# crumbL

Raymond Chee, Chris Denny, Mark Ebbole, Carolyn Tran, Sarah Wang, Angela Xie

Thursday December 1

## 1 Introduction

Crumbl, stylized as crumbL, is an interpreted imperative language with features similar to that of L. Like L, crumbL supports integer operations, function definitions, the use of identifiers, recursion, branching, and list operations. In addition, crumbL allows if statements with no else clause, while loops, string concatentation and comparisons, coercion of integers to strings, negative integer constants, and lazy assignments.

Our implementation of lazy assignments gives the best of both call-by-value and call-by-name semantics by only evaluating lazily assigned identifiers when they are needed in some other expression and then storing that value so it does not have to be evaluated for each additional reference.

Overall, crumbL brings L into the imperative world with a few additional benefits that make it a simple but powerful language.

## 2 Grammar

The Context-Free Grammar of crumbL is as follows:

$$\begin{array}{lll}
\text{P} & \longrightarrow & \epsilon \\
 & & \mid P\ S \\
\text{S} & \longrightarrow & \text{id} = \text{e;} \\
 & & \mid \text{lazy id} = e; \\
 & & \mid \text{if } (e) \text{ then } P \text{ fi} \\
 & & \mid \text{if } (e) \text{ then } P \text{ else } P \text{ fi} \\
 & & \mid \text{while } (e) \text{ do } P \text{ ob} \\
 & & \mid \text{print}(e); \\
 & & \mid \text{func id}(p\_list)\ P \text{ ret } e\text{; cnuf} \\
\text{e} & \longrightarrow & \text{id} \\
 & & \mid e_1 + e_2 \\
 & & \mid e_1 - e_2 \\
 & & \mid e_1 * e_2 \\
 & & \mid e_1\ /\ e_2
\end{array}$$

| | | |
|---|---|---|
| | | $\mid e_1$ % $e_2$ |
| | | $\mid e_1$ @ $e_2$ |
| | | $\mid e_1 :: e_2$ |
| | | $\mid e_1 < e_2$ |
| | | $\mid e_1 <= e_2$ |
| | | $\mid e_1 >= e_2$ |
| | | $\mid e_1 > e_2$ |
| | | $\mid e_1 == e_2$ |
| | | $\mid e_1 != e_2$ |
| | | $\mid$ not $e$ |
| | | $\mid e_1$ and $e_2$ |
| | | $\mid e_1$ or $e_2$ |
| | | $\mid -e$ |
| | | $\mid$ isNil $e$ |
| | | $\mid !e$ |
| | | $\mid \#e$ |
| | | \| INT_CONST |
| | | \| STRING_CONST |
| | | \| Nil |
| | | \| readString() |
| | | \| readInt() |
| | | \| func_name(call_list) |
| p_list | $\longrightarrow$ | $\epsilon$ |
| | | \| p_list2 |
| p_list2 | $\longrightarrow$ | id |
| | | \| lazy id |
| | | \| id, p_list2 |
| | | \| lazy id, p_list2 |
| call_list | $\longrightarrow$ | $\epsilon$ |
| | | \| call_list2 |
| call_list2 | $\longrightarrow$ | $e$ |
| | | $\mid e$, call_list2 |

### 2.1 Operator Precedence

```
print
= != < > <= >=
and or
+ - ::
* / %
isNil
@
!  #
not
```

# 3 Operational Semantics

## 3.1 Environments

$E$ - a mapping from identifiers to values.
$F$ - a mapping from identifiers to function definitions.
A function definition has three fields:

prog: a statement of the form $P$ (see context free grammar)

r: a statement of the form $e$ that comes after the ret keyword.

p_list: a list of identifiers that make up the parameter list of the function definition.

## 3.2 The Fundamental crumbL Statement

$$\frac{\begin{array}{c} E, F \vdash S_1 : E', F' \\ E', F' \vdash S_2 : E'', F'' \end{array}}{E, F \vdash S_1\ S_2 : E'', F''}$$

## 3.3 Constants

$$\frac{\text{integer i}}{E, F \vdash \text{i} : \text{i}}$$

$$\frac{\text{String s}}{E, F \vdash \text{s} : \text{s}}$$

$$\frac{}{E, F \vdash \text{Nil} : \text{Nil}}$$

## 3.4 Arithmetic

$$\frac{\begin{array}{c} E, F \vdash e_1 : i_1(\text{integer}) \\ E, F \vdash e_2 : i_2(\text{integer}) \end{array}}{E, F \vdash e_1 \oplus e_2 : i_1 \oplus i_2}$$

where $\oplus \in \{+, -, *, \%\}$

$$\frac{\begin{array}{c} E, F \vdash e_1 : i_1(\text{integer}) \\ E, F \vdash e_2 : i_2(\text{integer}) \\ i_2 \neq 0 \end{array}}{E, F \vdash e_1 / e_2 : i_1 / i_2}$$

$$\frac{E, F \vdash e : i(\text{integer})}{E, F \vdash -e : -i}$$

$$\frac{\begin{array}{c}E, F \vdash e_1 : s_1(\text{int or string}) \\ E, F \vdash e_2 : s_2(\text{int or string})\end{array}}{E, F \vdash e_1 :: e_2 : s_1 s_2}$$

## 3.5 Lists

$$\frac{\begin{array}{c}E, F \vdash e_1 : v_1(\text{not a list}) \\ E, F \vdash e_2 : v_2 \text{ (not Nil)}\end{array}}{E, F \vdash e_1 @ e_2 : [v_1, v_2]}$$

$$\frac{\begin{array}{c}E, F \vdash e_1 : v_1(\text{not a list}) \\ E, F \vdash e_2 : \text{Nil}\end{array}}{E, F \vdash e_1 @ e_2 : v_1}$$

$$\frac{E, F \vdash e : [v_1, v_2]}{E, F \vdash !e : v_1}$$

$$\frac{E, F \vdash e : [v_1, v_2]}{E, F \vdash \#e : v_2}$$

$$\frac{E, F \vdash e : v_1(\text{not a list})}{E, F \vdash !e : v_1}$$

$$\frac{E, F \vdash e : v_1(\text{not a list})}{E, F \vdash \#e : Nil}$$

## 3.6 Boolean Logic

$$\frac{\begin{array}{c}E, F \vdash e_1 : v_1 \\ E, F \vdash e_2 : v_2\end{array}}{E, F \vdash e_1 \odot e_2 : v_1 \odot v_2}$$

where $v_1$ and $v_2$ are either both strings or both integers. If strings, comparisons are lexicgraphic.

$\odot \in \{<, >, <=, >=, ==, ! =\}$

Let **False** be "", 0, or Nil.

**True** is any expression that evaluates to something that is not **False**.

$$\frac{\begin{array}{c}E, F \vdash e_1 : \text{True} \\ E, F \vdash e_2 : \text{True}\end{array}}{E, F \vdash e_1 \text{ and } e_2 : 1}$$

$$\frac{E, F \vdash e_1 : \text{False}}{E, F \vdash e_1 \text{ and } e_2 : 0}$$

$$\frac{\begin{array}{c} E, F \vdash e_1 : \text{True} \\ E, F \vdash e_2 : \text{False} \end{array}}{E, F \vdash e_1 \text{ and } e_2 : 0}$$

$$\frac{E, F \vdash e_1 : \text{True}}{E, F \vdash e_1 \text{ or } e_2 : 1}$$

$$\frac{\begin{array}{c} E, F \vdash e_1 : \text{False} \\ E, F \vdash e_2 : \text{False} \end{array}}{E, F \vdash e_1 \text{ or } e_2 : 0}$$

$$\frac{\begin{array}{c} E, F \vdash e_1 : \text{False} \\ E, F \vdash e_2 : \text{True} \end{array}}{E, F \vdash e_1 \text{ or } e_2 : 1}$$

$$\frac{E, F \vdash e_1 : \text{True}}{E, F \vdash \text{not } e_1 : 0}$$

$$\frac{E, F \vdash e_1 : \text{False}}{E, F \vdash \text{not } e_1 : 1}$$

$$\frac{E, F \vdash e_1 : Nil}{E, F \vdash \text{isNil } e_1 : 1}$$

$$\frac{E, F \vdash e_1 : \text{not Nil}}{E, F \vdash \text{isNil } e_1 : 0}$$

## 3.7 Conditional Statements

$$\frac{\begin{array}{c} E, F \vdash C : \text{True} \\ E, F \vdash S_1 : E', F' \end{array}}{E, F \vdash \text{if } (C) \text{ then } S_1 \text{ else } S_2 \text{ fi } : E', F'}$$

$$\frac{\begin{array}{c} E, F \vdash C : \text{False} \\ E, F \vdash S_2 : E', F' \end{array}}{E, F \vdash \text{if } (C) \text{ then } S_1 \text{ else } S_2 \text{ fi } : E', F'}$$

$$\frac{\begin{array}{c} E, F \vdash C : \text{True} \\ E, F \vdash S : E', F' \end{array}}{E, F \vdash \text{if } (C) \text{ then } S \text{ fi } : E', F'}$$

$$\frac{E, F \vdash C : \text{False}}{E, F \vdash \text{if } (C) \text{ then } S \text{ fi } : E, F}$$

$$\frac{E, F \vdash C : \text{False}}{E, F \vdash \text{while}(C) \text{ do } S \text{ ob } : E, F}$$

$$\frac{\begin{array}{c} E, F \vdash C : \text{True} \\ E, F \vdash S : E', F \\ E', F \vdash \text{while}(C) \text{ do } S \text{ ob } : E'', F \end{array}}{E, F \vdash while(C) \text{ do } S \text{ ob } : E'', F}$$

Note: Under this rule, the statement S must not change the function environment.

## 3.8 Identifiers and Functions

Note: The environment E has two separate mappings E.n and E.l.

E.n contains all non-lazy, evaluated expressions stored for an identifier, and E.l contains all lazily assigned expressions for an identifier.

This separation of mappings is necessary in order to avoid situations in which an identifier holds an unevaluated expression which contains itself.

$$\frac{E, F \vdash e : v}{E, F \vdash id = e; : E.n[id \leftarrow v], F}$$

$$\frac{}{E, F \vdash \text{lazy } id = e; : E.l[id \leftarrow e], F}$$

$$\frac{\begin{array}{c} E.l[id] \text{ exists} \\ e = E.l[id] \\ E, F \vdash e : v \\ E' = E.n[id \leftarrow v] \\ E'' = E.l[id \leftarrow \emptyset] \end{array}}{E, F \vdash id : v, E'', F}$$

$$\frac{\begin{array}{c} E.l[id] \text{ doesn't exist} \\ e = E.n[id] \end{array}}{E, F \vdash id : v, E, F}$$

$$\frac{\begin{array}{c} fentry = \{\text{prog: P, r: e, p\_list: p\_list}\} \\ F' = F[\textbf{fname} \leftarrow fentry] \end{array}}{E, F \vdash \text{func } \textbf{fname}(\text{p\_list})\ P \text{ ret } e;\ \text{cnuf}\ : E, F'}$$

$$\frac{\begin{array}{c} fentry = F[\textbf{fname}] \\ E' = E \cup \text{apply}(fentry.\text{p\_list},\ \text{call\_list}) \\ p = fentry.\text{prog} \\ E', F \vdash p : E'' \\ E'', F \vdash fentry.\text{r} : v \end{array}}{E, F \vdash \textbf{fname}(\text{call\_list}) : E, F, v}$$

Note: apply is just an operational semantics subroutine to construct a new environment for a called function, and is not useable from source code.

$$\frac{\begin{array}{c} E, F \vdash \text{p\_list} = [p_1, R_1] \\ E, F \vdash \text{call\_list} = [e_1, R_2] \\ E, F \vdash e_1 : v_1 \\ E, F \vdash \text{apply}(R_1, R_2) : E' \\ E'' = E'[p_1 \leftarrow v_1] \end{array}}{E, F \vdash \text{apply}(\text{p\_list}, \text{call\_list}) : E''}$$

$$\frac{\begin{array}{c} E, F \vdash \text{p\_list} = [\text{lazy } p_1, R_1] \\ E, F \vdash \text{call\_list} = [e_1, R_2] \\ E, F \vdash \text{apply}(R_1, R_2) : E' \\ E'' = E'[p_1 \leftarrow e_1] \end{array}}{E, F \vdash \text{apply}(\text{p\_list}, \text{call\_list}) : E''}$$

$$\frac{}{\text{apply}(\epsilon, \epsilon) : \emptyset}$$

## 3.9 I/O

$$\frac{E, F \vdash e : v}{E, F \vdash \text{print}(e); : E, F,\ \text{print out } v}$$

$$\frac{\begin{array}{c} \text{read string } s \\ \text{s can be parsed to an integer } v \end{array}}{E, F \vdash \text{readInt}(); : E, F, v}$$

$$\frac{\begin{array}{c} \text{read string } s \\ \text{s cannot be parsed to an integer} \end{array}}{E, F \vdash \text{readInt}(); : E, F, 0}$$

$$\frac{\text{read string } s}{E, F \vdash \text{print}(e); : E, F, s}$$

## 4 Abstract Semantics

## 5 Lexer

A lexer for crumbL was written using flex, allowing us to easily define tokens and rules for such tokens. Except for the tokens described in the following subsections, most tokens remained the same in regards to those in L.

### 5.1 Added Tokens

The following tokens do not exist in L and were added to crumbL. Most of these tokens were added for the purpose of making crumbL imperative. Some were added because they allow for useful operations that were not available in L.

TOKEN_ASSIGN - The assign token (=) is used to assign values to variables. This token would have a function similar to TOKEN_EQ from L, which was used in creating let-bindings.

TOKEN_LAZY - The "lazy" token (lazy) is used to indicate if an assignment is lazily evaluated.

TOKEN_SEMICOLON - The semicolon token (;) indicates the end of a statement. Statements were added to make crumbL imperative.

TOKEN_FI - The "fi" token (fi) indicates the end of a conditional block. "fi" is "if" backwards. This method of ending a conditional block is similar to how Bash ends conditional blocks.

TOKEN_FUNC - The "func" token (func) indicates the start of a function block.

TOKEN_CNUF - The "cnuf," "func" backwards, token (cnuf) indicates the end of a function block, similarly to how the "fi" token ends a conditional block.

TOKEN_RET - The "ret" token (ret) is used in a return statement at the end of a function.

TOKEN_WHILE - The "while" token (while) is used to indicate the conditional for a while loop. Loops were added to crumbL to make it more imperative.

TOKEN_DO - The "do" token (do) is used to indicate the start of statements over which the loop will iterate.

TOKEN_OB - The "ob" token (ob) is used to indicate the end of a do block. "ob" is "do" reflected.

TOKEN_NOT - The "not" token (not) is used to negate a boolean. L does not have any way to negate a boolean even though it can be very useful.

TOKEN_MODULO - The modulo token (%) is used for performing the modulo binop operation, which is useful but not included in L.

TOKEN_CONS - The string concatenation token (::) is used to concatenate strings. Like the not and modulo tokens, string concatenation is useful but not a part of L.

## 5.2 Changed Tokens

The following tokens exist in L, but their rules were changed in crumbL for the purpose of making boolean conditionals in crumbL more semantically similar to that of other imperative languages such as Java and C++.

TOKEN_EQ - The equals token in L (=) was used both to create let-bindings and to compare two values. In crumbL, a single equals sign is the assign token. To create a distinction, the equals token in crumbL is two consecutive equals signs (==), reflecting the syntax of that of other imperative languages such as Java and C++.

TOKEN_NEQ - The not equals token in L (<>) seemed a bit unintuitive. The not equals token in crumbL is an exclaimation point followed by an equals sign (!=), reflecting the syntax of that of other imperative languages such as Java and C++.

## 5.3 Removed Tokens

The following tokens exist in L but were removed in crumbL. Most of these tokens were removed due either to the fact that they are more associated with a functional languages.

TOKEN_LAMBDA - Lambda calculus is intergral to L's status as a functional language. However, crumbL is imperative so lambda calculus functionality and tokens associated with it, such as the lambda token (lambda), are no longer necessary.

TOKEN_DOT - The dot token in L (.) is only used to support lambda calculus functionalities so it was removed when developing crumbL.

TOKEN_LET - The "let" token in L (let) is used to indicate let-bindings. Since crumbL is imperative and has assignments, let-bindings, and therefore the let token as well, are no longer necessary.

TOKEN_FUN - Though L and crumbL both have functions, "func" seemed like a more intuitive keyword than "fun" so the "fun" token (fun) from L was removed in favor of the "func" token.

TOKEN_WITH - L used the "with" token (with) to precede arguments during function declarations. In crumbL, function declarations are more similar to that of other imperative languages, such as Java and C++, and use parenthesis to indicate arguments, thus making the "with" token unncecessary.

TOKEN_IN - The "in" token in L (in) is used to indicate movement from one context to another. In crumbL, other tokens, such as the semicolon token, are used to indicate the end of a context so the "in" token was no longer necessary.

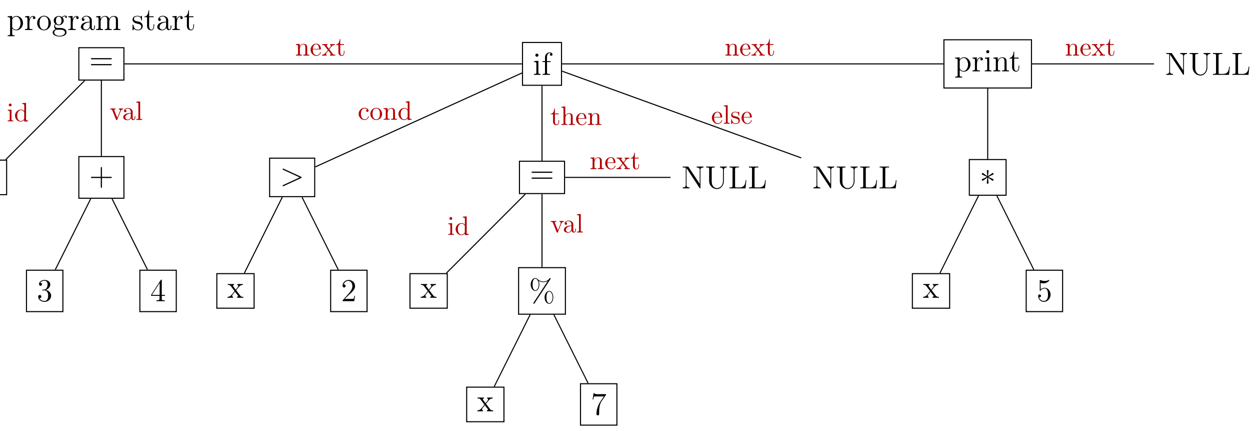
## 6 Parsing

### 6.1 Abstract Syntax Tree

To support our new grammar, we had to come up with a new form for our Abstract Syntax Tree. Now, all of the Expression nodes have a pointer to the next expression statement to evaluate, which is completely separate from the sub-expressions required to evaluate current expression. For instance, this program

```
x = 3 + 4;
if (x > 2) then
          x = x % 7
fi
print(x * 5);
```

will produce the following AST.



Only `Expressions` that are considered "statements" in our language can have `next` pointers that point to a non-NULL `Expression*`, and if they do point to something, they point to other statement nodes.

### 6.2 New Additions to Nodes

In addition to the new `next` pointer that all `Expression` nodes have, we also added several new `Expression` node types to represent our new language features. Since the nature of variables and functions has changed in our language, we decided to remove the existing `AstLet`, `AstLambda`, and `AstExpressionList`, nodes in favor of `AstAssign`, `AstFunc`, and `AstFunctionCall`. These nodes store slightly different information than their L counterparts.

`AstAssign` now records whether or not an assignment was lazy and the expression being assigned. `AstFunc` is drastically revamped from `AstLambda`. Functions in `crumbL` are named, so we don't have lambdas at all. Thus, they store the entire parameter list as opposed to the one formal that they had in L, since we can't have currying without lambdas. We also require that our functions have a return statement at the end, so we also store the return expression in `AstFunc`. Parameter lists in `crumbL` are similar to assignments, so they also allow the parameters to be initialized as lazy, so they also store the laziness of each variable. Both lets and lambdas expect a body in L, which is the place where further expressions are evaluated. However, `crumbL` does not need bodies because all statements move on to the next one, so we removed the body in favor of the more general `next` pointer that they inherit from `Expression`. Expression lists in L are function calls, but we decided to make our own `AstFunctionCall` so we can store the identifier for the function that's being called, as well as an `AstCallList` which represents the list of expressions being passed in as arguments.

Another important node in the AST that we had to add was the `AstWhile` class. This class simply stores the conditional that determines if the loop continues, as well as the loop body.

## 6.3 Using Bison

To actually do the parsing, we used Bison like we did in the previous projects.

### 6.3.1 Finding the First Program Expression

We struggled to handle the main program rule. We wanted the resulting expression that the program outputted to be the statement node represented by the first statement in the entire program, with the `next` pointer pointing to the subsequent statements. Some of our first attempts had issues where the program's AST would end up being the last statement of the program, or the last statement of a nested block, which could appear in a function, a branch statement, and a while loop. Our current approach to this problem is to use the following Bison rule for program.

```
program:
%empty {}
|
program statement {}
```

This construct seemed most effective at preventing shift/reduce conflicts. However, it is a bit weird because it matches the end of the program first, but since the terminal of this statement is in the beginning, it calls the appropriate actions in order of the statements. With the addition of inner programs in the nested blocks, it turns out that the first time the action is called is actually in the deepest inner block of the first inner block. Thus, we set the program's resulting expression to the first resulting AST node if the resulting expression had not been set already. This ensures that this program rule will only set the resulting expression the first time it is matched. In all of the Bison rules involving inner programs, we then check if any of the inner programs evaluated to the global resulting expression, and if it is, we know this block was the first inner block in the program, and we can bubble the resulting expression up to be the current one.

### 6.3.2 Setback with Conditionals

As we worked more on the parser, we discovered that there were some issues with the way we originally defined our grammar. In particular, we tried to define a special "conditional" rule, which consists of boolean expressions like comparison operators, and actual boolean operators like `and`, `or`, and `not`. These would evaluate to an integer where 0 is false and 1 is true. The purpose of this is to restrict the conditional expressions in `if` and `while` statements to have a specific form. However, we figured it would be reasonable to allow the value of conditional expressions be assigned to variables and have them be used as conditional statements as well, but this caused many shift/reduce and reduce/reduce conflicts, so we decided to modify the definition of `crumbL` to allow any non-statement expression to be used in the conditional.

### 6.3.3 Parameter and Call Lists

Another issue we came across was that our original definition of parameter lists and call lists accidentally allowed the construct $\epsilon$, parameter_list, and $\epsilon$, call_list, respectively, introducing a strange comma in the front. Though our parser and interpreter would gracefully handle this case the way it was defined, it was a strange syntax, so we modified it to not allow the $\epsilon$ in the beginning.

### 6.3.4 Removing Repeated Parameters

One interesting consideration we had was that with the new definition of functions, it is possible to have the same variable appear multiple times. This would have some strange semantics, so we decided to make it a parser error by checking the list for duplicate identifier names every time we evaluate a function.

## 7 Interpreter

For the most part, the interpreter has the same form as the L interpreter. The parser constructs the AST of the program and passes it to the Evaluator through the **eval** method. The one major difference that the crumbL interpreter has is that it must evaluate a sequence of expressions instead of only one. Thus, at the end of the **eval** method, the interpreter checks if the current expression has a next expression. If so, it evaluates it before returning.

The **eval** method checks the type of the expression to be evaluated, and then performs the necessary steps to evaluate an expression of that type.

## 7.1 Unary and Binary Operations

For the most part, a unary or binary operation expression directly mirrors the operational semantics. List operations are performed exactly the same way as they are in L. For logical operators, we allow any expression to have a truth value as defined in the operational semantics. Special care must be taken for binary operations to catch and report runtime errors. For the :: operator, we automatically cast integer operands to strings in order to make the print method more useful. Additionally, we allow strings to be compared lexicographically as in C++.

## 7.2 Conditionals

The branch and while loop expression both follow the operational semantics almost exactly, so there is not much to discuss. The while loop is backed by a C++ while loop.

## 7.3 Identifiers and Assignments

The **lazy** keyword makes the evaluation of identifiers and assignments quite interesting.

As an example, consider the following snippet of crumbL:

```
x = 4;
lazy x = x*x;
print(x);
```

After the second line is evaluated, *x* would hold the AST (x*x). print(x) then evaluates x and prints the result, but with a naive implementation of the lazy keyword, evaluating x produces an infinite loop. In general, whenever the AST for an identifier contains itself as a node, evaluation will never terminate.

Thus, it's clear that a correct implementation of lazy assignments must somehow 'save' the previous value of the variable in order to avoid infinite evaluation.

In order to solve this problem, we modified the existing **SymbolTable** class two hold *two* mappings: one for normal assignments, and one for lazy assignments.

Assignments then works as follows:
If a lazy assignment occurs to identifier x, then the expression being assigned is stored in the lazy mapping for identifier x. If a normal assignment occurs to identifier x, then the expression being assigned is evaluated, then stored in the normal mapping for x. Then, any mapping that exists in the lazy mapping is removed.

The evaluation of identifiers then works as follows:
An identifier x is looked up in the symbol table. Lookups in the symbol table still start at the current scope and search up through parents, but at each level the lazy mapping is searched before the normal mapping. If the value for x is found in the lazy mapping, x is

removed from the lazy mapping and then it gets evaluated. The new value is then placed in the normal mapping for x.

Meanwhile, if the value is in the normal mapping, the value is simply returned. The use of two mappings allows the previous code snippet to work as expected, producing the output '16'.

### 7.4 Functions

The interpreter has an extra **SymbolTable** in order to hold a mapping of function names to function bodies, return statements, and parameter lists.

A function definition expression simply adds this information to the table, first checking to see that a function of that name has not already been defined.

Function calls look up all the necessary function information in the function table based on the name of the function. The interpreter then pushes a new mapping onto the symbol table, and assigns each function argument to its parameter in the fresh map. If the parameter is lazy, the argument is not evaluated yet.

After the new environment has been set up, the interpreter then evaluates the body of the function. Finally, the return expression is evaluated and set as the final value of the function call.

We also modified the readString and readInt methods from L to look more like functions in our language by adding parentheses at the end.

## 8 Testing

Testing crumbL involved devising different sample programs that would test the limits of what the language is designed to handle. Through this process various errors were found and quickly fixed in the source code. This section includes sample crumbL programs and the constructs from the language they test.

This test makes sure that when an an expression is assigned to a lazy identifier it is not evaluated immediately. If in this test case the value of y is computed immediately an error will be thrown instead of the program finishing.

```
(* Should print 1 *)
lazy y = 1 + "duck";
print(1);
```

Similar to the previous test, this makes sure that lazy semantics work when an identifier is set to hold the value of a function call. Here, the function call will cause an error if actually evaluated and assigned to the identifier.

```
(* Should print 2 *)
func test1()
  y = 2 + "duck";
  ret 1;
cnuf
lazy y = test1();
```

Here we test that variables have a separation of scope between different blocks of code. When the identifier y inside the method is set to 2 it should not affect the mapping for the similarly named identifier outside the function.

```
(* Should print 3 *)
y = 3;
func test2()
  y = 2;
  ret 1;
cnuf
temp = test2();
print(y);
```

This is just a simple test for the functionality of modulo.

```
(* Should print 4 *)
print(4 % 5);
```

Here we test that while loops function as intended, repeating the body only as long as the conditional holds.

```
(* Should print 5 \n 6 *)
i = 5;
while(i < 7) do
  print(i);
  i = i + 1;
ob
```

This tests the functionality of methods that take in multiple arguments.

```
(* Should print 7 *)
func add(x, y)
  bleh = 3;
  ret x + y;
cnuf
print(add(3, 4));
```

Here we test the "truthiness" of various expressions. Primarily the number 0 and the empty string map to false while any nonzero number and nonempty string map to true.

```
(* Should print 8 *)
if(0) then
  print("wrong");
else
```

```
   if(1) then
         if("true string") then
           if("") then
             print("wrong");
           else
             print(8);
           fi
         else
           print("wrong");
         fi
   else
         print("wrong");
   fi
fi
```

The following tests the functionality of lazy parameters for functions. If working, the first argument x should never be evaluated and an error will never happen even though it's assigned the expression 7 + "duck."

```
(* Should print 9 *)
func testLazyFuncParams(lazy x, y)
  temp = 9;
  ret y + 1;
cnuf
print(testLazyFuncParams(7 + "duck", 8));
```

Here we test that identifiers can be repurposed between lazy and eager semantics. The identifier x starts out eager, changes to lazy holding the expression 7 + "duck" before switching back to eager.

```
(* Should print 10 *)
x = 9;
lazy x = 7 + "duck";
x = 10;
print(x);
```

This tests makes sure that lazy identifiers passed into new scopes can still be evaluated properly. It checks both that the unevaluated expression can still be evaluated in a new scope and that possible sub-identifiers in a lazy expression don't clash with identifiers in the new scope. Here the identifier y is made up of the identifier x which also appears in the function testLazyScope. This doesn't pose a problem as crumbL differentiates between the two.

```
(* Should print 11 *)
func testLazyScope(x)
  temp = 9;
  ret x + 5;
cnuf
```

```
x = 3;
lazy y = x + x;
print(testLazyScope(y));
```

This last test ensures that basic recursion works in crumbL.

```
(* Should print 12 *)
func recur(x)
  if(x == 0) then
        val = 0;
  else
        val = recur(x   1);
  fi
  ret x + val;
cnuf
print(recur(4) + 2);
```