LAB 11 - ciLisp Final Project

John Dempsey
COMP-232 Programming Languages
California State University, Channel Islands
November 6, 2024
Final Due Date: December 12, 2024 @ 11:59 PM

In this project, you will implement an interpreter for *CI LISP*, a *Channel Islands Lisp-like* language.

Each task below will expand the grammar. Your job is to expand the lexer, parser, data structures, parsing methods, and evaluation methods to meet the new requirements. The files provided will build and run (by running the "run" script), but isn't capable of actually doing anything other than crashing or quitting.

Tasks

TASK 1 : Tokenization, Parsing and Evaluation of Cambridge Polish Notation arithmetic

TASK 2 : Symbol definition and evaluation

TASK 3 : Typecasting

TASK 4 : Conditionals, comparisons, read, random and print

TASK 5 : Composable, recursion-capable user-defined functions

Point Breakdown

- **Task 1** **50 points**
- **Task 2** **20 points**
- **Task 3** **10 points**
- **Task 4** **10 points**
- **Task 5** **10 points**

Notes

1. **After completing a task, you need to upload your program to LAB11/TASK# on comp232.com so instructor can review your work.**

2. **After completing one task, you need to copy your code from the previous task and start with this code for the next task. Your code written for each task needs to be found in your LAB11/TASK# directories.**

3. **You must be able to compile and run your code on comp232.com. If instructor can't compile and/or run your code on comp232.com, I can't grade it.**

4. **You need to provide either a "run" script or a Makefile so that the instructor can run your code easily on comp232.com.**

5. **You need to run the test cases found in the INPUT directory for each task. After you have run the test cases on your system, you need to upload your test output files, which must end in .txt, into your LAB11/TASK# directories so instructor can review your work for each task.**

6. **On at least a weekly basis, you should upload all your current work to comp232.com in the event that the server is down on or before May 18th.**

7. **On or before May 21rst, all work must be uploaded to comp232.com.**

You will be given five (5) weeks to complete this assignment, but please manage your time wisely.

Student needs to create a **LAB11** directory and individual task directories on comp232.com and upload all source code and sample test run outputs regardless if the student has completed the task or not.

Instructor needs to see what tasks have been completed on-time or at least what progress has been made.

Points can be deducted if student does not meet all task requirements by the above due dates.

If student falls behind, instructor can request a one-on-one sessions to review and help with student's progress.

Test input files can be found in the **/home/LAB11/INPUTS** directory.

Sample test outputs from running your ciLisp program need to be found in your home directory under **~/LAB11/TASK#** directory with **file names ending in .txt** on comp232.com.

For example, output from running the Task 1 task_1_all.cilisp test case using your cilisp program need to be found in **~/LAB11/TASK1/task_1_all.txt** file. Likewise, output from running the Task 2 test case named task_2.cilisp using your cilisp program need to be found in **~/LAB11/TASK2/task_2.txt**.

Tasks must be completed in order. There is no extra credit for starting a new task if the previous task isn't completed first.

During finals week on December 7th, you need to have submitted all work before 11:59 PM. Late work will not be accepted as it's finals week.

Downloading /home/LAB11 Files

Files for this project are found on comp232.com in the /home/LAB11 directory.

To download these files from comp232.com onto your system, run the following commands:

```
john@oho:~$ pwd                    ← Start in your home directory on your laptop.
/home/john

john@oho:~$ mkdir LAB11            ← Create LAB11 directory.

john@oho:~$ cd LAB11               ← cd to LAB11.

john@oho:~/LAB11$ ls               ← Directory is empty.

john@oho:~/LAB11$ sftp comp232.com      ← Run sftp comp232.com.
Connected to comp232.com.
sftp> cd /home/LAB11               ← cd to /home/LAB11
sftp> mget -r *                    ← Run mget -r *      DON'T FORGET THE -r
Fetching /home/LAB11/INPUTS/ to INPUTS
Retrieving /home/LAB11/INPUTS
Retrieving /home/LAB11/INPUTS/task_1
/home/LAB11/INPUTS/task_1/hypot.cilisp          100%  219     2.3KB/s   00:00
/home/LAB11/INPUTS/task_1/max.cilisp            100%  111     1.3KB/s   00:00
/home/LAB11/INPUTS/task_1/number.cilisp         100%   58     0.8KB/s   00:00
/home/LAB11/INPUTS/task_1/sqrt.cilisp           100%  126     0.8KB/s   00:00
/home/LAB11/INPUTS/task_1/neg.cilisp            100%   64     1.1KB/s   00:00
/home/LAB11/INPUTS/task_1/exp.cilisp            100%   64     1.2KB/s   00:00
/home/LAB11/INPUTS/task_1/sub.cilisp            100%  133     2.5KB/s   00:00
```

```
/home/LAB11/INPUTS/task_1/composition.cilisp          100%  420     7.0KB/s   00:00
/home/LAB11/INPUTS/task_1/exp2.cilisp                 100%  104     2.0KB/s   00:00
/home/LAB11/INPUTS/task_1/mult.cilisp                 100%  106     1.6KB/s   00:00
/home/LAB11/INPUTS/task_1/log.cilisp                  100%  138     2.5KB/s   00:00
/home/LAB11/INPUTS/task_1/pow.cilisp                  100%  194     3.2KB/s   00:00
/home/LAB11/INPUTS/task_1/cbrt.cilisp                 100%  126     2.2KB/s   00:00
/home/LAB11/INPUTS/task_1/min.cilisp                  100%   83     1.5KB/s   00:00
/home/LAB11/INPUTS/task_1/remainder.cilisp            100%  241     4.6KB/s   00:00
/home/LAB11/INPUTS/task_1/div.cilisp                  100%  150     2.5KB/s   00:00
/home/LAB11/INPUTS/task_1/abs.cilisp                  100%   85     1.6KB/s   00:00
/home/LAB11/INPUTS/task_1/add.cilisp                  100%  110     2.0KB/s   00:00
/home/LAB11/INPUTS/task_5.cilisp                      100%  540     9.3KB/s   00:00
/home/LAB11/INPUTS/task_4_read_target.txt             100%   39     0.7KB/s   00:00
/home/LAB11/INPUTS/task_3.cilisp                      100%  112     1.8KB/s   00:00
/home/LAB11/INPUTS/task_4.cilisp                      100%  569    10.7KB/s   00:00
/home/LAB11/INPUTS/task_2.cilisp                      100%  705    13.0KB/s   00:00
Fetching /home/LAB11/bison_flex.log to bison_flex.log
Fetching /home/LAB11/cilisp.c to cilisp.c
/home/LAB11/cilisp.c                                  100% 5307    91.8KB/s   00:00
Fetching /home/LAB11/cilisp.h to cilisp.h
/home/LAB11/cilisp.h                                  100% 1605    28.6KB/s   00:00
Fetching /home/LAB11/cilisp.l to cilisp.l
/home/LAB11/cilisp.l                                  100% 1791    30.0KB/s   00:00
Fetching /home/LAB11/cilisp.y to cilisp.y
/home/LAB11/cilisp.y                                  100% 1140    21.7KB/s   00:00
Fetching /home/LAB11/yyreadprint.c to yyreadprint.c
/home/LAB11/yyreadprint.c                             100% 1957    31.3KB/s   00:00
sftp> quit

john@oho:~/LAB11$ ls                ← INPUTS is a directory containing test files.
INPUTS  bison_flex.log  cilisp.c  cilisp.h  cilisp.l  cilisp.y  yyreadprint.c

john@oho:~/LAB11$ ls -lR            ← Run a recursive list.
.:
total 32
drwxr-xr-x 3 root root 4096 Apr 12 22:11 INPUTS
-rw-r--r-- 1 root root 5307 Apr 12 22:11 cilisp.c
-rw-r--r-- 1 root root 1584 Apr 12 22:11 cilisp.h
-rw-r--r-- 1 root root 1821 Apr 12 22:11 cilisp.l
-rw-r--r-- 1 root root 1140 Apr 12 22:11 cilisp.y
-rwxr-xr-x 1 root root  300 Apr 12 22:13 run
-rw-r--r-- 1 root root 1957 Apr 12 22:11 yyreadprint.c
```

./INPUTS:
total 24
drw-r--r-- 2 root root 4096 Apr 12 22:42 task_1
-rw-r--r-- 1 root root 705 Apr 12 22:42 task_2.cilisp
-rw-r--r-- 1 root root 112 Apr 12 22:42 task_3.cilisp
-rw-r--r-- 1 root root 569 Apr 12 22:42 task_4.cilisp
-rw-r--r-- 1 root root   39 Apr 12 22:42 task_4_read_target.txt
-rw-r--r-- 1 root root 540 Apr 12 22:42 task_5.cilisp

./INPUTS/task_1:
total 72
-rw-r--r-- 1 root root 85 Apr 12 22:42 abs.cilisp
-rw-r--r-- 1 root root 110 Apr 12 22:42 add.cilisp
-rw-r--r-- 1 root root 126 Apr 12 22:42 cbrt.cilisp
-rw-r--r-- 1 root root 420 Apr 12 22:42 composition.cilisp
-rw-r--r-- 1 root root 150 Apr 12 22:42 div.cilisp
-rw-r--r-- 1 root root 64 Apr 12 22:42 exp.cilisp
-rw-r--r-- 1 root root 104 Apr 12 22:42 exp2.cilisp
-rw-r--r-- 1 root root 219 Apr 12 22:42 hypot.cilisp
-rw-r--r-- 1 root root 138 Apr 12 22:42 log.cilisp
-rw-r--r-- 1 root root 111 Apr 12 22:42 max.cilisp
-rw-r--r-- 1 root root 83 Apr 12 22:42 min.cilisp
-rw-r--r-- 1 root root 106 Apr 12 22:42 mult.cilisp
-rw-r--r-- 1 root root 64 Apr 12 22:42 neg.cilisp
-rw-r--r-- 1 root root 58 Apr 12 22:42 number.cilisp
-rw-r--r-- 1 root root 194 Apr 12 22:42 pow.cilisp
-rw-r--r-- 1 root root 241 Apr 12 22:42 remainder.cilisp
-rw-r--r-- 1 root root 126 Apr 12 22:42 sqrt.cilisp
-rw-r--r-- 1 root root 133 Apr 12 22:42 sub.cilisp

TASK 1

Recall **Cambridge Polish Notation** (**CPN**) in which operators and their operands are enclosed in parenthesis.

For example, `(add 1 2)` is the **CPN** equivalent of `1+2`.

Expressions in **CPN** can be nested, so the following expression would be valid, and would result in `6`.
`(add 1 (add 2 3))`

Your must create a functioning **CPN** calculator which works with the functions listed in the **FUNC_TYPE** enum in the provided code through the **MIN_FUNC** function (all functions appearing after **MIN_FUNC** in the enum will be implemented in later tasks.

This calculator will serve as the core functionality for **CI-LISP**.

You may want to check out the sample runs below better understand what will be implemented before reading further.

The initial grammar is as follows:

```
program ::= s_expr EOL | s_expr EOFT | EOL | EOFT

s_expr ::= f_expr | number | QUIT

f_expr ::= ( FUNC s_expr_section )

s_expr_section ::= s_expr_list | <empty>

s_expr_list ::= s_expr | s_expr s_expr_list

FUNC ::= neg | abs | add | sub |
  mult | div | remainder | exp |
  exp2 | pow | log | sqrt |
  cbrt | hypot | max | min

number ::= INT | DOUBLE

INT ::= optional +/-,
  then some digits

DOUBLE ::= optional +/-,
  then some digits,
  then a decimal point,
  then optionally some more digits

QUIT ::= quit
```

The non-terminals **s_expr** and **f_expr** are shorthand:

- **s_expr** means *symbolic* expression
- **f_expr** means *function* expression

DATA STRUCTURES

Next we'll discuss the structures provided in cilisp.h. For a more intuitive understanding of the structures and their names defined here, note that **AST** is short for *__A__bstract __S__yntax __T__ree*. As such, the structures discussed below are intended to house data in an abstract syntax tree.

NUMBERS

The **NUM_TYPE** enum and **AST_NUMBER** struct define everything necessary to house both numeric data types (integers and floating point values) in a single struct.

```
typedef enum num_type {
    INT_TYPE,
    DOUBLE_TYPE
} NUM_TYPE;

typedef struct {
    NUM_TYPE type;
    double value;
} AST_NUMBER;

typedef AST_NUMBER RET_VAL;
```

Note that even when an **AST_NUMBER**'s type is **INT_TYPE**, we will be storing it's actual value as a double under the hood. This is to make function evaluation a little less tedious; students wishing to instead use this alternative **AST_NUMBER** are welcome to do so:

```
typedef struct {
    NUM_TYPE type;
    union {
        double floating_point;
        long integral;
    } value;
} AST_NUMBER;
```

Each **AST_NUMBER** will house a member of the **NUM_TYPE** enum (denoting the type of number being stored) alongside a numeric value.

The last line, which gives the **AST_NUMBER** struct a second name (**RET_VAL**), exists exclusively for readability; when we evaluate a part of our syntax tree, the result will always be a number. We will use **AST_NUMBER**s to house numbers which are *part of* the syntax tree, and **RET_VAL**s to house the results of evaulation (*__RET__urned __VAL__ues*).

FUNCTIONS

The terms "operator" and "function" will be used interchangeably for our discussion here.

We'll start our discussion of functions with the enum and struct definitions in cilisp.h.

```
typedef enum func_type {
    NEG_FUNC,
    ABS_FUNC,
    ADD_FUNC,
    // TODO complete the enum
    CUSTOM_FUNC
} FUNC_TYPE;
```

This enum will list all functions which will be implemented throughout the course of the project. Clearly, it isn't yet complete, hence the `TODO`. You'll add more functions to it as you work through this task, and through future tasks.
When function calls are parsed as **f_expr**s, the data for said calls will be stored in an **AST_FUNCTION**:

```
typedef struct {
    FUNC_TYPE func;
    struct ast_node *opList;
} AST_FUNCTION;
```

This struct stores the operator used in the function call (a member of the **FUNC_TYPE** enum) and the operands for the function (a linked list of **struct ast_node**s, which we will discuss in the GENERIC NODES section below).

Take note of the **resolveFunc** function cilisp.c. This function takes as input a function's name (in string form), and outputs the corresponding **FUNC_TYPE** member. It will be used to assign value to **FUNC** tokens while lexing.

resolveFunc works because the array of function names (**funcNames**, in cilisp.c) lists all functions in the same order as the **FUNC_TYPE** enum. If either of these is edited, the other must also be edited to match. If they are not kept in sync, **resolveFunc** will not work. **CUSTOM_FUNC** should be left as the last element of **resolveFunc** for the same reason; user-defined functions will be implemented at the end of the project, and we'll know a function is user-defined if we make it through the whole **funcNames** array without finding the name of the function.

GENERIC NODES

We've already discussed two types of abstract syntax tree nodes for housing numbers and functions. In order to make these types of nodes interchangeable, it is necessary to wrap them in a generic abstract syntax tree node which can store either type of node.

As such, we have the **AST_NODE** type:

```
typedef enum {
    NUM_NODE_TYPE,
    FUNC_NODE_TYPE
} AST_NODE_TYPE;

typedef struct ast_node {
    AST_NODE_TYPE type;
    union {
        AST_NUMBER number;
        AST_FUNCTION function;
    } data;
    struct ast_node *next;
} AST_NODE;
```

An **AST_NODE** stores a member of the **AST_NODE_TYPE** enum in it's **type** field, so it can be determined what type of node is housed in its **data** field (a union which can store either a **AST_NUMBER** or a **AST_FUNCTION**). It also stores a pointer to another **AST_NODE** in

its **next** field, which allows **AST_NODE**s to be stored in a linked list style for use as operands (see the **opList** field in **AST_FUNCTION**).

LEXING

First, it is necessary to define all tokens and all non-terminals within the grammar. **token**s (and non-terminals, called **type**s by yacc), will be defined in cilisp.y in the definitions section. The provided token and type definitions are:

```
%union {
    double dval;
    int ival;
    struct ast_node *astNode;
};

%token <ival> FUNC
%token <dval> INT
%token QUIT EOL EOFT

%type <astNode> s_expr
```

Clearly, this is incomplete; you should know from the previous labs how to complete this definitions section, and we will not discuss it further here (but as always, questions are welcome).

The union contains all types that **token**s and **type**s will have. In this case, **token** and **type** values will all be stored as *double*, *int* or *ast_node **.

The tokens defined by the yacc file must be lexed. As we know, the lex file is used to configure a lexer.

The provided cilisp.l is barely started. It has rules to tokenize and return some tokens, but not all of them. You must complete it.

Pay attention to the **llog** calls made for debugging purposes each time a token is created. Just like in the lab, these calls print to a log file `src/bison-flex-outputs/flex_bison_log`. Every rule in cilisp.l should include an **llog** call.

PARSING

The goal of the parser is to construct an abstract syntax tree after tokenization. Most of the productions in your grammar will have an equivalent production in cilisp.y, which is the configuration file for the parser.

The first set of productions in cilisp.y are for parsing programs, which serve as an entry point. The productions are provided, and should be changed cautiously if at all. Productions for `s_expr ::= error` and `s_expr ::= QUIT` are also provided.

```
program:
```

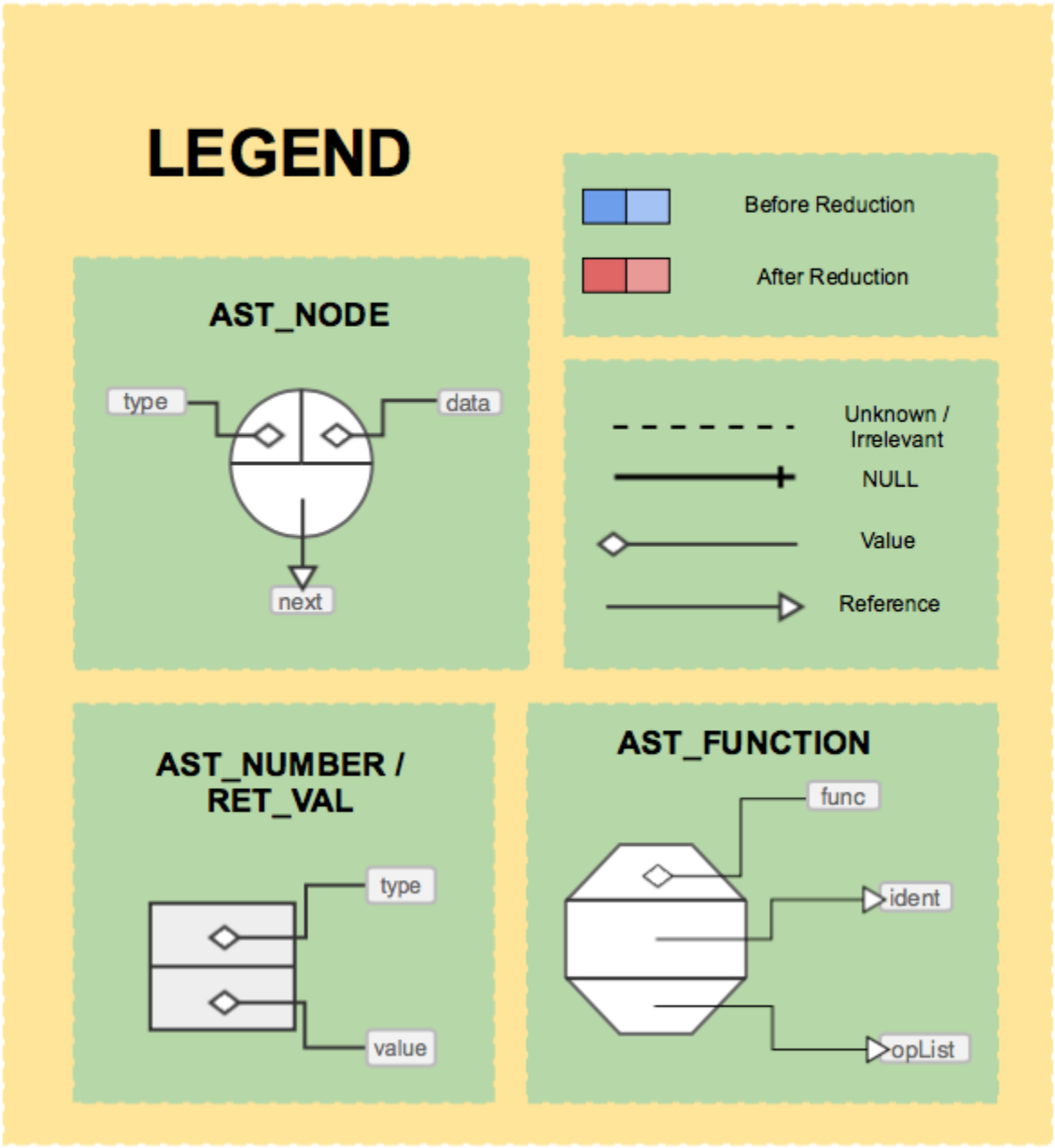
```
    s_expr EOL {
        ylog(program, s_expr EOL);
        if ($1) {
            printRetVal(eval($1));
            freeNode($1);
        }
        YYACCEPT;
    }
    | s_expr EOFT {
        ylog(program, s_expr EOFT);
        if ($1) {
            printRetVal(eval($1));
            freeNode($1);
        }
        exit(EXIT_SUCCESS);
    }
    | EOL {
        ylog(program, EOL);
        YYACCEPT;  // paranoic; main skips blank lines
    }
    | EOFT {
        ylog(program, EOFT);
        exit(EXIT_SUCCESS);
    };


s_expr:
    QUIT {
        ylog(s_expr, QUIT);
        exit(EXIT_SUCCESS);
    }
    | error {
        ylog(s_expr, error);
        yyerror("unexpected token");
        $$ = NULL;
    };
```

Note that every production has a **ylog** call; this function, like **llog**, prints to the log file stating what lexing and parsing steps are taken. Every production that you add should include a **ylog** file, as well, as the log will be a crucial tool in debugging the lexer and parser.

The rest of the yacc file is up to you.

The figure below (which can be found here) may help you visualize what needs to be done. Note that some reductions in the grammar are not illustrated in the figure.



You are **strongly** encouraged to illustrate productions like this as you go through the project, and to occasionally draw out entire syntax trees for test inputs.

Many of these productions will need to call the functions declared at the bottom of cilisp.h:

```
AST_NODE *createNumberNode(double value, NUM_TYPE type);
AST_NODE *createFunctionNode(FUNC_TYPE func, AST_NODE *opList);
AST_NODE *addExpressionToList(AST_NODE *newExpr, AST_NODE *exprList);
```

These functions are defined in cilisp.c; some of their definitions are partially completed already.

Whenever a function or value needs to be accessible by the lexer or parser, it must be declared in cilisp.h.

For instance, in the `number ::= INT` production, you will likely want to assign `$$` (the value of the **number**) to the result of a call to **createNumberNode**, which should create an **AST_NODE** housing an **AST_NUMBER** whose data is populated with that from the **INT** token (i.e. its double value) and the type (**INT_TYPE**, from the **NUMBER_TYPE** enum).
Both **createNumberNode** and **INT_TYPE** are accessible to the parser because they are declared in cilisp.h.

EVALUATION

GENERIC EVALUATION

Once parsing is complete (i.e. when a series of tokens is reduced to a sequence `s_expr EOL` or `s_expr EOFT`, the contents of a single program) the **eval** function is called on the root `s_expr`. You can see this call in the provided `program ::= s_expr EOL` production in cilisp.y. This function is the entry point for a recursive system of functions whose purpose is to evaluate the syntax tree and output the resulting **RET_VAL**.

While it is generally bad practice to pass structs by value, we will make an exception for the **RET_VAL** struct, which is sufficiently small for the issue to be irrelevant. Of course, if you wish to pass them around by reference instead, you're welcome to do so.

The **eval** function takes as input a generic **AST_NODE ***. Its job is to determine which type of **AST_NODE** is referrenced and call the corresponding function to evaluate it. That is, **eval** must determine if the referenced **AST_NODE** is of type **NUM_NODE_TYPE** or **FUNC_NODE_TYPE** and call **evalNumNode** or **evalFuncNode** on it respectively.

This process is similar to the evaluation prodedure carried out in the parser lab; no individual evaluation function should be too complex because of the recursive setup. For instance, when evaluating a function, you must evaluate its operands, but you don't need to do this by hand in **evalFuncNode**; you can instead simply call **eval** on each operand, and use the resulting **RET_VAL**s as operands for the function evaluation.

NUMBER EVALUATION

evalNumNode should be quick and easy; recall that a **RET_VAL** is really just an **AST_NUMBER** under a different name.

FUNCTION EVALUATION

evalFuncNode should check which **FUNC_TYPE** the **AST_FUNCTION** contains, and call an appropriate function to evaluate the result of running that operation with the input referenced linked list of operands. For instance, if the function being evaluated has type **NEG_TYPE**, you might pass it off to a function called **evalNegFunction** (which you'll need to add).

That is, **evalFuncNode** should be a glorified switch or if/else cascade, which just calls the individual function evaluation procedures. I **strongly** advise against doing said procedures inside of **evalFuncNode** instead of making these helper functions; it's totally doable, but it's also more tedious to read, debug, and expand.

Much of the work in function evaluation is catching and dealing with fringe cases. Often, you will need to print warnings as described in the function desciptions and sample runs below. Use the **warning** function, provided in cilisp.c for this purpose. It is essentically **printf** with a couple extra features to keep the console output pretty and well-spaced. It also prepends "WARNING: " onto whatever formatted string you pass it.

FUNCTION SPECIFICATIONS

Unary functions take 1 operand, binary take 2 operands, and *n*-ary take any number of operands. Functions which are given too few operands should print warnings and return `NAN_RET_VAL` (defined in cilisp.h) unless otherwise specified below. Functions which are given too many operands should print warnings stating that the extra operands were ignored, and then only use the required number of operands in calculation. If extra operands are provided, the leftmost operands should be used and the extras on the right ignored.

Check out the **math.h** library (open the console and type `man math` to get its documentation) before starting these implementations. Be careful: many of the **math.h** library functions don't behave as the specifications below describe, despite similar nomenclature. For instance, **math.h**'s **remainder** can return a negative value, but the specifications for the **remainder** function below specify that the result must be positive.

- neg
  - unary
  - returns the negation of the operand
  - i.e. given *x* returns *-x*
- abs
  - unary
  - returns the absolute value of the operand
- add
  - *n*-ary
  - returns the sum of its operands
  - return type depends on input types
  - if given no inputs, prints warning and returns *0* as an integer
- sub
  - binary
  - returns result of subtracting second operand from first operand
  - return type depends on operand types
- mult
  - *n*-ary

    - returns the product of the operands
    - return type depends on operand types
    - if given no inputs, prints warning and returns 1 as an integer
- div
    - binary
    - returns the result of division of the first operand by second operand
    - return type depends on operand types
    - int division if done with 2 integers
- remainder
    - binary (*a* and *b* input)
    - returns an *r* such that:
        - $a = q * b + r$
        - q is an integer
        - $0 <= r < |b|$
- exp
    - unary
    - returns *e* raised to the specified power
    - always returns a double
- exp2
    - unary
    - returns *2* raised to the specified power
    - if operand is negative, return type is always double
    - otherwise, return type depends on operand type
- pow
    - binary
    - returns first operand raised to power of second operand
    - return type depends on operand types
- log
    - unary
    - returns natural log of operand
    - always returns a double
- sqrt
    - unary
    - returns the square root of the operand
    - always returns a double
- cbrt
    - unary
    - returns the cube root of the input
    - always returns a double
- hypot
    - *n*-ary

    - returns the square root of the sum of squares of operands
    - if given no inputs, prints a warning and returns 0
    - always returns a double
    - `math.h`'s `hypot` only take 2 arguments; you'll need to build this by hand
- max
    - *n*-ary
    - returns maximum value among the operands
    - return type is the type of the maximum operand
    - if given no operands, prints a warning and returns `nan`
- min
    - *n*-ary
    - returns minimum value among the operands
    - return type is the type of the minimum operand
    - if given no operands, prints a warning and returns `nan`

SAMPLE RUNS

In the sample runs below, inputs are on the same line as the `> `, and the output is printed below (along with any warnings). The inputs shown in these sample runs are all provided in the `inputs/task_1` directory. You can edit the run configurations to provide a path to an input file, or run without any arguments and type inputs in by hand. The provided input files have `quit` so sample runs match runs done by hand, but this is unnecessary with file inputs; the program will end when an `EOF` is encounters as well.

You are encouraged to come up with a composite test input of your own, to test all of task 1, once your implementation matches the sample runs for all provided inputs.

When you're done implementing your lexing, parsing and evaluation procedures to match the sample runs below, don't forget to read the CLEANING UP section below.

number

```
> 0
Integer : 0

> 0.
Double : 0.000000

> 1
Integer : 1

> 1.
Double : 1.000000

> 0.0
Double : 0.000000

> 0.5
```

```
Double : 0.500000

> +0
Integer : 0

> +10.55
Double : 10.550000

> -12.87
Double : -12.870000

> -12.
Double : -12.000000

> -12
Integer : -12

> .34
WARNING: Invalid character >>.<<
Integer : 34

Process finished with exit code 0
```

neg

```
> (neg 5)
Integer : -5

> (neg 5.5)
Double : -5.500000

> (neg -5.0)
Double : 5.000000

> (neg -5)
Integer : 5

> (neg)
WARNING: neg called with no operands! nan returned!
Double : nan

> (neg 1 2)
WARNING: neg called with extra (ignored) operands!
Integer : -1

> quit

Process finished with exit code 0
```

abs

```
> (abs 1)
Integer : 1

> (abs 1.2)
```

```
Double : 1.200000

> (abs -3)
Integer : 3

> (abs 0)
Integer : 0

> (abs 0.0)
Double : 0.000000

> (abs -1.4)
Double : 1.400000

> (abs)
WARNING: abs called with no operands! nan returned!
Double : nan

> (abs -1 2)
WARNING: abs call with extra (ignored) operands!
Integer : 1

> quit

Process finished with exit code 0
```

add

```
> (add)
WARNING: add call with no operands, 0 returned!
Integer : 0

> (add 1)
Integer : 1

> (add 1.0)
Double : 1.000000

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

> (add 1 -2 3 -4 5 -6)
Integer : -3

> (add 0.0 1 -2 3 -4 5 -6)
Double : -3.000000

> (add 1 -1.0)
Double : 0.000000

> quit

Process finished with exit code 0
```

sub

```
> (sub)
WARNING: sub called with no operands!
Double : nan

> (sub 1)
WARNING: sub called with only one arg!
Double : nan

> (sub 1.0)
WARNING: sub called with only one arg!
Double : nan

> (sub 1 2)
Integer : -1

> (sub 2 1)
Integer : 1

> (sub 2 -1)
Integer : 3

> (sub 2.0 1)
Double : 1.000000

> (sub 2.0 -1)
Double : 3.000000

> (sub 1 1.0)
Double : 0.000000

> (sub 2.0 1.0)
Double : 1.000000

> (sub 1 2 3)
WARNING: sub called with extra (ignored) operands!
Integer : -1

> quit

Process finished with exit code 0
```

mult

```
> (mult)
WARNING: mult called with no operands! 1 returned!
Integer : 1

> (mult 1)
Integer : 1

> (mult 1.0)
Double : 1.000000
```

```
> (mult -1)
Integer : -1

> (mult -1 -1.0)
Double : 1.000000

> (mult 1 -2 3 -4 5)
Integer : 120

> (mult -1.0 2 -3.0 4 -5)
Double : -120.000000

> quit

Process finished with exit code 0
```

div

```
> (div)
WARNING: div called with no operands! nan returned!
Double : nan

> (div 1)
WARNING: div called with only one arg! nan returned!
Double : nan

> (div 1.0)
WARNING: div called with only one arg! nan returned!
Double : nan

> (div 1 2)
Integer : 0

> (div 1.0 2)
Double : 0.500000

> (div 2 1)
Integer : 2

> (div 2.0 1)
Double : 2.000000

> (div 5 2.0)
Double : 2.500000

> (div -20.0 4)
Double : -5.000000

> (div 1 2 3 4)
WARNING: div called with extra (ignored) operands!
Integer : 0

> (div 1 2 3)
WARNING: div called with extra (ignored) operands!
Integer : 0
```

```
> (div 5.0 2 3)
WARNING: div called with extra (ignored) operands!
Double : 2.500000

> quit

Process finished with exit code 0
```

remainder

```
> (remainder)
WARNING: remainder called with no operands! nan returned!
Double : nan

> (remainder 1)
WARNING: remainder called with one arg! nan returned!
Double : nan

> (remainder 0)
WARNING: remainder called with one arg! nan returned!
Double : nan

> (remainder -1.0)
WARNING: remainder called with one arg! nan returned!
Double : nan

> (remainder 1 2)
Integer : 1

> (remainder 2 1)
Integer : 0

> (remainder 2.5 1)
Double : 0.500000

> (remainder 2 3)
Integer : 2

> (remainder -6 10)
Integer : 4

> (remainder -6.0 10.0)
Double : 4.000000

> (remainder -6.0 -10.0)
Double : 4.000000

> (remainder 1 2 3)
WARNING: remainder called with extra (ignored) operands!
Integer : 1

> (remainder 23 7 10)
WARNING: remainder called with extra (ignored) operands!
Integer : 2

> quit
```

```
Process finished with exit code 0
```

exp

```
> (exp)
WARNING: exp called with no operands! nan returned!
Double : nan

> (exp 1)
Double : 2.718282

> (exp -1)
Double : 0.367879

> (exp 5.0)
Double : 148.413159

> (exp -2.0)
Double : 0.135335

> (exp 1 2)
WARNING: exp called with extra (ignored) operands!
Double : 2.718282

> quit

Process finished with exit code 0
```

exp2

```
> (exp2)
WARNING: exp2 called with no operands! nan returned!
Double : nan

> (exp2 1)
Integer : 2

> (exp2 1.0)
Double : 2.000000

> (exp2 0)
Integer : 1

> (exp2 0.0)
Double : 1.000000

> (exp2 0.5)
Double : 1.414214

> (exp2 -2)
Double : 0.250000

> (exp2 20.0)
Double : 1048576.000000
```

```
> (exp2 1 2)
WARNING: exp2 called with extra (ignored) operands!
Integer : 2

> quit

Process finished with exit code 0
```

pow

```
> (pow)
WARNING: pow called with no operands! nan returned!
Double : nan

> (pow 1)
WARNING: pow called with only one operand! nan returned!
Double : nan

> (pow 1.0)
WARNING: pow called with only one operand! nan returned!
Double : nan

> (pow 1 1)
Integer : 1

> (pow 1 1.0)
Double : 1.000000

> (pow 2 1)
Integer : 2

> (pow 2.1 1)
Double : 2.100000

> (pow -2 0.5)
Double : nan

> (pow -2 0)
Integer : 1

> (pow -2.0 0.0)
Double : 1.000000

> (pow -2.0 0)
Double : 1.000000

> (pow 3 3)
Integer : 27

> (pow 3.0 3)
Double : 27.000000

> (pow 27 (div 1 3.0))
Double : 3.000000
```

```
> (pow 1 2 3)
WARNING: pow called with extra (ignored) operands!
Integer : 1

> quit

Process finished with exit code 0
```

log

```
> (log)
WARNING: log called with no operands! nan returned!
Double : nan

> (log 1)
Double : 0.000000

> (log 0)
Double : -inf

> (log -1)
Double : nan

> (log 0.0)
Double : -inf

> (log -1.0)
Double : nan

> (log (exp 1))
Double : 1.000000

> (div (log 27) (log 3))
Double : 3.000000

> (div (log 27.0) (log 3))
Double : 3.000000

> (log 1 2)
WARNING: log called with extra (ignored) operands!
Double : 0.000000

> quit

Process finished with exit code 0
```

sqrt

```
> (sqrt)
WARNING: sqrt called with no operands! nan returned!
Double : nan

> (sqrt 1)
Double : 1.000000
```

```
> (sqrt 1.0)
Double : 1.000000

> (sqrt 0)
Double : 0.000000

> (sqrt 0.0)
Double : 0.000000

> (sqrt -1)
Double : nan

> (sqrt -1.0)
Double : nan

> (sqrt 4)
Double : 2.000000

> (sqrt 170.0)
Double : 13.038405

> (sqrt 2)
Double : 1.414214

> (sqrt 1 2)
WARNING: sqrt called with extra (ignored) operands!
Double : 1.000000

> quit

Process finished with exit code 0
```

cbrt

```
> (cbrt)
WARNING: cbrt called with no operands! nan returned!
Double : nan

> (cbrt 0)
Double : 0.000000

> (cbrt 0.0)
Double : 0.000000

> (cbrt -1)
Double : -1.000000

> (cbrt -1.0)
Double : -1.000000

> (cbrt 1)
Double : 1.000000

> (cbrt 1.0)
Double : 1.000000
```

```
> (cbrt 27)
Double : 3.000000

> (cbrt 27.0)
Double : 3.000000

> (cbrt 4)
Double : 1.587401

> (cbrt 1 2)
WARNING: cbrt called with extra (ignored) operands!
Double : 1.000000

> quit

Process finished with exit code 0
```

hypot

```
> (hypot)
WARNING: hypot called with no operands! 0 returned!
Double : 0.000000

> (hypot 1)
Double : 1.000000

> (hypot 1.0)
Double : 1.000000

> (hypot 3 4)
Double : 5.000000

> (hypot -3 4)
Double : 5.000000

> (hypot -30 -40.0)
Double : 50.000000

> (hypot 4 4 7)
Double : 9.000000

> (hypot 7.0 4 4.0)
Double : 9.000000

> (hypot 12 13 14)
Double : 22.561028

> (hypot 5 5 5)
Double : 8.660254

> (hypot -5 -5.0 (sqrt 25))
Double : 8.660254

> (hypot 0 0 0.0 -3 0 0 0 0 4 0.0 -0.0 12)
Double : 13.000000
```

```
> quit

Process finished with exit code 0
```

max

```
> (max)
WARNING: max called with no operands! nan returned!
Double : nan

> (max 1)
Integer : 1

> (max -1)
Integer : -1

> (max 1.0)
Double : 1.000000

> (max 232311.121)
Double : 232311.121000

> (max 1 2 3 4 5 6 7 8.0 9)
Integer : 9

> (max 1 2 25.0 -26.0 12)
Double : 25.000000

> quit

Process finished with exit code 0
```

min

```
> (min)
WARNING: min called with no operands! nan returned!
Double : nan

> (min 1)
Integer : 1

> (min 0.0)
Double : 0.000000

> (min 0)
Integer : 0

> (min -1 2 -3 4 -5 6)
Integer : -5

> (min -1.0 -12.0 12)
Double : -12.000000

> quit
```

```
Process finished with exit code 0
```

composition tests

```
> (exp (log 1))
Double : 1.000000

> (exp2 (div 1 2.0))
Double : 1.414214

> (cbrt (pow 3 3))
Double : 3.000000

> (cbrt (pow 3 6))
Double : 9.000000

> (log (exp (log (exp 1))))
Double : 1.000000

> (sub (mult 1 2 3 4) (add 1 2 3 4))
Integer : 14

> (sub (mult 1 2 3 -4.0) (add -1 -2 -3 -4))
Double : -14.000000

> (hypot (sqrt (div 100 7.0)) (mult 6 (sqrt (div 100.0 7))))
Double : 22.990681

> (hypot (sqrt (div 100 7.0)) (sqrt (mult 6 (div 100.0 7))))
Double : 10.000000

> (add 1 (add 2 (add 3 (add 4 (add 5 (add 6 (add 7)))))))
Integer : 28

> (add 1 (add 2 (add 3 (add 4 (add 5 (add 6 (sub 0 -7.0)))))))
Double : 28.000000

> quit

Process finished with exit code 0
```

CLEANING UP

The final task is cleaning up the syntax tree when we're done with it. Like evaluation, this should be done recursively.

The **freeNode** function is declared in cilisp.h, defined (incompletely) in cilisp.c, and called in cilisp.y after evaluation of an input is complete. **freeNode** will need to be completed for this task (though it should be relatively small). As you progress through the project, it will grow into a recursive system of functions to free the whole abstract syntax tree.

TASK 2

SUMMARY

Here you will add support for definition and evaluation of variables (called **symbols** in Lisp jargon). We'll add the following productions to the grammar:

```
s_expr ::= SYMBOL | ( let_section s_expr )

let_section ::= ( let let_list )

let_list ::= let_elem | let_elem let_list

let_elem ::= ( SYMBOL s_expr )

SYMBOL ::= at least 1 letter,
  followed by 0 or more letters and digits

letter ::= lowercase english alphabet letter
  | uppercase english alphabet letter
  | $ | _
```

You will need to update your project to support these additions.

While implementing the new productions above in your yacc file, be careful **not** to:

- allow `let_section`s to be empty (this in turn allows for redundant parenthesis, which will create parser conflicts when we implement 0-argument functions later on in the project).

- do left-recursion when making `let_list`s; this will create shift-reduce conflicts. In this case yacc can handle them, but still, never do left-recursion. In fact, you might want to go back and check your task 1 implementation to ensure that you didn't do left-recursion for `s_expr_list`s.

LEXING

Symbol tokenization should be a walk in the park; just make sure it comes after the function and keyword tokenizations in the lex file so that functions and keywords aren't tokenized as symbols!

In addition to symbol tokens, there is also a new keyword to add to your lexer.

DATA STRUCTURES

NEW NODE TYPES

First, you'll need a way to store symbols in **AST_NODE**s, just like you have already stored numbers and functions. You'll want to add two new elements to the **AST_NODE_TYPE** enum and create corresponding structs to store symbol data and the scope data in the syntax tree.

```
typedef enum {
    NUM_NODE_TYPE,
    FUNC_NODE_TYPE,
    SYM_NODE_TYPE,
    SCOPE_NODE_TYPE
} AST_NODE_TYPE;

typedef struct {
    char* id;
} AST_SYMBOL;

typedef struct {
    struct ast_node *child;
} AST_SCOPE;
```

SYMBOL DEFINITIONS

A `let_elem` defines a symbol. Its children are a `SYMBOL` token and an `s_expr`. The idea is to assign value to a variable using the identifier stored in the `SYMBOL` token and value of the `AST_NODE` corresponding to the `s_expr`.
You should use a linked list of symbols in each scope to keep track of the symbol names alongside their values. You may want to use this struct (or something similar) for this linked list:

```
typedef struct symbol_table_node {
    char *id;
    AST_NODE *value;
    struct symbol_table_node *next;
} SYMBOL_TABLE_NODE;
```

LINKING DEFINITIONS TO EXPRESSIONS

Finally, to link each scope to its contained symbol table and to parent scopes, and to allow **AST_SYMBOL**s to be stored in generic **AST_NODE**s you'll want to modify the **AST_NODE** struct:

```
typedef struct ast_node {
    AST_NODE_TYPE type;
```

```
    struct ast_node *parent;
    struct symbol_table_node *symbolTable;
    union {
        AST_NUMBER number;
        AST_FUNCTION function;
        AST_SYMBOL symbol;
        AST_SCOPE scope;
    } data;
    struct ast_node *next;
} AST_NODE;
```

SCOPING

Scope nodes will be the result of the `s_expr ::= ( let_section s_expr )` production; they should be minimal nodes, which store a reference to the child `s_expr` (the one to the right of the `let_section`) from the parent node (the one to the left).

Before we continue this discussion of scoping, it is important to clarify that every **AST_NODE** defines / contains a scope. **AST_SCOPE** nodes aren't special in this regard; to the contrary, they're special because they're **only** a scope.

Looking at the `s_expr ::= ( let_section s_expr )` production, it may not be readily apparent why scope nodes are necessary; why not just let the the `s_expr` on the left literally BE the `s_expr` on the right once the `let_section` has been referenced to store variable definitions. In other words, why not let this scope "collapse" into the child `s_expr` once it has been used to assign the `let_section` to that `s_expr`? To see why, consider an input in the form `( let_section ( let_section s_expr ) )`; without the use of a scope node, these two `let_section`s would have to be combined into a single `let_section` (i.e. into a single scope). But there are two scopes here! Symbols defined in the inner `let_section` should not be accessible in the outer `let_section`. For a more tangible example, check this input and explanation below.

SUBROUTINES

You will need to add methods for **AST_NODE** creation for both new types (**SCOPE_NODE_TYPE** and **SYMBOL_NODE_TYPE**), for linking symbol table nodes together into linked lists, for linking said linked lists of symbol table nodes up to the **AST_NODE**s whose scope they are in, and for looking through these symbol tables to find symbol definitions in order to evaluate them.

When evaluating symbols, you need to look not only through the local symbol table but also through the symbol tables in parent scopes; this is the purpose of the **parent** pointer added to **AST_NODE**s.

If an undefined variable is evaluated, a warning should be printed the **NAN_RET_VAL** should be returned. Symbol evaluation should look something like this:

```
current scope <- the ast_node being evaluated (a symbol node storing an identifier)

while the current scope is not null
{
        Go through the symbol table associated with the current scope.
        Check each symbol to see if it has the identifier you're looking for.
        eval and return if found.

        Set the current scope to its parent scope.
}

The variable was undefined, print a warning and return NAN.
```

You will need to assign these parent scopes in several of places:

- When the `s_expr ::= ( let_section s_expr )` production is performed:
  - The child scope (the `s_expr` on the right) must have its parent set to the parent scope (the `s_expr` on the left).
  - The child scope must have its symbol table assigned to the `let_section`.
  - The `s_expr`s stored in the `let_section` (i.e. the values of its defined symbols) must have their parent set to the child scope.

- When the `f_expr ::= ( FUNC s_expr_section )` production is performed, the `s_expr`s in the `let_section` should have their parent set to the `f_expr` (i.e. the parent of an operand is the function call containing it).

Whenever a symbol is evaluated, its value in the symbol table will be stored as an **AST_NODE**. That **AST_NODE** should be evaluated (whatever type it is). Then, if the **AST_NODE** is not a number node, it should be replaced with a number node storing the value which resulted from evaluating the original **AST_NODE**. This means that if a variable is accessed twice, the value found on the first evaluation will be used on the second evaluation; work will not be repeated and the value won't change if other variables change. Don't forget to call **freeNode** on the **AST_NODE** being replaced!

There are some specifications to be made with respect to how symbols and scoping should behave, and we will make them through examples discussed below. All of these inputs are in the task 2 test input, and included in the sample run (in order, for the most part, but with some others between them).

```
( (let (x 1) (x 2)) x )
```

Here the symbol `x` is defined twice in the same scope. The second definition should be discarded, (and a warning should be printed stating as much) and the output of the access to `x` should be `1`.

`( (let (x 1)) ( (let (x 2)) x ) )`
`x` is defined in two different (nested) scopes, and then accessed in the inner scope. The resulting value should be `2`, from evaluating the **more local** definition.
`( (let (x 1)) ( (let (y 2)) (add x y) ) )`
`x` and `y` are defined in different (nested) scopes. They should both be accessible where the addition call is made, becase it is in both scopes, and the result should be `3`.
`( (let (a ( (let (b 2)) (add b 3) ))) a )`
`a`'s let element contains a nested definition of `b`, and accesses it. `a` is given then value `5` (the result of adding `3` to `b`). There are no issues here; no variables are accessed outside of the scope in which they are defined.
`( (let (a ( (let (b 2)) (add b 3) ))) (add a b) )`
`b` is defined within the let element defining `a`. `b` is accessed, but outside of this let element (i.e. outside of the scope in which it is defined). `b` is undefined where it is accessed. A warning should be printed (undefined variable) and the attempted access to `b` should result in a **NAN_RET_VAL**.
`( (let (a 1) (b a)) ((let (a 2)) b ) )`
In the input above, `a` is defined to be `1`, and in the same scope `b` is defined to be `a`; in the scope in which `b` is accessed, `a` has been redefined to `2`.
Evaluating `b` requires first finding `b`'s definition (in the `let_section` on the left) and then evaluating the `s_expr` assigned to `b` (in this case, the symbol `a`). While the assignment of `a` to `2` is accessible to `b`, it is not accessible to the `let_elem` defining `b`, which is in an outer scope. The outer definition for `a`, however, is accessible to `b`'s `let_elem`; `b` should be `1`.
For the same reason, the input below should result in an undefined symbol warning and a **NAN_RET_VAL** output. The undefined symbol is the `a` within `b`'s let element.
`( (let (b a)) ((let (a 2)) b ) )`

SAMPLE RUN

Below is a sample run with the provided task 2 test input. These test cases should be used for initial debugging; the test inputs which will be used when grading will be qualitatively similar in their required functionality, but will be larger, more complex, deeper (in terms of parse tree depth, or depth of nested scopes if you prefer), and more varied in terms of the numeric inputs, functions, and identifiers used. The identifiers below are particularly bad for testing the symbol definition in the grammar; be sure to test with identifiers including digits, underscores, and dollar signs! In other words, these test the basic functionality for this task, but do little in terms of stress-testing or checking integration with task 1.

TLDR: Once your implementation matches this sample run for the provided task 2 inputs, you should create better test cases to more fully test the entire project so far.

```
> x
WARNING: Undefined Symbol "x" evaluated! NAN returned!
Double : nan

> ( (let (x 1) ) x )
Integer : 1

> ( (let (x 1) (x 2)) x )
WARNING: Duplicate assignment to symbol "x" detected in the same scope!
Only the first assignment is kept!
Integer : 1

> ( (let (x 1)) ( (let (x 2)) x ) )
Integer : 2

> ( (let (x 1)) ( (let (y 2)) (add x y) ) )
Integer : 3

> ( (let (a ( (let (b 2)) (add b 3) ))) a )
Integer : 5

> ( (let (a ( (let (b 2)) (add b 3) ))) (add a b) )
WARNING: Undefined Symbol "b" evaluated! NAN returned!
Double : nan

> ( (let (a 1) (b a)) ((let (a 2)) b ) )
Integer : 1

> ( (let (b a)) ((let (a 2)) b ) )
WARNING: Undefined Symbol "a" evaluated! NAN returned!
Double : nan

> ( (let (a 1) (b a)) ((let (c ( (let (a 2)) a ))) b ) )
Integer : 1

> ( (let (y 1) (x y)) ( (let (y 2) (c x)) x ) )
Integer : 1

> (add ((let (abc 1)) (sub 3 abc)) 4)
Integer : 6

> (mult ((let (a 1) (b 2)) (add a b)) (sqrt 2))
Double : 4.242641

> (add ((let (a ((let (b 2)) (mult b (sqrt 10))))) (div a 2)) ((let (c 5)) (sqrt c)))
Double : 5.398346

> ((let (first (sub 5 1)) (second 2)) (add (pow 2 first) (sqrt second)))
Double : 17.414214

> ((let (abc 1)) (sub ((let (abc 2) (de 3)) (add abc de)) abc))
Integer : 4

> quit
Process finished with exit code 0
```

TASK 3

In this task, you will add typecasting functionality to the `let_elem`, allowing symbols to have a specified type.

The new additions to the grammar are:

```
let_elem ::= ( TYPE SYMBOL s_expr )

TYPE ::= "int" | "double"
```

You'll need to expand the **SYMBOL_TABLE_NODE** to include a **type** field (storing a **NUM_TYPE**). You should also expand the **NUM_TYPE** enum to include a **NO_TYPE** element, for when **SYMBOL_TABLE_NODE**s are created without type-casting. Numbers with **NO_TYPE** should never exist; **NO_TYPE** is solely a placeholder to say "this variable will take whatever type it was assigned, without casting it", i.e. the variable will be duck-typed.

When evaluating symbols, you'll need to take their type into consideration; the **AST_NUMBER** (which should be stored in the **SYMBOL_TABLE_NODE**'s referenced value **AST_NODE** after evaluation) should be stored with typecasting applied. If this causes a loss of precision (i.e. if a double is cast as an integer) then a warning should be printed stating that this is the case.

This task should be quick; if you find you need to make more than a few simple changes, you're likely overthinking it!

The sample run below coincides with the provided task 3 test input. This **only** tests the basic new functionality, and does not ensure that it is well integrated with the old functionality. You should create more complex test cases, testing the entire project so far.

```
> ((let (int a 1.25))(add a 1))
WARNING: Precision loss on int cast from 1.25 to 1.
Integer : 2

> ((let (double a 5))(add a 25))
Double : 30.000000

> ((let (double $_1 (add 1 2))) (mult $_1 3))
Double : 9.000000

> quit

Process finished with exit code 0
```

TASK 4

SUMMARY

In this task, you will:

- Implement the **rand**, **read**, **equal**, **less**, **greater** and **print** functions.
- Implement conditionals (CI Lisp's equivalent of the ternary operator)

FUNCTION SPECIFICATIONS

The functions will will work as follows:

- **rand**
  - no arguments
  - generates and returns random double from `0` to `1`
  - You may use C's **rand** function (which has "limited randomness", i.e. if left unseeded it will output the same sequence of values on every run), but you must normalize the answer to be between 0 and 1 exactly.
  - `man rand` to learn more
  - If you use the **rand** function without seeding, and you normalize properly, then your sample run's **rand** outputs will match those below.
- **read**
  - No arguments.
  - Prints `read ::` to the `stdout`, and gets the user's response from the **read_target** file.
  - The **read_target** is already set up for you; it is declared in the header file and initialized in the main.
    - If you enter a second argument in the run configurations, it will be used as a file path to open the **read_target**.
    - If no second argument is provided, the **read_target** will be the console, so these inputs will be typed by hand (as was done in the sample runs).
  - If the user's response is an integer or double literal, as defined by CI Lisp's grammar, **read** returns the corresponding duck-typed number.
  - If the user's response is not a valid CILisp number literal, **read** should print a warning and return **NAN_RET_VAL**.
- **equal**, **less**, **greater**
  - Binary.
  - Return an integer with value `1` if the condition holds (i.e. if the first argument is equal to, less than, greater than the second respectively). returns an `int` with value `0` otherwise.
  - `1` is "true" and `0` is "false".

    - **equal** should ignore type, and should only check if the two numbers' stored values are **exactly** equal; no error threshold for floating point rounding errors should be used.
- **print**
    - Unary.
    - Evaluates its operand.
    - Prints the result of evaluation with **printRetVal**, and then returns it.
    - This should be very simple, as **printRetVal** is already implemented.

CONDITIONAL SPECIFICATIONS

The conditional operator will require adding the following to the grammar:

```
s_expr ::= ( COND s_expr s_expr s_expr )

COND ::= "cond"
```

You'll also need to add several more functions to the **FUNC** tokenization definition, the **FUNC_TYPE** enum and **funcNames** array.

Conditionals can be implemented via adding a new **AST_NODE_TYPE** member and corresponding struct, and integrating this new type into the **AST_NODE**. **Do not** implmenet conditionals by adding a new function instead; syntax errors should be thrown if a conditional doesn't have exactly 3 `s_exprs`.

If the first of the three `s_expr` (the condition) is nonzero (i.e. "true") when evaluated, then the conditional returns the value of the second `s_expr`. Otherwise, the condition is zero (i.e. "false") so the third `s_expr`'s value is returned.

If `A`, `B` and `C` are expressions, then CI Lisp's `( cond A B C )` is the equivalent of `A ? B : C` in C (the ternary operator).

SAMPLE RUN

The inputs for the sample run are provided in the task 4 sample input. It demonstrates the new functionality in very simple inputs, and tests some of the old functionality. As always, you should construct better test inputs to ensure that the new functionality is fully integrated with the old. A file is also provided for read inputs which matches those in this run; running with this file as the **read_target** will lead to the same outputs (but it won't be quite as pretty).

Make sure you understand why each value is printed in the sample run. Note that the result of the full program evaluation is always printed, so the last printed value following each input is not from a call to the **print** function, but from the **printRetVal** calls in the `.y` file, in the `program` productions. The last two inputs (not including the `quit`) in the sample run provide an opportunity to test specifications from task 2 which are often overlooked:

- In the second to last input, if x's value changes between first two prints, then variable evaluation is not being done as specified. The value stored in the symbol table should be replaced with an **AST_NODE** of type **NUMBER_NODE_TYPE** housing an **AST_NUMBER** the first time x is evaluated, so the second evaluation of x should not run the **rand** function again. The third printed value below is, of course, the result of the entire expression, which is `(add x x)`.

- Similarly, in the last input, while x and y are both accessed twice, there is only a single **read** call for each one; if x's definition is left as a function node containing a **read** call after the first evaluation, then **read** will be incorrectly called a second time for x, but if x's definition is replaced with a numerical value the first time it is accessed, then the second access simply gets this numerical value. The same applies to y. As such, the last input should do two reads (one for x and one for y), not four reads.

```
> (print)
WARNING: print called with no operands!
Double : nan

> (print 1)
Integer : 1
Integer : 1

> (print 1 2)
WARNING: print called with extra (ignored) operands!
Integer : 1
Integer : 1

> (add 1 (print 2) )
Integer : 2
Integer : 3

> (rand)
Double : 0.000008

> (rand)
Double : 0.131538

> (rand)
Double : 0.755605

> (rand)
Double : 0.458650

> (rand)
Double : 0.532767

> (rand)
Double : 0.218959

> (rand)
Double : 0.047045

> (rand)
Double : 0.678865
```

```
> (rand)
Double : 0.679296

> (rand)
Double : 0.934693

> (read)
read :: 1
Integer : 1

> (read)
read :: asdf
WARNING: Invalid read entry! NAN returned!
Double : nan

> (read)
read :: .5
WARNING: Invalid read entry! NAN returned!
Double : nan

> (read)
read :: -5.5
Double : -5.500000

> (add (read) (read))
read :: 10
read :: -10
Integer : 0

> (equal 0 0)
Integer : 1

> (equal 0 0.0)
Integer : 1

> (equal 0 0.0001)
Integer : 0

> (less 0 0)
Integer : 0

> (less -1 0)
Integer : 1

> (less 0 -0.00001)
Integer : 0

> (greater 0 1)
Integer : 0

> (greater 1 0)
Integer : 1

> (greater 0 0.0)
Integer : 0
```

```
> ( ( let (x 0) (y 1) ) (less x y) )
Integer : 1

> (cond 0 5 6)
Integer : 6

> (cond 1 5 6)
Integer : 5

> (cond (less 0 1) 5 6)
Integer : 5

> (cond (less 1 0) 5 6)
Integer : 6

> ( ( let (x (read)) (y (read)) ) (add (print x) (print x) (print y) (print y)) )
read :: -17.2
Double : -17.200000
Double : -17.200000
read :: +127
Integer : 127
Integer : 127
Double : 219.600000

> ( ( let (x (rand)) ) (add (print x) (print x) ) )
Double : 0.383502
Double : 0.383502
Double : 0.767004

> quit

Process finished with exit code 0
```

TASK 5

SUMMARY, LEXING AND PARSING

Your last task is to add support for user-defined functions via lambda expressions. These functions will work with composition with each other and with built-in functions. They will also work with recursion.

The final additions to the grammar will be:

```
f_expr ::= ( SYMBOL s_expr_section )

arg_list ::= SYMBOL | SYMBOL arg_list | <empty>

let_elem ::= ( SYMBOL LAMBDA ( arg_list ) s_expr )
  | ( TYPE SYMBOL LAMBDA ( arg_list ) s_expr )

LAMBDA ::= "lambda"
```

An example `let_elem` defining a function could be:

```
( int integerAdd lambda (x y) (add x y) )
```

The function defined above is called `integerAdd`. It takes 2 arguments, specified in its `arg_list`, called `x` and `y`. It returns the result of addition of `x` and `y`, but cast to an integer.

DATA STRUCTURES

Custom functions will need identifiers; you'll need to add identifier storage to the **AST_FUNCTION** struct for this purpose.

Function definitions will be stored in a symbol table. You may decide whether to store them in the same symbol table as variables or a separate table just for function definitions; I personally chose to do the prior.

In either case, you will need to expand the `SYMBOL_TABLE_NODE` to specify which type of symbol is being stored. Prior to this task, there was only one symbol type; we called it "symbol", but from here on out we'll refere to it as "variable". There are now two additional symbol types: lambda and argument. All three types of symbols will need to be treated differently.

It is strongly recommended that arguments are placed in their own symbol table for the following reasons:

- They should be checked last when looking up symbols, so local variables can be made to "reassign" arguments within a custom function's body.

- Values will need to be added to and removed from arguments in order; this is easier if you don't have to go through a list containing variable and function definitions to find the argument entries.

In order for custom functions to work recursively, you will need to implement a stack on which the argument values will be stored (arguments will not use the **value** field in the symbol table). The final data structures are below:

```
typedef enum {
    VAR_TYPE,
    LAMBDA_TYPE,
    ARG_TYPE
} SYMBOL_TYPE;

typedef struct symbol_table_node {
    char *id;
    AST_NODE *value;
    SYMBOL_TYPE symbolType;
    NUM_TYPE numType;
    struct stack_node *stack;
    struct symbol_table_node *next;
} SYMBOL_TABLE_NODE;

typedef struct stack_node {
    RET_VAL value;
    struct stack_node *next;
} STACK_NODE;
```

CUSTOM FUNCTION EVALUATION

Evaluating a custom function should be done as follows:

1. Evaluate all arguments. (You can temporarily stick their stack nodes together in a linked-list fashion).
2. Put all argument values on the stacks of their respective **ARG_TYPE** definitions in the the custom function's argument table.
3. Evaluate the custom function definition, using the values on the tops of the stacks when arguments are accessed. Store the result.
4. Pop the argument values off of the tops of their stacks.
5. Return the result.

Straying from the sequence of instructions above may result in functions which work on their own, but do not work recursively and/or with composition.

Note that a custom function and a variable may be defined in the same scope with the same identifier, because the parenthesis in `f_expr` syntax will make it clear which is being used. This means that when you are looking up a symbol for evaluation, you must ensure that the definition has not only the right identifier, but also the right symbol type; when looking up a function symbol, the defining symbol table node's type should be **LAMBDA_TYPE**, but when looking up a variable symbol the definition can come from either a **VAR_TYPE** or an **ARG_TYPE**.

Be sure to **not** replace custom function definitions with numeric values, as was done with variable evaluation.

Sample Run

The inputs for sample run below are provided in the task 5 sample input. As always, you should generated more complete composite tests. Depending on what markdown editor you're using, you'll likely need to scroll right to see the end of the last input (or just open the sample input file).

```
> ( (let (int integerAdd lambda (x y) (add x y))) (integerAdd 1.495 1.495) )
WARNING: Precision loss on int cast from 2.99 to 2.
Integer : 2

> ( (let (x 1.495) (int x lambda (a b) (add a b))) (x x x) )
WARNING: Precision loss on int cast from 2.99 to 2.
Integer : 2

> ( (let (double myFunc lambda (x y) (mult (add x 5) (sub y 2)))) (sub (myFunc 3 5)
2) )
Double : 22.000000

> ( (let (f lambda (x y) (add x y))) (f (sub 5 2) (mult 2 3)) )
Integer : 9

> ( (let (int a 1) (f lambda (x y) (add x y))) (f 2 (f a 3)) )
Integer : 6

> ( (let (x lambda () (rand))) (add (print (x)) (print (x))) )
Double : 0.000008
Double : 0.131538
Double : 0.131546

> ((let (double gcd lambda (x y) (cond (greater y x) (gcd y x) (cond (equal y 0) x
(gcd y (remainder x y)))))) (gcd 95.5 55))
Double : 0.500000

> quit

Process finished with exit code 0
```