex specification to recognize digits in the input stream. The lex specification declares a variable `digit` that matches any digit from 0 to 9 and uses it to define a rule that matches one or more digits. The rule prints out the matched text using the `printf` function. The output of the scanner is a stream of tokens representing the digits in the input. This experiment demonstrates how to use the lex tool to generate a lexical analyzer for recognizing digits in the input stream.