Lab 2 Report (13319368)

The second lab’s goal was to implement a lexical analyzer for processing a sequence of 32-bit octal, hexadecimal and signed integer constants described by the following regular expression:

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

Here is the specification of the lexical analyzer which accepts and processes all valid inputs: (the first row consisting of inputs, the first column of the states, and particular row/column combo as the transition from state to state on specific input:

Inputs: {Sign(‘-‘, ‘+’}, Octal Digit (0..7), Decimal Digit (8..9), Hexadecimal Digit (a..f, A..F), Octal Identifier(‘b’, ‘B’), Hexadecimal Identifier (‘h’, ‘H’), End Symbol}.

**States:**  1. Starting state

2. Octal digit seen

3. Octal identifier seen.

4. Decimal digit seen.

5. Hexadecimal digit seen.

6. Hexadecimal identifier seen.

7. Sign seen (can only appear as the first symbol, otherwise illegal).

8. Digit after sign seen.

**Accepting states**: {3, 4, 6, 8}.

**Starting state:** {1}.

**Transition table:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| State | Sign | Octal Digit | Decimal Digit | Hexadecimal Digit | Octal Identifier | Hex Identifier | End symbol |
| 1. Starting state | 7 | 2 | 4 | 5 |  |  |  |
| 2. Octal digit seen |  | 2 | 4 | 5 | 3 | 6 |  |
| 3. Octal identifier seen |  | 5 | 5 | 5 |  |  | “Process valid octal digit.” |
| 4. Decimal digit seen |  | 4 | 4 | 5 |  | 6 |  |
| 5. Hex digit seen |  | 5 | 5 | 4 |  | 6 |  |
| 6. Hex identifier seen |  |  |  |  |  |  | “Process valid hex digit.” |
| 7. Sign seen |  | 8 | 8 |  |  |  |  |
| 8. Digit after sign seen |  | 8 | 8 |  |  |  | “Process valid signed decimal digit” |

Note: all blank states lead to rejection.

**Processing:** (this can all be found in the code, expressed in a more condensed manner)

This is where I go out on a limb: It’s my belief that the actual processing is system-dependent, and that there is no one-fits-all algorithm to jot down here.

As it so happens, we got to use C to process this, and so using the char array in which the input string comes makes sense. If this were not the case, chars could be popped on as they are registered et cetera (say, if the chars arrived in a stream or something). As it stands, all processing is done once the number is known to be valid (i.e. when one of the “Process…” states is reached).

The chars are converter to numbers in the following way: Initialize result to 0. Begin with the first number(ie. 8 in 879). Convert this number to its numerical value (so 8 is 8, but a is 10). Adjust the number for position, which means multiply it by the base exponentiated to the number’s position (0-based indexing in regards to the number position, so position of 9 is 0). For example, in the decimal number 879, the value 8 will be multiplied by 10^2. Then add this value to the result. Repeat this for the rest of the numbers remaining (so result += 7\*10^1 + 9\*10^0).

The only caveat here is the possibility of positive or negative decimals. This is kept track of via a “negative” Boolean value (see code). If the sign == ‘-‘, then negative is “True”, and instead of adding the numbers to the result (i.e. result += 8\*10^2; result += 7\*10^1 and so on) the intermediate terms are subtracted (so it is not result -= 8\*10^2; result -=7\*10^1 and so on).

**Overflow handling:** (this can all be found in the code, expressed in a more condensed manner)

Finally, overflow is dealt with case by case. It’s simple for octal and hexadecimal values because: A they can only be positive, and B they conform to a nice pattern.

**Hexadecimal:**

Max hex 32 bit constant is 7FFFFFFF, which is 8 digits in length. Thus, the constant is too large if: the number is more than 8 digits long, or if the number is 8 digits long and the value of the first digit > 7. If neither of those conditions is true, the constant will be within bounds and no additional checking is required.(the second condition holds because all digits after the 7 can be F’s, the largest possible, and still be within bounds. There is no digit > F which could cause overflow).

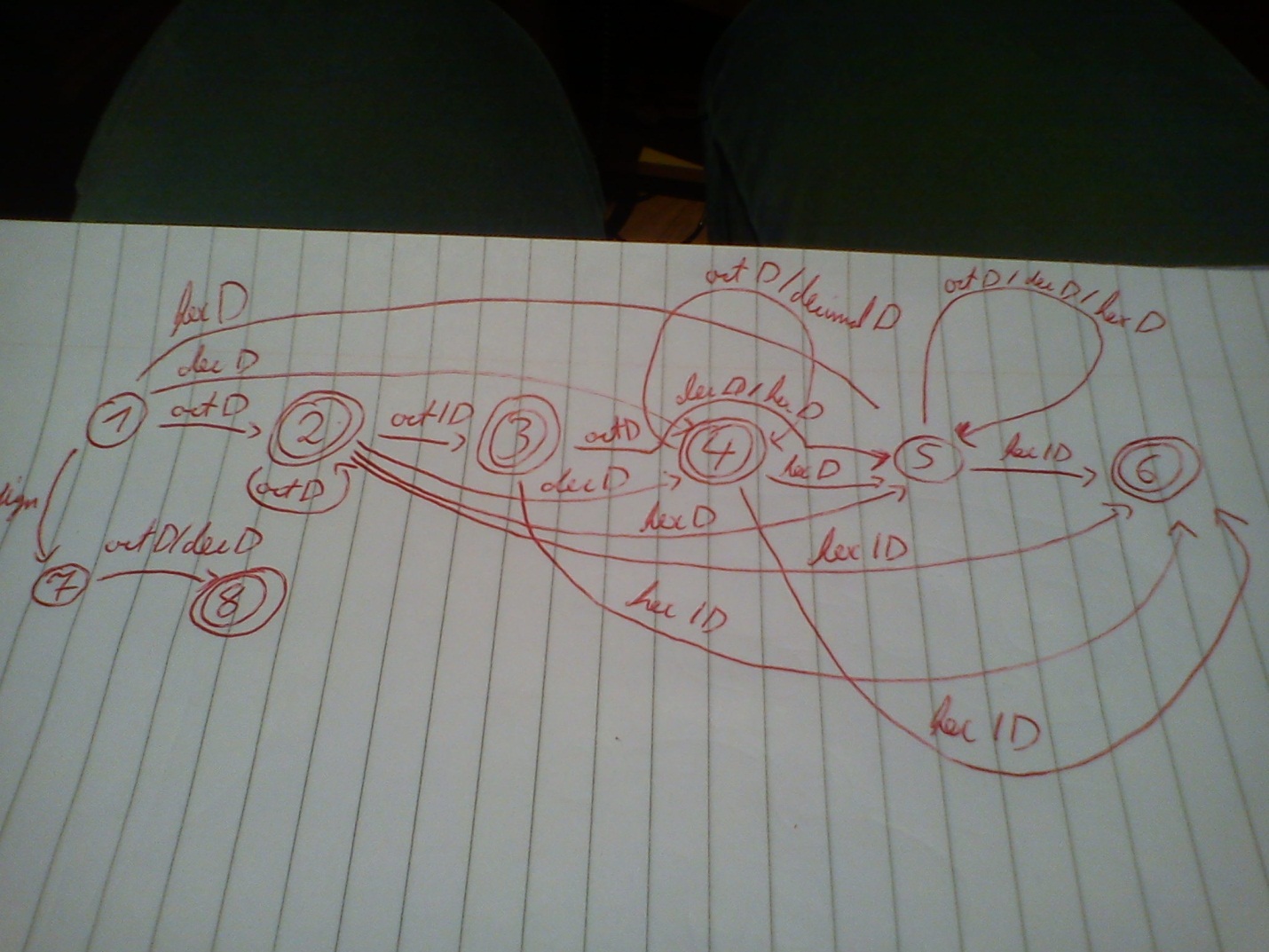
**Octal:**

The max oct 32 bit constant is 17777777777 is 11 digits long. So, the constant is too large if the number of digits is more than 11, or if the number of digits is 11 and the value of the first digit > 1. If neither of those conditions is true, then the constant will be within bounds and no additional checking is required. (the second condition holds because 7 is the max possible octal digit, and so, for example 07777777777 will not overflow, no need to check other digits past the first one.).

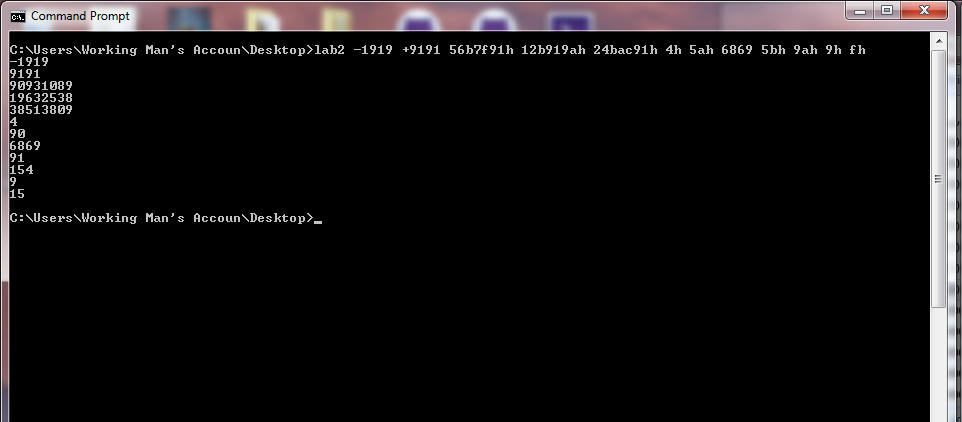
**Decimal:**

The overflow checking for the decimal is a little trickier. The number (2147483647 for positive constants, 2147483648 for negative values) conforms to no particular pattern. The solution then breaks down into two sections. For positive decimals, the most significant bit is monitored. If the MSB is ever 1, overflow has occurred. Converse for the negative. If the most significant bit of a negative number is ever 0, overflow has occurred (aside from when the number is 0).

Here is the state diagram used to create the debug sequences:



The sequences that cover all possibilities are: -1919, +9191, 56b7f91h, 12b919ah, 24bac91h, 4h, 5ah, 6869, 5bh, 9ah, 9h, fh. Here is a screen dump of the converted values:



All values are correctly converted (feel free to check). Overflow values have been tested for each case, and all work (feel free to also check).