# Project - ciLisp

Started: May 8 at 6:27pm

# Quiz Instructions

In this multi-task project that will use s-expressions that were defined in previous lectures and labs. We will call the language **ciLisp (Channel Islands Lisp)**, since it is similar to Lisp programming language.

This is a challenging project that requires your focus throughout a span of several weeks. There are eight tasks in the project with progressing level of complexity. The later parts are much more challenging than the initial tasks, so to stay on track and being able to wrap things up you should attempt to implement 1-2 tasks per week in the initial phase of the project. Completing the later tasks may take longer than a week to accomplish.

To implement the compiler, you will use flex (lex) and bison (yacc) that you learned in the previous labs. The implementation will span several stages; each stage will add a feature to the compiler. If you complete all stages (i.e., implement all specified features) then you will get maximum points for the whole project; otherwise, you will earn partial credits according to the number of features that you will have implemented.

At each stage, expand the ciLisp compiler to handle compilation of ciLisp programs utilizing the expanded grammar. Then extend the evaluator of abstract syntax trees so it handles the new features. Write a program in ciLisp that forces both the compiler and the evaluator to thoroughly test the new functionality.

---

***Task 1 (5 points)***

ciLisp.zip file contains seed code for an implementation of a compiler of ciLisp. We will be constructing abstract syntax trees representing the input s-expressions.

Subsequently, the root of the abstract syntax tree will be passed to an evaluation function `eval()` that should evaluate the abstract syntax tree executing de facto the compiled version of the s-expression-based program. In this way, function `eval()` implements a virtual machine executing compiled ciLisp programs in a way analogous to the way a JVM (Java Virtual Machine) executes compiled Java bytecode.

Recall that ciLisp programs use the **Cambridge Polish Notation** (**CPN**) that is a special prefix notation for expressions (that we will call **s-expressions**), in which the operator name and the operands are enclosed in parentheses. For example, `1+2` (or `add(1,2)`) is denoted as:

```
(add 1 2)
```

A number of predefined functions are given in the data structures included in the zip file.

The initial grammar for ciLisp is as follows:

```
program ::= s-expr EOL

s-expr ::= quit | number | f-expr

f-expr ::= ( func s-expr ) | ( func s-expr s-expr )

func ::= neg|abs|exp|sqrt|add|sub|mult|div|remainder|log|pow|max|min|exp2|cbrt|hypo
t

number ::= [+|-] digit+ [ . digit+ ]

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The project already includes the code to construct basic abstract syntax trees consisting of numbers and one- or two-parameter functions. ciLisp is still a purely functional language with no side effects. All computation is based on function composition.

Your task is to implement the evaluation function `eval()`.

Please note that a parameter `VERBOSE` has been added to `BISON_TARGET` in `CMakeLists.txt`. That forces generation of a report file from the parser called `ciLispParser.ouput` and located in `cmake-build-debug` directory. It is worth exploring that file to get insight in the state machine that the parser generates and to check if there are any issues in the parsing like shift/reduce conflicts. Keep in mind that bison can deal with many conflicts, but it's possible for it to get confused with more complex issues.

***Task 2 (25 points)***

The following grammar extends the ciLisp grammar to accommodate variables that in Lisp jargon are called **symbols**. Variables exist in potentially nested **scopes** that can be defined by the let section construct preceding ciLisp s-expressions. Each symbol assumes a value represented by the associated s-expression. A symbol can be any number of small and capital letters.

```
program ::= s-expr EOL

s-expr ::= quit
| number
| symbol
| f-expr
| ( let_section s_expr )
```

```
f-expr ::= ( func s-expr ) | ( func s-expr s-expr )

func ::= neg|abs|exp|sqrt|add|sub|mult|div|remainder|log|pow|max|min|exp2|cbrt|hypo
t

let_section ::= <empty> | ( let_list )

let_list ::= let let_elem | let_list let_elem

let_elem ::= ( symbol s_expr )

number ::= [+|-] digit+ [ . digit+ ]

symbol ::= letter+

letter ::= [a-zA-Z]

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Here are some sample expressions that use symbols:

```
(add ((let (abc 1)) (sub 3 abc)) 4)
(mult ((let (a 1) (b 2)) (add a b)) (sqrt 2))
(add ((let (a ((let (b 2)) (mult b (sqrt 10))))) (div a 2)) ((let (c 5)) (sqrt c)))

((let (first (sub 5 1)) (second 2)) (add (pow 2 first) (sqrt second)))
```

As you see, s-expressions are first class objects, so they can be assigned to variables.

Implement a bison-based parser of the language built over this grammar that creates a syntax tree and a symbol table of all identifiers. Assume static scoping. That means that there will be multiple symbol tables associated with the roots of the expressions within which they are defined. For example, using the following code sample, there would be a symbol table associated with `sub` in the outer expression, and another associated with `add` in the inner expression:

```
((let (abc 1)) (sub ((let (abc 2) (de 3)) (add abc de)) abc))
```

Missing symbols, for example as in the following expression:

```
((let (abc 1)) (sub ((let (abc 2)) (add abc de)) abc))
```

should be treated as compilation errors.

Redefining a symbol in the same scope is not allowed and should be flagged as an error.

You should use a linked list to keep the symbols. You will need to keep symbol names along with their values. Note that the value may be an s-expression, so you should just

keep a pointer to the root of the corresponding abstract syntax tree (that obviously may be just one element, the root holding the value, in the extreme case).

```
typedef struct symbol_table_node {
   char *ident;
   struct ast_node *val;
   struct symbol_table_node *next;
} SYMBOL_TABLE_NODE;
```

Add a link to the symbol table to `AST_NODE`:

```
typedef struct ast_node {
   AST_NODE_TYPE type;
   SYMBOL_TABLE_NODE *symbolTable;
   struct ast_node *parent;
   union {
      NUMBER_AST_NODE number;
      FUNCTION_AST_NODE function;
      SYMBOL_AST_NODE symbol;
   } data;
} AST_NODE;
```

You will need to add an AST node for symbol references to the nodes available for constructing abstract syntax trees of the programs.

```
typedef struct symbol_ast_node {
   char *name;
} SYMBOL_AST_NODE;
```

To implement the evaluation function that executes this and similar programs the code will have to perform symbol table lookups. ciLisp uses static scoping rules with the nested scoping principle,so you need to explore not only the symbol tables associated with a given s-expression, but also the symbol tables of all its ancestors. That's what the link `parent` in `AST_NODE` is for.

Such lookups may go up to the very root of the AST tree. A symbol reference in an s-expression should be declared undefined on after exploring all nested scopes.

Keep in mind that symbols may hold s-expressions rather than straight numerical values. Therefore, you may need to evaluate s-expressions that are bound to the symbols before using the symbols for further evaluation.

Write a program in ciLisp that forces the compiler and the evaluator to test the new functionality (both positively and negatively). For example, the following expressions should compile and evaluate with no errors:

```
(add ((let (abcd 1)) (sub 3 abcd)) 4)

(mult ((let (a 1) (b 2)) (add a b)) (sqrt 2))
```

```
(add ((let (a ((let (b 2)) (mult b (sqrt 10))))) (div a 2)) ((let (c 5)) (sqrt c)))

((let (first (sub 5 1)) (second 2)) (add (pow 2 first) (sqrt second)))

((let (a ((let (c 3) (d 4)) (mult c d)))) (sqrt a))
```

***Task 3 (15 points)***

The following grammar further expands the capabilities of ciLisp by adding data types:

- integers should provide precision equivalent to a long integer in C,
- reals should have a precision equivalent to double in C.

The types apply to numbers, symbols, and s-expressions. The type of a number must be derived from from its textual format: a number with a dot is an integer number, and a number with a dot is a real number. The type of a symbol must be assigned in the extended let section that now allows specifying types of symbols. The type of an s-expression has to be derived during evaluation of the expression from the types of its elements as described later.

The following is a revised grammar:

```
program ::= s-expr EOL

s-expr ::= quit
         | number
         | symbol
         | f-expr
         | ( let_section s_expr )

f-expr ::= ( func s-expr ) | ( func s-expr s-expr )

func ::= neg|abs|exp|sqrt|add|sub|mult|div|remainder|log|pow|max|min|exp2|cbrt|hypo
t

let_section ::= <empty> | ( let let_list )

let_list ::= let_elem | let_list let_elem

let_elem ::= ( type symbol s_expr )

type ::= <empty> | integer | real

real_number ::= [+|-] digit+ . digit+

integer_number ::= [+|-] digit+
```

```
symbol ::= letter+

letter ::= [a-zA-Z]

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Type is optional; symbols are assumed to be real (double) by default (i.e., if no type is provided).

This task needs an extension to `SYMBOL_TABLE_NODE` definition to indicate the type of the data:

```
typedef enum { NO_TYPE, INTEGER_TYPE, REAL_TYPE } DATA_TYPE;

typedef struct symbol_table_node {
   DATA_TYPE val_type;
   char *ident;
   struct ast_node *val;
   struct symbol_table_node *next;

} SYMBOL_TABLE_NODE;
```

To handle types of computed s-expressions, rather than a double the evaluator needs to return a structure that includes the value along with its type.

```
typedef struct return_value {
   DATA_TYPE type;
   double value;
} RETURN_VALUE;
```

```
RETURN_VALUE eval(AST_NODE *);
```

If an expression of type real is assigned to a symbol declared as `integer`, then the value of the expression should be rounded down or up before the assignment; e.g., `1.49` to `1`, and `1.69` to `2`. Each time such a coersion is performed, the evaluator should issue a warning:

```
"WARNING: precision loss in the assignment for variable <name>"
```

For example:

```
> ((let (integer a 1.25))(add a 1))
WARNING: precision loss in the assignment for variable a
2.000000
>
```

If the type of at least one of the operands is real, then type of the s-expression should be real. Adding, subtracting, multiplying, or negating operands of type integer should yield

an s-expression with type integer. Dividing two integer s-expressions that yield a result that is not integer, should result in rounding the value to an integer, assigning the integer type to it, and All other operations should return real values.

***Task 4 (5 points)***

The following grammar expands the capabilities of ciLisp by adding `print` function.

```
program ::= s-expr EOL

s-expr ::= quit
         | number
         | symbol
         | f-expr
         | ( let_section s_expr )

f-expr ::= ( func s-expr ) | ( func s-expr s-expr )

func ::= neg|abs|exp|sqrt|add|sub|mult|div|remainder|log|pow|max|min|exp2|cbrt|hypo
t|print

let_section ::= <empty> | ( let let_list )

let_list ::= let_elem | let_list let_elem

let_elem ::= ( type symbol s_expr )

type ::= <empty> | integer | real

real_number ::= [+|-] digit+ . digit+

integer_number ::= [+|-] digit+

symbol ::= letter+

letter ::= [a-zA-Z]

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The following expression:

```
> (print s_expr)
```

should print the value of the s-expression in the format depending on the type of the expression:

- real variables should print with double precision with two spaces after the dot (even if there are only zeros to the right of the dot),
- untyped values should print as real values, and
- integer variables should print without any fraction part.

The width of the print field should be minimized; i.e., values should use a tight format (no padding on the left).

For example:

```
> ((let (integer a 1))(print a))
=> 1
> ((let (real b 10))(print b))
=> 10.00
> (print 100)
=> 100.00
```

***Task 5 (10 points)***

The following grammar expands the capability of ciLisp by adding support for parameter lists of arbitrary length.

```
program ::= s-expr EOL

s-expr ::= quit
         | number
         | symbol
         | ( func s_expr_list )
         | ( let_section s_expr )
         | ( cond s_expr s_expr s_expr )

func ::= neg|abs|exp|sqrt|add|sub|mult|div|remainder|log|pow|max|min|exp2
         |cbrt|hypot|print|equal|smaller|larger

s_expr_list ::= s_expr s_expr_list | s_expr | <empty>

let_section ::= <empty> | ( let let_list )

let_list ::= let_elem | let_list let_elem

let_elem ::= ( type symbol s_expr )

type ::= <empty> | integer | real

real_number ::= [+|-] digit+ . digit+

integer_number ::= [+|-] digit+
```

```
symbol ::= letter+

letter ::= [a-zA-Z]
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The number of parameters in the list of parameters must meet the needs of the function. If there are too few parameters provided, the evaluator should issue a runtime error:

```
"ERROR: too few parameters for the function <name>"
```

and return the value of `0.0` (with type real, of course).

If more than necessary parameters are provided for a given function, then the evaluator should use as many as appropriate, and issue a runtime warning:

```
"WARNING: too many parameters for the function <name>"
```

The operations of addition and multiplication should accommodate any number of parameters.

To allow creations of lists of s-expressions, a link `next` is added to `AST_NODE:`

```
typedef struct ast_node {
   AST_NODE_TYPE type;
   SYMBOL_TABLE_NODE *symbolTable;
   struct ast_node *parent;
   union {
      NUMBER_AST_NODE number;
      FUNCTION_AST_NODE function;
      COND_AST_NODE condition;
      SYMBOL_AST_NODE symbol;
   } data;
   struct ast_node *next;
} AST_NODE;
```

The structure for the ast node for functions is also modifed, so it points to a list of parameters rather than just two:

```
typedef struct {
   char *name;
   struct ast_node *opList;
} FUNCTION_AST_NODE;
```

Expand the functionality of `add`, `mult`, and `print`, so they can handle multiple operands.

```
> (add 1 2 3 4 5)
15.00
>
```

`print` should print the values of all s-expressions in the list in one line separated by spaces and return the value of the last s-expression in the list as its own value.

```
> ((let (integer a 1)(real b 2))(print a b 3)
=> 1 2.00 3.00
3.00
>
```

***Task 6 (10 points)***

The following grammar expands the capability of ciLisp by adding conditional expressions and capability to read user provided and random values:

```
program ::= s-expr EOL

s-expr ::= quit
         | number
         | symbol
         | ( func s_expr_list )
         | ( let_section s_expr )
         | ( cond s_expr s_expr s_expr )

func ::= neg|abs|exp|sqrt|add|sub|mult|div|remainder|log|pow|max|min|exp2
        |cbrt|hypot|print|rand|read|equal|smaller|larger

let_section ::= <empty> | ( let let_list )

let_list ::= let_elem | let_list let_elem

let_elem ::= ( type symbol s_expr )

type ::= <empty> | integer | real

real_number ::= [+|-] digit+ . digit+

integer_number ::= [+|-] digit+

symbol ::= letter+

letter ::= [a-zA-Z]

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The following is the summary of changes:

1. Added `read` as a parameter-less function that takes values from the user.
   The user should be prompted to enter a value by displaying a prompt:

```
read :=
```

A symbol can be set to a value obtained in from the input as follows:

```
>((let (integer a (read)) (real b (read) (c (read)) (d (read))) (print a b c d))
read := 3
read := 5.0
read := 10
read := 5.175
=> 3 5.00 10 5.18
5.175000
>
```

*NOTES:*

1. *The numbers entered by the user are in red just for visualization. Your program should not attempt to apply any colors to printouts.*
2. If `read` is used to set a variable, then the user should be prompted only once for its value. The successive references to the variable should be resolved to the value that was read in the `read` section. Furthermore, if the entered value includes a dot, then the type of the variable should be set to real; otherwise - to integer.
3. `5.18` is printed in the example due to the rounding by `printf()`.

2. Added `rand` as a function that generates pseudo-random numbers.
   The function does not take any parameters. It can be called as follows:

   ```
   ((let (a 100)) (cond (smaller (rand) 100) (add a 2) (sub a 2)))
   ```

   The values generated by `rand` and `read` should be considered double unless assigned to a variable typed as integer.

3. Added comparison functions `equal`, `smaller`, and `larger`. The functions return 1 if the condition holds, and 0 otherwise.

4. Added `cond` function that checks if the first s-expression is true (non-zero). If so, then the next expression is evaluated and returned. Otherwise, the third s-expression is evaluated and returned.
   For example:

   ```
   ((let (myA (read))(myB (rand)))(cond (smaller myA myB) (print myA) (print myB)))
   ```

   should print the value of `myA` if the expression `(smaller myA myB)` is true (non-zero); otherwise, the value of `myB` should be printed.
   If the function is not one of the relationship functions (i.e., neither `equal`, `smaller`, nor `larger`), then the expression should be evaluated, and if it equals to zero, then

the condition should be considered false; otherwise (any value other than zero), the condition should be considered true.

You will need another `AST_NODE` type to encode the condition in the abstract syntax tree. The following is a prototype for that node:

```
typedef struct {
   struct ast_node *cond;
   struct ast_node *zero;
   struct ast_node *nonzero;
} COND_AST_NODE;

typedef struct ast_node {
   AST_NODE_TYPE type;
   SYMBOL_TABLE_NODE *scope;
   struct ast_node *parent;
   union {
      NUMBER_AST_NODE number;
      FUNCTION_AST_NODE function;
      COND_AST_NODE condition;
      SYMBOL_AST_NODE symbol;
   } data;
} AST_NODE;
```

***Task 7 (20 points)***

The following grammar expands the capability of ciLisp by adding support for user-defined functions:

```
program ::= s-expr EOL

s-expr ::= quit
         | number
         | symbol
         | ( func s_expr_list )
         | ( let_section s_expr )
         | ( cond s_expr s_expr s_expr )
         | ( symbol s_expr_list )

func ::= neg|abs|exp|sqrt|add|sub|mult|div|remainder|log|pow|max|min|exp2
        |cbrt|hypot|rand|read|print|equal|smaller|larger

s_expr_list ::= s_expr s_expr_list | s_expr

let_section ::= <empty> | ( let let_list )

let_list ::= let_elem | let_list let_elem
```

```
let_elem ::= ( type symbol s_expr )
           | ( type symbol lambda ( arg_list ) s_expr )

type ::= <empty> | integer | real

arg_list ::= symbol arg_list | symbol

real_number ::= [+|-] digit+ . digit+

integer_number ::= [+|-] digit+

symbol ::= letter+

letter ::= [a-zA-Z]

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Function definitions should be placed in the let section. A function should have a positive number of formal arguments (symbols). The body of the function is an expression that may use formal parameters. For example:

```
((let (real myFunc lambda (x y) (mult (add x 5) (sub y 2)))) (sub (myFunc 3 5) 2))
```

Similarly to the use of `let` and `cond`, the `lambda` keyword should be used just for parsing.

As shown, a function must be called with the number of actual parameters that comply with the arity rules described in the previous task for lists of function parameters. The same errors and warnings should be issues as needed.

The value of the expression that is the body of the function is the value that the function returns.

Formal parameters have type `ARG_TYPE` and have `NULL` as values. For example, in the tree for

```
((let (f lambda (x y) (add x y)))(f (sub 5 2) (mult 2 3)))
```

`x` and `y` are symbols in the scope of `add` with type `ARG_TYPE`, and with `x.val=NULL` and `y.val=NULL`.

When the evaluator processes `(f (sub 5 2) (mult 2 3))`, it will construct an individual stack for each of the formal parameters using `x.stack` and `y.stack`. The stacks are implemented as linked lists that use `STACK_NODE`. The `x.stack->val` field of the top of the first stack points to the root of the tree for `(sub 5 2)` and `y.stack->val` - to the root of the tree for `(mult 2 3)`. In this way, both formal parameters now have individual stacks with the tops pointing to the actual parameters (i.e., the roots of the corresponding s-expression trees). Now, `(add x y)` can be evaluated as both `x` and `y` can be resolved by evaluating the actual parameters from the top of their respective

calling stacks. After the evaluation, the tops of both `x.stack` and `y.stack` must be popped.

This task needs some modifications and extra definitions to the list of symbols in a scope:

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

typedef struct stack_node {
   struct ast_node *val;
   struct stack_node *next;
} STACK_NODE;

typedef struct symbol_table_node {
   SYMBOL_TYPE type;
   DATA_TYPE val_type;
   char *ident;
   struct ast_node *val;
   STACK_NODE *stack;
   struct symbol_table_node *next;
} SYMBOL_TABLE_NODE;
```

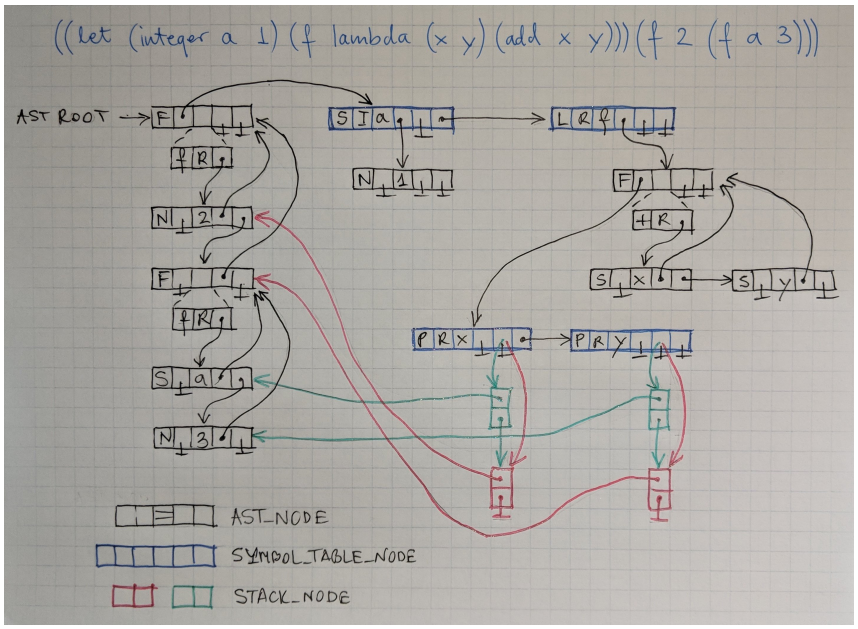
***Task 8 (10 points)***

Implement evaluator handling of recursive custom functions.

The evaluator should handle expression similar to the following:

```
((let (integer a 1) (f lambda (x y) (add x y)))(f 2 (f a 3)))
```

To handle recursive calls, subsequent recursive invocations require pushing sets of actual parameters on the stack. All individual stacks associated with formal parameters hold actual parameters corresponding to subsequent recursive invocations of a function. For our sample expression, after the recursive invocation of `f` the stack for `x` is `x.stack->a->2` (simplified view; the actual stack uses the `STACK_NODE` structure) and the stack for `y` is `y.stack->3->(f a 3)` (where `a` and `3` are correspondingly stack tops for formal parameters `x` and `y`). After `(f a 3)` is evaluated (as `(add a 3)` with `x<-a` and `y<-3` substitutions) the stacks will be `x.stack->2` and `y.stack->4`, since `a` is popped from `x.stack` and `3` is popped from `y.stack`, and the evaluation of `(add a 3)` is `4` for `a` set to `1` in the scope.




## Question 1

100 pts

***SUBMISSION***

Submit a zip file of your ciLisp CLion project. You can submit your work in progress as many times as you wish, however only the last submission will be graded.

Provide a commentary that clearly describes the scope of your implementation (i.e., how many tasks have you manage to implement) and what issues does it have.

Make sure that the following test cases pass and the transcript of your test run is included in your submission:

```
(add ((let (abcd 1)) (sub 3 abcd)) 4)
(mult ((let (a 1) (b 2)) (add a b)) (sqrt 2))
(add ((let (a ((let (b 2)) (mult b (sqrt 10))))) (div a 2)) ((let (c 5)) (sqrt c)))

((let (first (sub 5 1)) (second 2)) (add (pow 2 first) (sqrt second)))
((let (a ((let (c 3) (d 4)) (mult c d)))) (sqrt a))
((let (integer a 1))(print a))
((let (real b 10))(print b))
((let (integer a (read)) (real b (read))) (print a b))
((let (a 100)) (cond (smaller (rand) 100) (add a 2) (sub a 2)))
((let (myA (read))(myB (rand)))(cond (smaller myA myB) (print myA) (print myB)))
(add 1 2 3 4 5)
((let (integer a 1)(real b 2))(print a b 3))
((let (real myFunc lambda (x y) (mult (add x 5) (sub y 2)))) (sub (myFunc 3 5) 2))
((let (f lambda (x y) (add x y)))(f (sub 5 2) (mult 2 3)))
((let (a 1) (f lambda (x y) (add x y)))(f 2 (f a 3)))
```

Upload Choose a File

No new data to save. Last checked at 6:31pm Submit Quiz