Neil Barry-Murphy

13327106

Compiler Design: Lexical Analyser Report

Compiler Design – Lab 2 (Lexical Analyser):

Lab Task:

Using finite state techniques design a lexical analyser for processing a sequence of 32-bit octal, hexadecimal and signed integer constants (lexemes) described by the following regular expression:

[0-7] + [bB] | [0-9a-fA-F] + [hH] | ((+|–) ? [0-9] +)

Code the lexical analyser in either C or C++ and verify that it works correctly by writing a test program to read a sequence of lexemes from the standard input, printing each one recognized on the standard output together with a description of its corresponding lexical token. Any invalid lexemes should be flagged.

Design:

Please refer to the comments on my attached source code when reading this report. They will explain function purposes, as well as the general structure of the program. Firstly, the lexical analyser was designed purely around finite state machines. The program is spilt into 3 separate finite state machines. The first deals with decimal only input, the second handles octal constants, and the third deals with hexadecimal constants. All machines are designed to accept user input, and are designed as follows: (Maximum permitted decimal constant is: 2147483647)

Side Note:

The **getCurrentExponent(...)** function not only calculates the exponent of a given octal or hexadecimal constant (bases 8 & 16 respectively), but also returns the original integer number with all calculations applied. For example, give the function **getCurrentExponent(...)** the parameters 8 (n) & 2 (exponent). The method will return the integer value 64.

Decimal FSM:

An input is entered by the user as a char array (String), and is converted to a 32-bit integer value using the checkDecimalConstantValidity(...) method. See the included source code for more documentation on this method. If this value is greater than the max, it is rejected with the error message:

```
if ((stringToDecimal > MAX_DECIMAL_LIMIT) || (i > 10) || (stringToDecimal < 0)) {
        printf("\n!*!*Value processed is outside the permissable range of 2^31 - 1.
                (Overflow)\n");
        printf("\n");
}
```

The guard: **(stringToDecimal < 0)** is extremely important at this stage of the loop. Without this error check, the method would print a negative value due to overflow. If the user-entered lexeme manages to get through all of the error-check guards, the string is then confirmed as a lexeme, and is printed by the method as confirmation. (See method on attached source code).

Finally, the following is used to ensure that a hexadecimal or octal lexeme taken as the user defined parameter is not simply converted to a decimal:

```
if (((int) lexeme[i] - ASCII_INT_OFFSET >= 0) && ((int) lexeme[i] –
        ASCII_INT_OFFSET <= 9)) {
        ...
}
```

This guard ensures that the value stored with **(int) lexeme[i]** for the current value of "i" within the for loop is not a character, and is an integer value between 0 and 9 inclusive.

Hexadecimal FSM:

Again, this method takes a character array as user defined input, and returns true if the input is a hexadecimal lexeme, and false if not. There are 4 main guards within this method:

(1) Check if current character >= (int) 0 && <= (int) 9 ||
(2) Check if current character >= 'a' && <= ' f' ||

(3) Check if current character >= ‘A’ && <= ‘ F’
(4) Check if the lexeme passed is actually a hexadecimal constant. (‘h’ ll ‘H’ appended to the end of the user defined string).

These guards are necessary, as they ensure that the character values, ‘a’, ‘f, ‘C’, etc. are converted to their respective integer representations.

If the lexeme is not hexadecimal, the following message is returned:
printf("\nERROR. This lexeme is not a valid hexadecimal constant");

If overflow has occurred in the calculation of the exponent in the currently processed character of the **(int) lexeme[i]**, (the value returned is negative, etc.), the following message is returned:
printf("\nArithmetic operation has exceeded permissable parameters.");
printf("\nError, the converted decimal constant contains too many digits. (Overflow has occurred)");

If overflow has occurred at the end of the calculation, (the value returned is greater than the maximum permitted integer constant), the following is returned:
printf("\nValue processed is outside the permissable range of 2^31 - 1. (Overflow)");

If the user defined input is in fact a hexadecimal lexeme, the lexeme constant is printed to the console for confirmation purposes.

<u>Octal FSM:</u>

This finite state machine is very similar to the hexadecimal FSM, except that there is only one main guard:
(1) Check if the lexeme passed is actually an octal constant. (‘b’ ll ‘B’ appended to the end of the user defined string).

Again, the same error messages are printed if the error checks fail at any point in the method:

If the lexeme is not octal, the following message is returned:
printf("\nERROR. This lexeme is not a valid octal constant");

If overflow has occurred in the calculation of the exponent in the currently processed character of the **(int) lexeme[i]**, (the value returned is negative, etc.), the following message is returned:
printf("\nArithmetic operation has exceeded permissable parameters.");
printf("\nError, the converted decimal constant contains too many digits. (Overflow has occurred)");

If overflow has occurred at the end of the calculation, (the value returned is greater than the maximum permitted integer constant), the following is returned:
printf("\nValue processed is outside the permissable range of 2^31 - 1. (Overflow)");

If the user defined input is in fact an octal lexeme, the lexeme constant is printed to the console for confirmation purposes.

<u>**Final Method:**</u>

The final **int main()** method is used to accept input from the user, and confirm if this constant is a valid octal, hexadecimal or decimal only lexeme. Once again, please see attached source code for further documentation.

<u>**Testing:**</u>

The following Boolean values were created and used to test all forms of input form users. The following will confirm if all error checks have been implemented correctly. The user input system was also tested, and the results were also found to be valid.

```
//a few quick manual tests relevant to the commented sections of code
below.
char lex1[] = "4096";                               //is a lexeme
char lex2[] = "4321056b";                           //is a lexeme
char lex3[] = "4321056h";                           //is a lexeme
char lex4[] = "123456789h";                    //is NOT a lexeme
char lex5[] = "4321086b";                           //is NOT a lexeme
char lex6[] = "7ff329h";                       //is a lexeme
char lex7[] = "324FA2Bh";                           //is a lexeme
```

```
bool test = checkOctalConstantValidity(lex2);
printf("%s", test ? "true" : "false");
printf("\n");

bool test2 = checkOctalConstantValidity(lex5);
printf("%s", test2 ? "true" : "false");
printf("\n");

bool test4 = checkHexadecimalConstantValidity(lex4);
printf("%s", test4 ? "true" : "false");
printf("\n");

bool test5 = checkHexadecimalConstantValidity(lex6);
printf("%s", test5 ? "true" : "false");
printf("\n");

bool test6 = checkHexadecimalConstantValidity(lex7);
printf("%s", test6 ? "true" : "false");
printf("\n");
```

Place the above tests into the main() method in the source code to confirm that all tests have been correctly implemented, and that all error guards are in working order.

Octal Finite State Machine Diagram:

If any tables are occupied other than those of the "ACCEPT" row, the lexeme is invalid. We assume that the lexeme here is compared to octal validations only for illustrative purposes. ('b/B' appended to String, lexeme converted to Decimal).

Example 1:
char lexeme[];
lexeme = "24516b"
Usage = | 2 | 4 | 5 | 1 | 6 | b
Usage = 0 | 1 | 2 | 3 | 4 | 5 | 6

	2	4	5	1	6	B/b	
0-7	1	2	3	4	5	6	ACCEPT
8-9							REJECT
a-z							REJECT
A-Z							REJECT
<2^31 || >2^31 -1							REJECT

Conclusion: VALID

Example 2:
char lexeme[];
lexeme = "31481a9b"
Usage = | 3 | 1 | 4 | 8 | 1 | a | 9 | b
Usage = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8

	3	1	4	8	1	a	9	B/b	
0-7	1	2	3		5		7	8	ACCEPT
8-9				4					REJECT
a-z						6			REJECT
A-Z									REJECT
<2^31 || >2^31 -1									REJECT

Conclusion: INVALID

<u>Hexadecimal Finite State Machine Diagram:</u>

If any tables are occupied other than those of the "ACCEPT" row, the lexeme is invalid. We assume that the lexeme here is compared to hex validations only for illustrative purposes. ('h/H' appended to String, lexeme converted to Decimal).

<u>Example 1:</u> (VALID)
char lexeme[];
lexeme = "24B1dH"
Usage = | 2 | 4 | 5 | 1 | 6 | b
Usage = 0 | 1 | 2 | 3 | 4 | 5 | 6

	2	4	B	1	d	H/h	
0-9	1	2		4		6	ACCEPT
a-f					5		ACCEPT
A-F			3				ACCEPT
g-z							REJECT
G-Z							REJECT
<2^31 || >2^31 -1							REJECT

<u>Example 2:</u> (INVALID)
char lexeme[];
lexeme = "3G481a9h"
Usage = | 3 | G | 4 | 8 | 1 | a | 9 | h
Usage = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8

	3	G	4	8	1	a	9	H/h	
0-7	1		3	4	5		7	8	ACCEPT
a-f						6			ACCEPT
A-F									ACCEPT
g-z									REJECT
G-Z		2							REJECT
<2^31 || >2^31 -1									REJECT

<u>Decimal Finite State Machine Diagram:</u>

If any tables are occupied other than those of the "ACCEPT" row, the lexeme is invalid. We assume that the lexeme here is compared to decimal validations only for illustrative purposes. (<u>nothing</u> appended to String, lexeme converted to Decimal).

<u>Example 1:</u> (VALID)
char lexeme[];
lexeme = "987423"
Usage = | 2 | 4 | 5 | 1 | 6 | b
Usage = 0 | 1 | 2 | 3 | 4 | 5 | 6

	9	8	7	4	2	3	
0-9	1	2	3	4	5	6	ACCEPT
a-z							REJECT
A-Z							REJECT
<2^31 || >2^31 -1							REJECT

<u>Example 2:</u> (INVALID)
char lexeme[];
lexeme = "3G481a9"
Usage = | 3 | G | 4 | 8 | 1 | a | 9
Usage = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

	3	G	4	8	1	a	9	
0-7	1		3	4	5		7	ACCEPT
a-z						6		REJECT
A-Z		2						REJECT
<2^31 || >2^31 -1								REJECT

<u>Note:</u> *Assume that the above lexical constants have been converted to decimal format before any conclusions are drawn regarding the final state of the lexeme (Valid || Invalid). Please reference the attached code to understand how this conversion takes place. It has already been briefly outlined above.*

General Transition Table:

Input -> State	0-7 *Decimal*	a, c-f, A, C-F *Hex*	8, 9 *!Octal*	B/b *Terminate Octal/End*	H/h *Terminate Hex*	+ / - *Sign*	*Accept ?*
1	2	4	3			6	
2	2	4	3	8			Y
3	3	4	3	4	5		Y
4	4	4	4	4	5		
5							Y
6	7		7				
7	7		7				Y
8	4	4	4				Y

(1) Initial State
(2) 0-7
(3) 8,9
(4) a, c-f
(5) Terminate Hex State
(6) Sign
(7) Decimal
(8) Valid Octal digit 'b'