CS3071 Lab 2

Name: Laura Murphy
Student ID: 13326075

INTRODUCTION

In the following pages you will find the design documentation for my attempt at coding a lexical analyzer that processes a sequence of 32-bit octal, hexadecimal and signed integer constants. I will explain how the program works and discuss some of the decisions I made. You will see the design for the finite state-processing machine that describes the solution. To discover the states I would need, I explored usages that could be created using the given regular expression. This helped me realize how I needed to categorize the valid inputs and ultimately shaped my code.

STATES

0 – START: Starting state
1 – LEAD: Have seen one or more leading zeros.
2 – SLEAD: Have seen a sign, followed by one or more leading zeros.
3 – ALL: Have only seen one or more digits from 0-7, so all bases are still possible.
4 – NOCT: Have seen at least an 8 or a 9, so the constant is **not octal.**
5 - EOCT: Just saw a b or B, so if we've reached the end, the number is octal. Otherwise it may be hexadecimal (but is definitely **not decimal**).
6 – HEX: We've seen at least one letter that isn't a b/B, or we've seen a b/B that wasn't the octal indicator (we know that since it wasn't followed by the end marker). This means the constant is hexadecimal, but is missing the hex indicator.
7 – EHEX: Just saw the hex indicator.
8 – DECS: Just saw a sign (still needs to be followed by a digit to be valid)
9 – DECD: Have seen digits following a sign.

There are states for acknowledging leading zeros separately since the program needs to keep count of them. The count is then used when the program is calculating whether overflow will occur or not.

INPUTS

I decided to map the different input types to numbers. We are not particularly concerned whether an input is a "5" or a "6", once we know that is a digit between 1 and 7 (i.e. its "input type"). To make it easier to look up the next state in the table, I created a function (***get_input_type***) that will take in a char and output the input number/type that it corresponds to. Having the types as numbers allows me to use them as the indexes for the table columns in the code.

The inputs are mapped to numbers as follows:

0	-> 0
1-7	-> 1
8, 9	-> 2
b, B	-> 3
h, H	-> 4
a, A, c, C, d, D, e, E, f, F	-> 5
-, +	-> 6

FINITE STATE PROCESSING MACHINE

Starting state: START

States: {START, LEAD, SLEAD, ALL, NOCT, EOCT, HEX, EHEX, DECS, DECD, error}

Inputs: {0, 1-7, 8|9, b|B, h|H, a|A|c|C|d|D|e|E|f|F, -|+}

Transitions: See transition table below.

Accepting States: {LEAD, SLEAD, ALL, NOCT, EOCT, EHEX, DECD}

Rejecting States: {START, HEX, DECS}

TRANSITION TABLE

Inputs222

States	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**End**
0	1	3	5	6		6	8	
1	1	3	5	4	7	6		"DEC"
2	2	9	9					"DEC"
3	3	3	5	4	7	6		"DEC"
4	6	6	6	6	7	6		"OCT"
5	5	5	5	6	7	6		"DEC"
6	6	6	6	6	7	6		
7								"HEX"
8	2	9	9					
9	9	9	9					"DEC"

All empty entries are transitions to the error state.

A note on implementation: I kept the translation from transition table to code as literal as possible.

Originally, I toyed with the idea of using switch and conditional statements for finding the next state. I decided against this because I felt it was too lengthy and overcomplicated the code. Instead, I used a 2-dimensional array to represent the transition table, giving me a clear and concise program.

In the image below, you can see the table in the code. The structure of the table is immediately obvious, so it is very readable.

```
181     // table[X][Y] = Z; ....When in state X, on input Y, go to state Z
182     table[0][0] = 1; table[0][1] = 3; table[0][2] = 5; table[0][3] = 6;                 table[0][5] = 6; table[0][6] = 8;
183     table[1][0] = 1; table[1][1] = 3; table[1][2] = 5; table[1][3] = 4; table[1][4] = 7; table[1][5] = 6;
184     table[2][0] = 2; table[2][1] = 9; table[2][2] = 9;
185     table[3][0] = 3; table[3][1] = 3; table[3][2] = 5; table[3][3] = 4; table[3][4] = 7; table[3][5] = 6;
186     table[4][0] = 6; table[4][1] = 6; table[4][2] = 6; table[4][3] = 6; table[4][4] = 7; table[4][5] = 6;
187     table[5][0] = 5; table[5][1] = 5; table[5][2] = 5; table[5][3] = 6; table[5][4] = 7; table[5][5] = 6;
188     table[6][0] = 6; table[6][1] = 6; table[6][2] = 6; table[6][3] = 6; table[6][4] = 7; table[6][5] = 6;
189
190     table[8][0] = 2; table[8][1] = 9; table[8][2] = 9;
191     table[9][0] = 9; table[9][1] = 9; table[9][2] = 9;
```

As I mentioned previously, the input type is used to index the column of the table. The current state is used to index the row. The content of the element gives you the next state that you must transition to (see line 181).

HOW THE PROGRAM WORKS

The user inputs one or more space-separated constants as command line arguments to the program. The program works on each constant one by one and outputs the results as it goes along.

The program first evaluates whether or not the constant is a valid Hexadecimal, Octal or Integer value. Then it calculates whether or not overflow will occur if we try to store the constant as a 32-bit decimal value. If overflow will occur, the program prints an error and moves on to the next constant (or ends, if there are no more constants). Otherwise, it converts the constant to its decimal equivalent and prints a description of its corresponding lexical token.

To work out whether the constant is a valid hex, oct or int value, we transition through the table. ***get_next_state*** is called on each char in the input. The function is very simple – it just updates the current state to the next state given by accessing the table. If the new state happens to be the error state, ***get_next_state*** will return a 0 so we know that the constant is invalid. If the new state indicates that we have just seen a leading zero, the function increments the relevant counter.

When it reaches the end of the constant, the program checks that it is in a valid end state for at least one of the bases (hex, oct, int). If it is, ***print_lexical_token*** is called. This function checks for overflow and converts the constant to decimal.

To check for overflow with a hex number, we simply have to make sure the number of digits (not including leading zeros) is not greater than 8.

For octal numbers, if there are greater than 11 digits (not including leading zeros) overflow will occur. If there are exactly 11 digits, overflow will occur if the most significant digit is greater than 3.
For integers, if there are greater than 10 digits (not including leading zeros) overflow will occur. If there are exactly 11 digits, we must compare our value with the max and min ints to ensure it lies between them. Otherwise, overflow will obviously occur.

TESTING

All non-error outputs (upon input X) will be in the form

"Lexeme "X" Lexical token (constant, Y)"

where Y is the decimal equivalent of X.

INPUT	***EXPECTED Y***	***WHAT IT TESTS***	***TRANSITIONS***
1833h	6195	Hex values can be composed of inputs 1, 2 and finish with 4	0–>3 3–>5 5–>5 5–>5 5–>7
1800499	1800499	Recognizes non-leading 0's correctly	0–>3 3–>5 5–>5 5–>5 5–>5 5–>5 5–>5
00000007604	00000007604	Ensures leading zeros disregarded when calculating overflow	0–>1 1–>1 1–>1 1–>1 1–>1 1–>1 1–>1 1–>3 3–>3 3–>3 3–>3
-00000007604	-00000007604	Ensures sign in front of leading zeros does not affect overflow	0–>8 8–>2 2–>2 2–>2 2–>2 2–>2 2–>2 2–>2 2–>9 9–>9 Contd..

			9–>9 9–>9
8B0H	2224	Hex values can contain inputs 2,3 and 0	0–>5 5–>6 6–>6 6–>7
bh	11	That b isn't recognized as the octal indicator, even when there are no other numbers or letters in the hex value	0–>6 6–>7
aH	10	Hex values can be single letters	0–>6 6–>7
+0018	18	Positive sign works with leading zeros	0–>8 8–>2 2–>2 2–>9 9–>9
09bh	155	Leading zeros work with hex	0–>1 1–>5 5–>6 6–>7
0bh	11	Leading zeros followed only by letters	0–>1 1–>4 4–>7
0h	o	Zero followed by indicator still gives zero	0–>1 1–>7
2b0h	688	Input 1 can be followed by a b without being recognized as octal	0–>3 3–>4 4–>6 6–>7
4h	4	Hex composed of only input 1	0–>3 3–>7
5ah	90	Input 1 can be followed by input 5	0–>3 3–>6 6–>7
-10	-10	Sign (not followed by leading zero)	0–>8 8–>9 9–>9

2b	2	Oct	0–>3 3–>4
-0	0	Minus 0 gives 0	0–>8 8–>2
0	0	0 gives 0	0–>1

All non-error transitions and endings are covered by the above tests.
I also tested some values that I expected would give errors:

INPUT	***EXPECTED MESSAGE***	***WHAT IT TESTS***
123A56789H	Error, constant contains too many digits	Hex Overflow – too many digits
12345678910	Error, constant contains too many digits	Int Overflow – too many digits
2147483648	Error, constant cannot be represented as a 32-bit value.	Int Overflow - 11 digits and > max int
-2147483649	Error, constant cannot be represented as a 32-bit value.	Int Overflow - 11 digits and < min int
123456770027b	Error, constant contains too many digits	Oct Overflow – too many digits
42345671234b	Error, constant cannot be represented as a 32-bit value.	Oct Overflow – 11 digits with Most Significant Digit > 3
98b	Error, constant is not a vaild Hexadecimal, Octal or Integer value.	Invalid constant
7a451bc	Error, constant is not a vaild Hexadecimal, Octal or Integer value.	Invalid constant
34h2	Error, constant is not a vaild Hexadecimal, Octal or Integer value.	Invalid constant

All inputs gave the expected output.

CONCLUSION

Given that all of the above tests meet expectations and that I have followed the typical design process, I believe that my lexical analyzer fulfills the

criteria for this assignment. I also conclude that the program is clear and relatively concise. It is also easily adapted to include the conversion of even more inputs, states and bases.
In hindsight, I wish I had kept better records of the usages I explored while designing the system. I would also like to make the overflow checks cleaner and more readable – Although they are perfectly functional, I wish I had spent a little more time refining them. All in all however, I am happy with my attempt at designing the lexical analyzer specified in the assignment.