LAB 10 – Bison and Flex

John Dempsey
COMP-232 Programming Languages
California State University, Channel Islands
October 29, 2025
Hard Due Date: November 7, 2025

In this lab, we will be using **flex** to build a lexer via regular expressions and **bison** to parse the sequence of tokens output by said lexer.

The instructions for LAB 10 are found below and files are found on comp232.com in /home/LAB10.

Before you start, you will want to install flex and bison:

- Mac: `brew install flex bison`
- WSL: in the Ubuntu console, `sudo apt-get install -y flex bison`

`sudo apt-get install -y flex bison` will work on Linux distributions in general, if you're not using one of the two above.

Part 1: Lexing with Flex

We will cover a minimal introduction to the use of flex in this lab. The LEX & YACC Tutorial will provide some more detail, and you will likely want to refer to it throughout the lab to fill in the gaps, and particularly to explore the capabilities of flex's regular expressions.

If you are not familiar with regular expressions, or it's been a while and you're rusty, you may want to watch this video first.

Consider this grammar:

```
<program> ::= <statement> | <statement> <program>
<statement> ::= <assignStmt> | <ifStmt> | <whileStmt> | <repeatStmt> | <printStmt>

<assignStmt> ::= <ident> = <expr> ;
<ifStmt> ::= if ( <expr> ) <statement>
<whileStmt> ::= while ( <expr> ) <statement>
<repeatStmt> ::= repeat ( <expr> ) <statement>
<printStmt> ::= print <expr> ;
```

```
<expr> ::= <term> | <term> <addop> <expr>
<term> ::= <factor> | <factor> <multop> <term>
<factor> ::= <ident> | <number> | <addop> <factor> | ( <expr> )

<number> ::= <int> | <float>
<int> ::= <digit> | <int> <digit>
<float> ::= <digit>. | <digit> <float> | <float> <digit>

<ident> ::= <letter> | <ident> <letter> | <ident> <digit>

<addop> ::= + | -
<multop> ::= * | / | %

<digit> ::= 0-9
<letter> ::= a-z | A-Z | _ | $
```

Enumerate all of the token types in this grammar in the `TOKEN` enum in `flex.h`. In `flex.c`, fill out the `tokenTypeStrings` array in the same order as the `TOKEN` enum. This way, the elements of the `TOKEN` enum can be used to index the `tokenTypeStrings` array and get the corresponding string for printing purposes. This works because enum elements are really just named integers, starting at `0` by default and progressing upward in the order they appear in the enum.

Once you've enumerated all of the `TOKEN` types, you're ready to begin working on the `flex.l`. This file is essentially a configuration file, which flex will use to generate a scanner.

`flex.l` is divided into 3 sections. These sections are separated by lines containing `%%`. The first section is for **definitions**, the second for **rules**, and the third for **subroutines**. Your task is to complete the first and second sections (the third section will be left empty).

The Definitions Section

The first section can, in theory, be left empty for what we are doing today, but it should not be. In this section, you can define shortcuts for regular expressions.

Consider these lines in `flex.l`:

```
letter          [a-z]
digit           [0-9]
```

The two lines above define `letter` to be a shorthand for the regular expression `[a-z]` and `digit` one for `[0-9]`.

Next, consider these lines:

```
float           {digit}+\.{digit}*
ident           {letter}+
```

The two lines above define `float` to be one or more digits, followed by a period and then 0 or more digits, and define `ident` to be 1 or more letters.

Note that when the `digit` and `letter` definitions were used in the `float` and `ident` definitions, they were encased in curly braces `{}`; this is how definitions are referenced. The `letter` and`ident` definitions do not match the grammar, and will therefore need to be rewritten. Moreover, regular expressions cannot be recursively defined in `flex` (or in any regular expression implementation that I've seen), so you will have to solve the `ident` production's recurrence to determine what sorts of strings an `ident` can be made out of.

The Rules Section

The second section (after the first `%%`) is for tokenization rules. Each line consists of a regular expression (or a definition from the first section) followed by a block (written in C) specifying what action is to be taken when a string is encountered which matches said regular expression. The goal in each block is to return the correct token type (and do anything else that needs doing, but for this lab we will just be returning token types). `flex.h` has been included at the top of the definitions section, so the blocks of C code in the rules section can reference anything accessible to `flex.h`. This is necessary; the token types being returned are defined in `flex.h`.

Consider these lines provided in the rules section:

```
if          {return IF_TOKEN;}
{float}     {return FLOAT_TOKEN;}
{ident}     {return IDENT_TOKEN;}
```

These lines mean, respectively:

- When the string "if" is encountered, return an `IF_TOKEN`.
- When a string matching the `float` definition is encountered, return a `FLOAT_TOKEN`.
- When a string matching the `ident` definition is encountered, return an `IDENT_TOKEN`.

Some of the strings which need to be tokenized (such as parenthesis) have meta-meaning in regular expressions, and therefore will need to either be escaped or put in quotes in order to function as literal characters in a regular expression. For instance:

```
This does not work:

(                       {return LPAREN_TOKEN;}

Either of these work:

\(                      {return LPAREN_TOKEN;}
"("                     {return LPAREN_TOKEN;}
```

The order in which tokenization rules appear matters. For example, any keyword (print, repeat, etc) will also match the regular expression for an identifier. As such, all of the keyword tokenizations must happen *before* the identifier tokenization, so keywords match their patterns and return the correct token type before they are ever compared to the `ident` definition.

Three more rules have been provided at the bottom of the rules section (and they should be the last three in the rules section when you are done):

```
<<EOF>>     {return EOF_TOKEN;}
[ \n\r\t]   ; //skip whitespace
.           {printf("ERROR: invalid character: >>%s<<\n", yytext);}
```

These rules are to tokenize the end of file, skip whitespace, and catch any invalid characters, respectively.

The Subroutines Section

We'll talk about this section briefly in a future lab, but for now we're just going to leave it blank. Feel free to read up on it in the provided PDF!

Output

When you have completed the `TOKEN` enum in `flex.h`, the `tokenTypeStrings` array in `flex.c`, and definitions and rules sections in `flex.l`, you are ready to test! There is a provided sample input, `input.txt`, with the following contents:

```
while (0.4) abc_1_2 = agd + 1;
if (condition) print ("hello") ;
```

For this sample input, the output should be:

```
{<while> "while"}
{<lparen> "("}
{<float> "0.4"}
{<rparen> ")"}
{<ident> "abc_1_2"}
{<assign> "="}
{<ident> "agd"}
{<addop> "+"}
{<int> "1"}
{<semicolon> ";"}
{<if> "if"}
{<lparen> "("}
{<ident> "condition"}
{<rparen> ")"}
{<print> "print"}
{<lparen> "("}
ERROR: invalid character: >>"<<
{<ident> "hello"}
ERROR: invalid character: >>"<<
{<rparen> ")"}
{<semicolon> ";"}
{<eof> ""}

Process finished with exit code 0
```

You do not need to edit the main to match this output; the only change you need to make to `flex.c` is to fill out the `tokenTypeStrings` array.

Each token is printed with both its type and the string value that was tokenized. In some cases this is redundant, and that's fine; we will work on more complex tokenization rules which process the string value or simply ignore it in a later lab.

When you run, a lexer called `lexer.c` is generated in the `src/lexer` directory; your `flex.l` file served as a configuration file specifying how this lexer should function.

Submission Checklist

Your submission should:

- Complete the `TOKEN` enum in `flex.h`.
- Complete the `tokenTypeStrings` array in `flex.c` such that the token types are named in the same order they are declared in the `TOKEN` enum.
- Improve `input.txt` to rigorously test your scanner for the provided grammar.

Part 2 - Lexing and Parsing with Flex and Bison

Bison is a tool which allows for the generation of a parser from a configuration file. It is a free equivalent to yacc, a proprietary tool for the same purpose; the bison configuration file's extension will be `.y`, for "yacc".

As with flex in the last lab, we will cover a minimal introduction to bison here. You will want to refer to LEX & YACC Tutorial for a more detailed overview throughout the lab.

In this lab, you will create a lexer using flex and a parser using bison in order to evaluate expressions in **Cambridge-Polish Notation (CPN)**. The results of evaluation will be numeric, and we will house them in the `NUMBER` struct provided in `calc.h`; read through it.

Cambridge-Polish Notation

Cambridge-Polish Notation (CPN) is a notation which lists functions and their operands enclosed together in parenthesis, with the operands following the operator. For example, the arithmetic expression `1+2` in CPN would be `(+ 1 2)` or `(add 1 2)` (our implementation will use the latter representation).

CPN expressions are **nestable**. That is, a CPN expression can be used as one of the operands in another CPN expression. For example, the expression `(sub 3 (add 1 2))` is valid and would evaluate to `0` (note here that `sub` is subtraction).
You will be making a lexer and parser for expressions in CPN from the following grammar:

```
<program> ::= <expr> EOL | QUIT
<expr> ::= <number> | <f_expr>
<f_expr> ::= ( FUNC <expr> ) | ( FUNC <expr> <expr> )
<number> ::= INT | FLOAT
```

The grammar is incomplete; it does not include definitions for the tokens `FUNC`, `INT`, `FLOAT`, `QUIT`, and `EOL`. These will be defined as follows:

- `FUNC` : One of the following strings (function names):
    - "neg"
    - "abs"
    - "exp"

    - "log"
    - "sqrt"
    - "add"
    - "sub"
    - "mult"
    - "div"
    - "rem"
- `INT`: an optional `+` or `-` sign, followed by one or more digits.
- `FLOAT`: an optional `+` or `-`, one or more digits, a period, and 0 or more trailing digits.
- `QUIT`: the string "quit".
- `EOL`: the newline character `\n`

Lexing with Flex and Bison

Defining Grammar Elements with Bison

The task of tokenization is a bit more complex than it was in the previous part, because the lexer will interact with the parser.

Open `calc.y`, the yacc file for this project, and `calc.l`, the lex file for this project. Like the `.l` file, the `.y` file is divided into three sections by lines containing `%%`; these three sections are for **definitions**, **rules** and **subroutines** respectively. Before the lexer can be filled out in `calc.l`, it is necessary to fill out the **definitions** section of `calc.y`; this section will enumerate the tokens and **types** (the composite syntax tree elements, i.e. those "above the token level"). It will also specify what data types will be used to house the data for said tokens and types.

`calc.y`'s definitions section has the following contents:

```
%{
    #include "calc.h"
    #define ylog(r, p) {fprintf(flex_bison_log_file, "BISON: %s ::= %s \n", #r, #p);
fflush(flex_bison_log_file);}
%}

%union
{
    double dval;
    struct number *nval;
}
```

```
%token <dval> INT FLOAT
%token EOL

%type <nval> number
```

We will cover a brief description of what these lines mean; more insight can be found here (pdf here).

Let's start with the first part, which isn't really parser-related:

```
%{
    #include "calc.h"
    #define ylog(r, p) {printf(flex_bison_log_file, "BISON: %s ::= %s \n", #r, #p);}
%}
```

The two lines above are read directly into the parser that bison generates based on the contents of the yacc file. We already know what `#include "calc.h"` does; the other line is a preprocessor definition that creates a macro called `ylog`, which we'll use to log which productions are used during parsing, for debugging purposes. The outputs from this `ylog` function (and it's lexer equivalent, `llog`) will be in the `logs` folder, in a file called `flex_bison_log`.

Next, we get into the actual parsing configurations:

```
%union
{
    double dval;
    struct number *nval;
}
```

The `%union` command in bison specifies the data types that semantic values (tokens and types) have. This particular `%union` denotes that the tokens and types in our grammar will have their data stored in data types `double` and `NUMBER *` (`struct number` and `NUMBER` are identical, check the typedef in `calc.h`).

This union is stored in a variable called `yylval` for each individual token. If a `FLOAT` token has data is stored in `double` form, that data could be accessed through `yylval.dval`, because `dval` is the name we've given to the `double` member of the union.

Then, we have the declarations for tokens:

```
%token <dval> INT FLOAT
%token EOL
```

The following line, `%token <dval> INT FLOAT` creates two types of tokens, `INT` and `FLOAT`, and specifies that their values will be stored in `yylval.dval` (so, as type `double`). This is not a mistake; we'll be storing the numeric values of both `INTs` and `FLOATs` in `double` form, for the sake of simplicity in performing calculations later on.

Next, `%token EOL` creates one more type of token. It does not specify an enum element in which to store data, because there is no additional data necessary for an `EOL` token; the fact that it is an `EOL` ("end of line") token is all that is needed for its use as a delimiter. Finally, there is a definition for an element of the grammar which is above the token level:

```
%type <nval> number
```

This line declares the `number` type, and specifies that the data for any given `number` will be stored `yylval.nval`, which we can see in the union is a `NUMBER *`.
The following tokens are missing; you'll need to add them:

- `FUNC`
- `LPAREN`
- `RPAREN`
- `QUIT`

Multiple tokens can be included on the same line; the missing tokens which don't require any additional data other than their type can simply be added in with the `EOL`, just like `INT` and `FLOAT` were listed on the same line because they both store `double` values. Add `LPAREN` and `RPAREN` to the data-less tokens, alongside `EOL`.
The `FUNC` token, however, will require some additional data; we need to know which function it is. You might be tempted to add a `char *` to the `%union`, to store the function names. While this would work, it would be wasteful; we would need to allocate space for the function name, copy into it, free it up later... it would be a hassle.

There is already an enum listing all of the functions in `calc.h`, called `FUNC_TYPE`. Conveniently, a function called `resolveFunc`, which takes a function name in string form as an input and outputs the corresponding `FUNC_TYPE`, is declared in `calc.h` and defined in `calc.c`. The lexer can simply call `resolveFunc` on `yytext` (the buffer used to temporarily store tokens' string values while they are being processed) to find the correct `FUNC_TYPE` value for each `FUNC` token. (You don't need to this yet, we will do it in the next section).

From here, it seems pertinent to add `FUNC_TYPE fval;` to the `%union`, and declare function tokens with a line like `%token <fval> FUNC`. Unfortunately, our typedefs from `calc.h` cannot be accessed `calc.y` (which is why we use `struct number *nval` instead of `NUMBER *nval`). Recall that enum elements are just named integers! Add `int fval;` to the `%union` instead, and the token declaration `%token <fval> FUNC` will do the trick. This way, our `FUNC` tokens will store an integer value, which we will ensure is one of the `FUNC_TYPE` elements.

You may notice that we haven't declared the `expr` or `f_expr` syntax tree elements yet; these, like `number`, are above the token level and will be `%type`s. We will come back to them when we're done lexing.

Lexing with Flex

Now that all types of tokens have been declared, we can move on to lexing.

At the top of `calc.l`, you can see the following, which aren't really lexer-related:

```
%option noyywrap

%{
    #include "calc.h"
    #define llog(token) {fprintf(flex_bison_log_file, "LEX: %s \"%s\"\n", #token, 
yytext);fflush(flex_bison_log_file);}
%}
```

The `%option noyywrap` specifies that there is only one input file, as opposed to a sequence of input files. That is an oversimplification, but you can comfortably ignore it and nothing will break, or research it further if you're curious.
Then, the two lines in braces are nearly identical to those at the start of `calc.y`; they include `calc.h`, and define a macro that we will use for logging lexer actions, for debugging purposes.

Next, we can move on to the meat of the definitions section; part of it has been done for you:

```
digit       [0-9]
int         [+-]?{digit}+
```

You will need to complete this definitions section, just like in the previous part of this lab. We won't discuss it further here, but you can refer to the previous part for review.

Then, you'll need to complete the **rules** section, denoting what to do when each type of token is encountered. This section will be a little more complex than it was in the last part, as this time we're interacting with the configurations made in the `.y` file. Rules for `INT` and `EOL` tokens have been included, as have rules to skip whitespace and warn of invalid characters.

Let's look first at the `EOL` rule:

```
[\n] {
    llog(EOL);
    return EOL;
}
```

This rule specifies that when a newline character is encounters, a call to the `llog` function should be made, specifying that the `EOL` tokenization process has started, and then an `EOL` token should be returned. Because `EOL` tokens don't require any additional data, nothing else needs to be done here.

Next, consider the `INT` rule:

```
{int} {
    llog(INT);
    yylval.dval = strtod(yytext, NULL);
    return INT;
}
```

This rule specifies what should be done when a sequence of characters matching the previously defined `{int}` definition is encountered. It does the following

- logs that the `INT` tokenization has started.
- converts the matched string value `yytext` to `double` with with `strtod`
- stores the `double` value in `yylval.dval`; look back at the `%union` in `calc.y` to confirm that `dval` is the identifier we've given to `double` values for tokens.
- returns an `INT` token (to which we've just assigned a `double` value).

You'll need to complete the definitions and rules sections of `calc.l` to complete the lexer.

Parsing with Bison

Now that the lexer configurations are set up, we can start on the parser configurations. The first order of business is to add the missing `%type`s. Specifically, we are missing the grammar elements `expr` and `f_expr`. In order to decide what data type we should store these in, it is necessary to clarify that we will not be building the entire syntax tree this time; we will instead be evaluating it from the bottom up, as it is built. Thus, when we "parse" an `expr` or `f_expr`, we won't be connecting it to its children as designated in the grammar; we will instead be evaluating, i.e. using the children's values to determine the value of the parent. TLDR: `expr` and `f_expr` will be `NUMBER*`s, just like the `number` type.

Add `expr` and `f_expr` to the line declaring the `number` type to complete the definitions section of the `.y` file.

```
%type <nval> number expr f_expr
```

Much like the flex **rules** section specifies rules to translate sequences of characters into tokens, the bison **rules** section specifies rules to process sequences of grammar elements (tokens and types) using the productions in the grammar (encoded into the y file).

These rules very closely match the grammar itself. Some rules have been provided; you can see the `program` rules below:

```
program:
    expr EOL {
        ylog(program, expr EOL);
        printNumber(stdout, $1);
    }
    | QUIT {
        ylog(program, QUIT);
        exit(0);
    };
```

The rules above are the embodiment of this set of productions:

```
<program> ::= <expr> EOL | QUIT
```

Here, the `program` is the grammar element being produced, so it must be assigned. The `program` being produced is refered to with the shorthand `$$`. Furthermore it is being produced as a product of a sequence of grammar elements in the form `expr EOL`, whose values are referenced with `$1` and `$2` respectively (and if there were a third element comprising the `program`, its value would be referenced with `$3`, and so on...).

We know that the data for an `expr` is stored in a `NUMBER *` becasue that is how the `expr` type was declared in the definitions section. Note that there **is not a type definition** for a `program`. The `program` type serves as an entry point. We're building a CPN calculator, so in this case we just want to print the result of the `expr` comprising the program.

The block of C code contained within this rule specifies what should action should be taken when a `program` is produced from an `expr` followed by an `EOL` token. In this application, we make a call to `printNumber` (declared in `calc.h` and defined in `calc.c`) and pass in the `NUMBER *` value of the `expr`, so whenever we enter a valid expression the evaluated result of that expression will be printed. Recall: `$1` is the `expr` from the `expr EOL` comprising the `program`, so we pass it into `printNumber` as `$1`.

We also make a call to `ylog` which, as discussed earlier, simply prints to the `flex_bison_log` file in the `logs` directory, so we can check which rules were used and in what order for debugging purposes.

The rules for `expr` creation are also provided, and they serve as a better model for the the ones you'll need to add to complete the grammar:

```
expr:
    number {
        ylog(expr, number);
        $$ = $1;
    }
    | f_expr {
        ylog(expr, f_expr);
        $$ = $1;
    };
```

These rules above match productions in the grammar:

```
<expr> ::= <number> | <f_expr>
```

As seen above, when a grammar element can be produced from several different sequences of elements, these options are separated with | (semantically, "or"). An `expr` differs from a `program` in a key way: an `expr` has a value. In the `expr ::= number` production, the line `$$ = $1;` assigns the `NUMBER *` value associated with the `number` element to the `expr` element being produced; in other words, if an `expr` is comprised of just a single `number`, then the `expr`'s value is that of the `number`. Recall, `$$` refers to the value of the element being produced (in this case the `expr`), and `$n` refers to the `n`'th element comprising the production (that is, the `n`'th element being **reduced**).

You must complete this rules section, by filling out rules for the remaining productions:

```
<f_expr> ::= ( FUNC <expr> ) | ( FUNC <expr> <expr> )
<number> ::= INT | FLOAT
```

An `f_expr` consists of parenthesis, a `FUNC` token and 1-2 `expr`s whose values serve as the operands for the specified function. A function called `calc` is declared in `calc.h` and defined in `calc.c`:

```
NUMBER *calc(FUNC_TYPE func, NUMBER *op1, NUMBER *op2);
```

This `calc` function takes as arguments the enum element corresponding to the function being performed and the value(s) of its operand(s) (in `NUMBER *` form). Its task is to calculate and return the result. We will cover **how** it should do so later on.

For now, assume that the `calc` function works (because you'll make it work later). Use it in the `f_expr` productions to get the value of the `f_expr` being produced by passing in the function specifier (i.e. the value of the `FUNC` token) and the value(s) of the operand(s) (i.e. the value(s) of the `expr`(s)).

For single-operand function calls (those using the production `f_expr ::= ( FUNC expr )`), `NULL` should be passed into the `calc` call in place of a second operand.
In the rules to produce `number` elements from `INT` and `FLOAT` tokens,
the `createNumber` function should be called. It is declared in `calc.h` and defined
in `calc.c` and has already been completed, but you will need to read its contents in order to determine how to use it!

Evaluation

Once parsing is complete, evaluation of expressions comes down to completion of the functions called from the `calc` function in `calc.c`. These functions are all declared, but must be filled out (these are the `//TODO`s in `calc.c`).

The definitions of `evalNeg` and `evalAdd` have been provided as an example. If you're not sure what one of the functions is supposed to do, refer to the comments in the `resolveFunc` function near the top of `calc.c`.

Binary functions with two integer arguments should output an integer. Binary functions in which one or more input is a float should output a float. Some unary functions should always output a float, and others should output a number whose type is the same as that of their input; you should be able to figure out which should behave which way.

Sample Run

The following is a sample run with no arguments provided in the run configurations, and with the inputs from the provided `input.txt` typed in one at a time. If you choose instead to include a path to `input.txt` as the first argument, you will see just the outputs; the inputs (on lines starting with `>`) won't be typed in the console, as they'll be read from the file instead.

```
> 1
INT : 1

> 1.0
FLOAT : 1.000000
```

```
> -1
INT : -1

> -1.5
FLOAT : -1.500000

> +1
INT : 1

> +1.50
FLOAT : 1.500000

> 10
INT : 10

> 10.15
FLOAT : 10.150000

> -10.50
FLOAT : -10.500000

> (neg 1)
INT : -1

> (neg 1.0)
FLOAT : -1.000000

> (abs 1)
INT : 1

> (abs -1)
INT : 1

> (abs 1.5)
FLOAT : 1.500000

> (abs -1.0)
FLOAT : 1.000000

> (exp 1)
FLOAT : 2.718282

> (exp 1.0)
FLOAT : 2.718282

> (exp 0)
FLOAT : 1.000000

> (exp 0.)
FLOAT : 1.000000

> (log 1)
FLOAT : 0.000000

> (log 1.0)
```

```
FLOAT : 0.000000

> (log 10)
FLOAT : 2.302585

> (log 0)
FLOAT : -inf

> (log -1)
FLOAT : nan

> (sqrt 1)
FLOAT : 1.000000

> (sqrt 1.0)
FLOAT : 1.000000

> (sqrt 0)
FLOAT : 0.000000

> (sqrt 0.0)
FLOAT : 0.000000

> (sqrt 4)
FLOAT : 2.000000

> (sqrt 4.0)
FLOAT : 2.000000

> (sqrt -1)
FLOAT : nan

> (sqrt -1.0)
FLOAT : nan

> (add 1 2)
INT : 3

> (add 1.0 2)
FLOAT : 3.000000

> (add 1 2.0)
FLOAT : 3.000000

> (add 1.0 2.0)
FLOAT : 3.000000

> (sub 2 1)
INT : 1

> (sub 2.0 1)
FLOAT : 1.000000

> (sub 2 1.0)
FLOAT : 1.000000
```

```
> (sub 2.0 1.0)
FLOAT : 1.000000

> (mult 2 3)
INT : 6

> (mult 2 3.0)
FLOAT : 6.000000

> (mult 2.0 3)
FLOAT : 6.000000

> (mult 2.0 3.0)
FLOAT : 6.000000

> (div 1 2)
INT : 0

> (div 1 2.0)
FLOAT : 0.500000

> (div 1.0 2)
FLOAT : 0.500000

> (div 1.0 2.0)
FLOAT : 0.500000

> (div 1 0)
INT : inf

> (div 1.0 0.0)
FLOAT : inf

> (div 0 0)
INT : nan

> (rem 8 3)
INT : 2

> (rem -8 3)
INT : -2

> (rem 8.0 3)
FLOAT : 2.000000

> (rem 8 3.0)
FLOAT : 2.000000

> (rem 8.0 3.0)
FLOAT : 2.000000

> (rem 1 0)
INT : nan
```

```
> (rem 1 0.0)
FLOAT : nan

> (add 1 (add 2 3))
INT : 6

> (add 1 (add 2.0 3))
FLOAT : 6.000000

> (log (exp 1))
FLOAT : 1.000000

> (exp (log 1))
FLOAT : 1.000000

> (add 1 (add 2 (add 3 (add 4 5))))
INT : 15

> quit
Process finished with exit code 0
```

Submission Checklist

You need to:

- Complete the definitions section of `calc.y` to declare all `%token`s and `%type`s to match the grammar and complete the `%union` as necessary to store their data.

- Complete the definitions and rules sections of `calc.l` to configure a lexer which populates any necessary data in tokens and returns the correct tokens.

- Complete the rules section of `calc.y` to evaluate the parse tree from the bottom up.

- Complete the `TODO`s in `calc.c` to do the actual function evaluations.

You'll know when you're done, because your sample runs will match mine. You do not need to stress about immaterial differences (such as whether your nans and zeros are positive or negative).