CPEG621

Lab 1

Michael Hutts

A Calculator Compiler Front-end

Objectives

The primary goal of this lab was to better comprehend lex/flex and yacc/bison by developing two projects using these tools. This lab was broken up into two tasks, the first being to developer a infix calculator, and the second task being to use what was developed in task 1 to create an infix-to-postfix converter. For task 1 was able to create a fully working infix calculator, with full variable support. For task 2, I was able to making a working and consistent converter, but with some caveats due to limitations I could not overcome.

The process for task 1 was relatively straightforward. Calc.l and calc.y provided a helpful blueprint, and Tom Nieman's tutorial provided a better understanding to complete the task. Once the overall grammar and flow was setup, the operations simply fell into place. Task 2 turned out to be a much more complicated issue. Creating a prefix calculator with what we had was extremely simple, and I in-fact did this initially due to me having misread the instructions initially. Creating a prefix converter that followed all the principles described in the lab on the other hand turned out far more complicated. A stack-based approach provided the core solution, but for 100% accuracy, high levels of manual parsing for different cases was required that turned out complex and unmanageable for the timeline of this lab.

Strategy

Task 1

Infix.l

After comprehending the existing code provided in calc.l and infix.l, the next step was expand infix.l to include all the operations we'd need. This was a somewhat simple process, requiring only repeating existing sections, with simple modifications for the new characters and symbols. The only part that needed any special solution was handling variables, but this was as simple as looking for [a-z], and converting these char characters to values [0-25] to fit inside an array that we'd use later.

Infix.y

Modifying infix.y was the core challenge of task 1. Calc.y gave a blueprint, but as the code expanded, following the exact process of the existing code proved difficult. Using the tutorial suggested in the lab manual, I modified the code structure somewhat, breaking nearly everything into an expression block, that handled all operations with numbers, saving only a base “cal” to handle output and some formatting. Most of infix.y was very simple to construct, going in a format like so:

```
| exp '+' exp { $$ = $1 + $3; }
| exp '-' exp { $$ = $1 - $3; }
| exp '*' exp { $$ = $1 * $3; }
| exp '/' exp { $$ = $1 / $3; }
| exp POW exp { $$ = (int)(pow($1,$3)); }
| exp INC { $$ = $1 + 1; }
| exp DEC { $$ = $1 - 1; }
| '(' exp ')' { $$ = $2; }
```

The exception to this was variables, which required some additional logic beyond the single line required for other operations. A single definition like “x=4” required only a single line like the others, but to handle back-to-back variable assignment, some additional logic was required.

```
| VAR { $$ = sym[$1]; }
| VAR '=' exp {
    if (equals_flag) {
        sym[$1] = sym[$3];
    }
    else {
        sym[$1] = $3;
    }
    equals_flag=1;
}
```

As you can see, we use a simple flag to track if we’re preceding another equal’s sign/variable combination. Without this, I ran into the problem of the left-most variable becoming the integer equivalent value of the character representing that variable. With this flag in-place, a fully working infix calculator was developed.

Task 2

This task proved to be a far greater struggle than task 1. As described in the abstract, I initially misunderstood this part (my own mistake of mixing the slide instructions with a cursory glance at the lab) as me needing to implement a prefix calculator. This turned out to be an easy task that worked well. I’ve included this with my submission since I already did the work, but it’s not relevant to the real task 2, which was to convert the infix input to prefix. I was confident going into this, but it turned out much more complicated than I had anticipated due to some quirks.

I had done some work with polish notation before, and given we already had broken our input into its fundamental groups, I imagined this was as simple as reordering everything before pushing it down the stack. Unfortunately, this did not prove to be fruitful. Formatting issues and inconsistencies in the requirements for the string required for the following section of the input proved to be something that could not simply be pushed through mindlessly, careful flagging and intelligent logic was necessary. I was able to complete the most fundamental means of this. Basically, converting any single operation to its prefix form, but simultaneously respecting these examples made it extremely difficult to stick to one format:

> "3*(1+2)" => "* 3 + 1 2", NOT " * + 3 1 2"
>
> *"4+5+6" => "+ + 4 5 6" NOT "+ 4 + 5 6"*
>
> "7+8*9" => "+ 7 * 8 9"

I flip-flopped between implementations which prioritized leading with operators, and prioritizing explicit orders of operations, but could not find a way to make both work harmoniously. The best I could do was to choose the most parseable form to myself, even if it did not respect all the rules laid out.

With this approach locked in, there was still one other problem. The way the infix parser works, you'll end up with repeating numbers, as expressions are supposed to "pop down" the stack and simplify into a single numerical answer, but choosing which numbers to keep and which to throw out proved difficult. A compromise I found was to track when we'd "jump" in order of operation, that way we'd always pick the right value regardless of if we started out of order or not. This resolved choosing which side to keep of an operation (though some bugs with this seem to exist still).

Results

Task 1

With the simple and elegant logic implemented, all required operations were able to be successfully completed in any context.

Addition, subtraction, multiplication, and division

```
1+2-3*4/5
=1
```

Unary INC/DEC operator “++” and “--“

```
1++
=2
1--
=0
1-10++
=-10
1-10--
=-8
```

Power operator

```
=16
5+2**16
=65541
```

Support parentheses

```
3*(5-3)
=6
3*(5-(1+(3-4)))
=15
```

Support variables / Assignment operator

```
x=10
=10
x=x*20
=200
a=b=4
=4
a
=4
b
=4
```

Task 2

For this task, support is much more flawed and partial. Regardless, the general structure and solution is sound and works accordingly with consistency:

```
1+2
+ 1 2
3*4
* 3 4
5**6
** 5 6
(7+8)*9
* 9 + 7 8
10++
++ 10
x=11
= x 11
```

The incorrect behavior, as I described earlier, is in how it handles repeated operations. According to the lab manual, instructions with identical orders of operation should be combined, but I was unable to get this methodology working without breaking other things in the process

```
1+2+3
+ 3 + 1 2
1+2-3
- 3 + 1 2
4*5*6
* 6 * 4 5
7*8/9
/ 9 * 7 8
```

Conclusion

In conclusion, throughout this lab I was able to gain a much stronger understanding of lex and yacc that I feel I can successfully carry onto future labs. Although there was some stumbling when it came to the process of completing task 2, I feel like this was more a struggle with coding a complex problem in C, not with lex or yacc. I have a relatively strong background when it comes to parsing with regex tools, but this was an interesting problem to tackle that was very different in many ways. Despite having experience with C, my strongest weakness felt like it was with C, not lex or yacc. Multiple roadblocks with task 2 felt like they were purely to do with the complexity of doing string parsing in C. Parsing it with lex was easy, and building the rules with yacc was intuitive after some reading, but with everything tokenized, the real challenge turned out to be the pure C. With this in mind, I'm exciting for the future labs and how I might expand on and improve what I've done. At the moment, task 2 feels like a dead-end, but I can only hope after my submission I can uncover a missing piece to my puzzle that overcomes the hurdles I could not tackle all at once.