CS3071 Lab 2, Lexical Analyser

Calvin Nolan, 13325852

October 2015

1 Introduction

In this lab we were tasked to design and implement a lexical analyser 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]+)

In this report I will walk through my process of designing this Lexical Analyser, starting with designing the analyser by finding all possible inputs and cases of error then drawing out a complete state table with the relevant actions and finally implementing my solution in C and the further optimizations that can be made in the transition from paper to code.

2 Designing the Analyser

A big design decision that had to be made before specifying any details was deciding what was the least amount of passes over the input value was possible and then designing accordingly. I came to the conclusion that two passes had to be made over the value. The first pass ensures the input is valid and notes out the specified base for use in the second pass. The second pass has two parts to it, the first calculates if overflow will occur only if the specified base is octal or hexadecimal and it can also decide if the value is positive or negative. The second part actually calculates the value while remaining in the 32bit restraint while also checking for overflow if the specified base is a signed integer.

Before diving head first into numbering and naming the different states possible in the analyser, I needed to think about any possible error cases that were not outlined obviously from the regular expression. Two distinct inputs that needed to be looked at separately in this system that can be cause for errors are

1. Leading zeroes.
2. Inputting the value b or B when it does not signify a base.

The reason for leading zeroes is that my solution will make use of the number of values input, the value of the first character and the base to calculate if overflow will occur, leading zeroes will throw this solution off so by counting them in our first pass and noting how many there are, we can account for them when calculating overflow. The reason for b or B when not a base is to ensure the base is affected wrongly since it is only acceptable in base 16 yet it indicate base 8. With these in mind we can begin to list off all possible states:

1. Starting State.
2. Sign input before a constant.
3. Leading zero input.
4. 0 - 7 input, currently satisfying base 8.
5. 0 - 9 input, currently satisfying base 10.
6. 0 - 9, a-f, A-F input, currently satisfying base 16.
7. Digit after a sign (and after leading zeroes).
8. Hexadecimal base indicated (hH).
9. Octal base indicated (bB).

With these 9 states in a structured state table with actions we can create a successful and efficient two pass lexical analyser.

3 State Table & Actions

A lot of the states are self quite self-explanatory when looking at the regular expression provided.

State 2 is when a sign is input as the first value and then calculates the leading zeroes (a special case if made here where it returns to state 2) and then only allows digits 0-9 to be input in state 7 as any other bases other than 10 infer an error.

State 3 is used to calculate the number of leading zeroes in the system.

State 4, 5 and 6 are used to indicate which base we currently satisfy, an input can move upwards (base 8 -¿ base 10 -¿ base 16) but if it moves downwards by it's base indicator then we have an error. State 4 can be accepted at the end of input since a signed integer has no base indicator at the end of it.

State 7 is used only when a sign has been input at the start of the value, as we can only accept 0-9 for a signed integer as per the regular expression.

State 8 is used for the Hexadecimal indicator, since hH is only accepted as an end of input. We can accept from this state to give us a base of 16.

State 9 is used for the Octal indicator, although bB can also be accepted as a value in a Hexadecimal value so we must allow for also moving up a base if it is not the end of input.

The actions that we must take upon a state transition can be decided after we have chosen what variables must be created and updated for our system to succeed. For the first pass the system needs to keep a note of:

- The number of leading zeroes - leadingZeroesCount.
- If a sign was input at the start - sign.
- What the first value that's not a leading zero is for calculating overflow and negativity in pass two - firstValue.
- What base is specified - base.
- How long the input value is - totalCount.

And in the second pass:

- The integer value of the input string - total.

The system increases leadingZeroesCount when transitioning into state 3 or an input of 0 in state 2. Sign is used only if state 2 is entered. firstValue is used when transitioning into state 4, 5, 6, 7 or 9 for the first time. Base is changed upon any transitions between 4, 5 and 6. TotalCount is increased upon every input that's not a sign, base indicator or end input. Total is updated with every value input past leading zeroes in the second pass.

State		Sign	1-7	8-9	a-f	hH	bB	End Input
1		2	3	4	5	6		
2		2	7	7				
3		3	4	5	6		6	
4		4	4	5	6	8	9	
5		5	5	5	6	8		Accept
6		6	6	6	6	8	6	
7		7	7	7				Accept
8								Accept
9		6	6	6	6	8		Accept

All blank entries lead to the error state.

Table 1: State Table for a Lexical Analyser

4 Programmed Solution

The programmed solution follows the exact same structure with the first pass, and two parts of the second pass.

The first pass checks that the input is valid and takes note of a few attributes for the second pass. Continue statements are used to move onto the next input when the system is satisfied with the executed actions for the current state.

```
// Check if the first character is a sign.
// State 2.
if((i == 0) && (characterI == '-')) {
    sign = -1;
    continue;
} else if ((i == 0) && (characterI == '+')) {
    sign = 1;
    continue;
}
```

The code used to count the leading zeroes uses a boolean leadingZeroes to know when the input is after a chain on leading zeroes.

```
// Check if the first value is a leading zero.
// State 3.
if((totalCount == 0) && characterI == '0') {
    leadingZeroes = 1;
    leadingZeroesCount++;
    totalCount++;
    continue;
}
// Check if this character is a leading zero.
// State 3.
else if(leadingZeroes == 1 && characterI == '0') {
    leadingZeroesCount++;
    totalCount++;
    continue;
}
// Mark the end of all leading zeroes.
// Transition out of state 3.
else if(firstValue == ' '){
    leadingZeroes = 0;
    firstValue = characterI;
}
```

The last value has to be looked at seperately to actually decide on the base of the value and whether it's valid for what has been previously input.

```
// Check the base value
if(i == (strlen(stringValue)-1)) {
    // State 8.
    if(characterI == 'h' || characterI == 'H') {
        base = 16;
    }
    // State 7.
    else if((characterI == 'b' || characterI == 'B')
                            && base != 16 && base != 10) {
        base = 8;
    }
    // State 5.
    else if((characterI >= 48 && characterI <= 57) && base != 16) {
        base = 10;
        totalCount++;
    } else {
        error = 1;
        printf("Error, incompatible base! \n");
    }

    if(sign != 0 && base != 10) {
        error = 1;
        printf("Error, You can only specify a sign for base 10 values!\n");
    }

    continue;
}
```

To ensure that the base at the end is valid, the system needs to take note of every input and calculate which base is currently being satisfied.

```
// Check that the value is valid and which base we are currently satisfying.
if(leadingZeroes == 0) {
    totalCount++;
    // State 4.
    if((characterI >= 48 && characterI <= 55) && base != 10 && base != 16){
        base = 8;
    }
    // State 5.
    else if((characterI >= 48 && characterI <= 57) && base != 16) {
        base = 10;
    }
    // State 6.
    else if((characterI >= 48 && characterI <= 57)
            || (characterI >= 65 && characterI <= 70)
            || (characterI >= 97 && characterI <= 102)){
        base = 16;
    } else {
        error = 1;
        printf("Error, invalid input '%c'! \n", characterI);
    }
}
```

For the second pass we can separate the first part which will look at whether overflow will occur and if the value is negative.

```
//Second Pass.
//Part I.
//Checks if the value will be negative and if we will have overflow for base 8 o
if(base == 8) {
    if(count == 11 && firstValue > '3') {
        printf("Error, Overflow in base 8! \n");
        return;
    } else if(count > 11) {
        printf("Error, Overflow in base 8! \n");
        return;
    }

    if(count == 11 && firstValue > '1') {
        negative = 1;
    }
} else if(base == 16) {
    if(count > 8) {
        printf("Error, Overflow in base 16! \n");
        return;
    }

    if(count == 8 && firstValue > '7') {
        negative = 1;
    }

} else if(base == 10) {
    if(count == 10 && firstValue > '2') {
        printf("Error, Overflow in base 10! \n");
        return;
    } else if(count > 10) {
        printf("Error, Overflow in base 10! \n");
        return;
    }

    if(count == 10 && firstValue == '2') {
        overflowCheck = 1;
    }
}
```

The last part of the program, the second pass part II, just accumulates the integer value depending on the base and input.

```
for(i = leadingZeroesCount; i < totalCount; i++) {
    char characterI = stringValue[i];
    if(base == 8) {
        int32_t currentChar = characterI - 48;
        if(negative == 1) {
            total = (total * 8) + (currentChar - 7);
        } else {
            total = (total * 8) + currentChar;
        }
    } else if(base == 16) {
        int32_t currentChar = atoi_base_16(characterI);
        if(negative == 1) {
            total = (total * 16) + (currentChar - 15);
        } else {
            total = (total * 16) + currentChar;
        }
    } else if(base == 10) {
        int32_t currentChar = characterI - 48;
        if(overflowCheck == 1){
            switch (i - leadingZeroesCount + 1) {
                case 1:
                    if(currentChar > 2) {
                        printf("Error, Overflow in base 10! \n");
                        return;
                    } else if(currentChar < 2) {
                        overflowCheck = 0;
                    }
                    break;
                // Cases 2-8 work the exact same for the values of 2147483647
                case 10:
                    if(currentChar > 7) {
                        printf("Error, Overflow in base 10! \n");
                        return;
                    } else if(currentChar < 7) {
                        overflowCheck = 0;
                    }
                    break;
            }
        }
        total = (total * 10) + currentChar;
    }
}
```

```
if(sign == -1) {
    total = total * -1;
}

if(negative == 1) {
    total--;
}
```

The final part of the program outputs the total calculated as specified in the the assignment document.

5 Testing

Test Case	Input	Expected Output	Actual Output
Valid Decimal	4096	Signed Integer Constant, 4096	Signed Integer Constant, 4096
+ Valid Decimal	+4096	Signed Integer Constant, 4096	Signed Integer Constant, 4096
- Valid Decimal	-4096	Signed Integer Constant, -4096	Signed Integer Constant, -4096
Invalid Decimal	4096a	Error, incompatible base!	Error, incompatible base!
Valid Octal	4321000b	Octal Constant, 1155584	Octal Constant, 1155584
Negative Octal	27777777777b	Octal Constant, -1073741825	Octal Constant, -1073741825
Invalid Octal	1118b	Error, incompatible base!	Error, incompatible base!
Invalid Octal	1118ABCb	Error, incompatible base!	Error, incompatible base!
Valid Hex	4321ABCh	Hex Constant, 70392508	Hex Constant, 70392508
Negative Hex	8FFFFFFFh	Hex Constant, -1879048193	Hex Constant, -1879048193
Invalid Hex	4321ABCGh	Error, invalid input 'G'!	Error, invalid input 'G'!
+ Hex	+123h	Error	Error
Leading Zeroes	000123	Signed Integer Constant, 123	Signed Integer Constant, 123
- Leading Zeroes	-000123	Signed Integer Constant, -123	Signed Integer Constant, -123
Only Zeroes	0	Signed Integer Constant, 0	Signed Integer Constant, 0
- Only Zeroes	-0	Signed Integer Constant, 0	Signed Integer Constant, 0

6 Conclusion

If I were to start this project again I would take a more structured approach to the coding, I think a function that takes in the current state and input and gives back the next state would be easier to read than the current implementation. Although I believe my implementation saved more space and since it's split into two distinct pieces there are some sections of code that never use up resources in the piece that it doesn't belong to.