Damo Final Report

Alan Gou	**Abhiroop Gangopadhyay**	**Ian Covert**	**Hari Devaraj**
Tester	Language Guru	Manager	System Architect
ajg2233	ag3661	icc2115	sd2920

May 2017

Contents

Chapter 1

Introduction

1.1 Goal

Our goal is to create a programming language that provides a superior way of modeling mathematical functions. Our system would rely on an underlying graph to capture the dependencies of the many variables and intermediate values that make up a function.

The advantage of such a system is that unlike functions in most programming languages, our functions can be differentiated. That capability is extremely useful in machine learning applications, where algorithms like stochastic gradient descent might require differentiating a loss function with respect to several million tunable parameters.

1.2 Background

```
# z is a function of x and y
def z_func(x, y):
    return x + y

# x is a function of a and b
def x_func(a, b):
    return a * b

# y is a function of c and d
def y_func(c, d):
    return c - d

# Compute z as a function of a, b, c, d
def f1(a, b, c, d):
    x = x_func(a, b) y = y_func(c, d)
    return z_func(x, y)

# Compute z as a function of x, y (bypassing a, b, c, d)
def f2(x, y):
    return z_func(x, y) f1(1, 2, 3, 4)
    # Returns 1 f2(2, -1) # Returns 1
```

The functions f1 and f2 do an adequate job of capturing the desired dependencies between our variables. However, in Python, as in most programming languages, the

functions f1 and f2 could not be differentiated. That functionality is critical for certain use cases, and our language is constructed to support that feature natively.

Our language is inspired by a Python library called Theano, which was developed specifically to provide this kind of functionality. It serves as the back-end in several deep learning libraries, including Lasagne and Keras.

1.2.1 Machine Learning Use Case

To motivate the need for such a language, consider the following scenario. Suppose we're using a machine learning algorithm for which the loss function doesn't have a closed-form solution, as in the case of neural networks. We optimize our parameter choices to best fit the data by iteratively taking the gradient of the loss function and slightly tweaking the model's parameters. The process of taking a small batch of data, computing the gradient of a multinomial loss function, and shifting the parameters in the direction of the gradient is known as stochastic gradient descent, and has the objective of minimizing the loss function.

Our new language offers an easier way to optimize loss functions without a closed form solution by making the process of taking the gradient of a multinomial function a core feature.

Consider how the loss function is constructed: it is a function of both the data (perhaps a subset of the data) and the current values assigned to the tunable parameters. When computing the gradient of the loss function, the developer is interested only in the derivative with respect to the parameters, and not the derivative with respect to the data. In other words, from the perspective of the loss function, the data act as constants. Our language provides features that make this functionality possible.

Chapter 2

Language Tutorial

2.1 Putting Together a Basic Program

We will go through an example in which we create a basic program that calculates the greatest common divisor of two integers.

Damo is meant to be a scripting language. There is no requirement for a main function or some other sort of entry point - statements are generally executed from the top of the file to the bottom.

First, let us define our GCD function.

```
def gcd(int a, int b): int {
    // internal code goes here
}
```

We use the **def** keyword to begin a function declaration. The name of the function follows, and then a pair of parentheses, inside of which are our parameter declarations for **a** and **b**. These are both of type **int** - Damo has two mathematical primitives, which are **ints** (integers) and **nums** (floating-point numbers).

The colon followed by another **int** denote the return type of the function - in this case, our GCD function is supposed to return the GCD, which is an integer. The body of the function is contained inside the curly braces. Now let us look inside the GCD function.

```
def gcd(int a, int b): int {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

We see here that we define a while loop that contains an if-else block. This is where the logical aspects of the GCD function are defined. We have comparison operators that work for integers and numbers. At the end of the function, we return the GCD value we calculate.

Now that we have defined our GCD function, we can print its results.

```
print_int(gcd(2, 14));
print_int(gcd(3, 15));
print_int(gcd(99, 121));
```

This will print out 2, 3, and 11. Our built-in print functions are all type-specific - you cannot use **print_int** to print out a string. The basic **print** function is used only for strings. The complete **gcd.dm** file we have just written looks like this:

```
def gcd(int a, int b): int {
    while (a != b) {
        if (a > b)  {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}

print_int(gcd(2, 14));
print_int(gcd(3, 15));
print_int(gcd(99, 121));
```

Chapter 3

Language Manual

3.1 Identifiers

Identifiers are how we assign names to variables, constants, and other data structures in Damo. Once an identifier has been declared, it cannot be redeclared in any scope.

3.1.1 Declaration

Identifiers can be of arbitrary length, but must always have their type specified during declaration. You do not need to assign a value upon declaration.

```
int x;
x = 15;

int y = 30;
```

They can consist of any sequence of numbers and letters; however, every identifier must begin with a letter and they cannot use underscores because the "_" is reserved for the log function.

```
int validIdentifier = 10;
num 0_invalid_identifier = 7.5;
symbol _another_invalid_identifier;
bool so_many_invalid_identifiers;
```

Every identifier uniquely identifies a resource within its scope. There can only be one identifier with a given name, even in regards to an identifier in a different scope; only one name per program. For example, the following set of statements is invalid:

```
int id = 10;
num id = 10.0;
def id(): void {
    /* function body */
}
```

```
int id = 10;

def alsoForbidden(): void {
    num id = 10.0;
    /* function body */
}
```

3.1.2 Reserved Keywords

Built-in Types

```
int
num
symbol
string
true
false
bool
```

Control Flow

```
if
else
elseif
for
while
```

Function

```
def
return
void
```

3.1.3 Literals

We can declare various types of literals.

int

```
int literalInt = 153;
```

num

```
num literalNum = 17.5;
```

bool

```
bool falseBool = false;
bool trueBool = true;
```

string

```
string literalString = "Hello, world";
```

symbol

```
symbol x;
```

arrays

Damo does not support array literals.

3.1.4 Comments

Damo has single-line comments.

```
// This is a comment
int a = 3; // this is another comment
```

Damo also has multi-line comments.

```
/*
def notUsedFunction: void {
}
*/
```

3.1.5 Whitespace

Whitespace is not significant in Damo. The following two functions are equivalent.

```
def func() : void { if (3 == 3) { print("okay"); } else { print("nope"); }
↪ }


def func() : void {
  if (3 == 3) {
    print("okay"); }
   } else {
    print("nope");
  }
}
```

3.1.6 End of lines

There are four syntactical structures that must end with a semicolon.

1. Data type declarations
2. Identifier assignments
3. Function calls
4. Single expressions

```
int c = 15;
num numericalValue;
foo();
4 + 5;
```

There are three syntactical structures that do not end with a semicolon. These, instead, are bounded by curly braces.

1. If-else statements
2. While and for loops
3. Function declarations

```
for (int i = 0; i < 10; i = i + 1) {
        print_int(i);
}

if (x > 5) {
        print("Greater!");
} else {
        print("Not greater!");
}

def foo() : void {
        /* function body */
}
```

3.2 Data Types

3.2.1 Primitives

- **int** - this is a 4 byte signed integer. You can use them in the following way:

```
    int x = 1;
```

- **num** - this is an 8 byte floating point number. You can use them in the following way:

```
    num y = 3.0;
```

- **bool** - this is a 1 byte boolean and is either *true* or *false*. You can use them in the following way:

```
1    bool t = true;
```

- **string** - this is used to express words and sentences, but they are not represented as a sequence of characters (you cannot access individual characters of the string). You can use them in the following way:

```
1    string z = "What is my purpose?";
```

3.2.2 Composite Data Types

- **array** - holds multiple instances of the same type, and its size must be declared on initialization. You use them in the following way:

```
1    int a[2];
2    a[0] = 1;
3    a[1] = 2;
```

- **symbol** - this is the building block of mathematical functions. It has five members - the operator it contains, the left child, the right child, the value it holds, and a flag that denotes whether it is a constant or not. When instantiated with an expression, it allocates memory to form the nodes of a dependency graph that represents that expression. You can declare them in the following manner:

```
1    symbol a;
2    symbol b;
3    symbol c;
4    a = b + c;
```

3.2.3 Symbols

If a math variable is instantiated with an expression with multiple operators, then the expression will be translated into it's fully parenthesized form according to order of operations, and the top level operation is assigned to the node to the left of the assignment operator. The user can use parentheses to specify which parts of the expression should be evaluation.

For every operation after the first operator, an additional implicit symbol node will be created to represent the value of the operator. If an equation is instantiated with a coefficient, then a symbol node will be created with the coefficient flag set to true and the numerical value will be placed in the node's value member. This allows users to traverse a function graph and avoid mutating nodes that are meant to be coefficients.

Example 1

```
1  symbol a;
2  symbol b;
3  symbol c;
```

```
a = b + c;
// the node a has b as a left pointer and c as a right pointer, with \+"
 ↪  as its operator
// b and c have a as their parent and all other fields are empty
```

Example 2

```
symbol a;
symbol b;
symbol c;
symbol d;
a = b + c * d;
/* Under the hood the expression is converted to the following: a = b + (
 ↪  c * d ) The parent node a has operator \+" and left pointer b and
 ↪  right pointer to a node created under the hood representing the
 ↪  parenthesized expression with a \*" as its operator and c as its left
 ↪  pointer and d as its right pointer.
*/
// the implicitly created node can be accessed as follows: symbol e; e =
 ↪  a.right;
```

Example 3

```
symbol a;
symbol b;
symbol c;
a = 2 * b + c
/* Under the hood, we have a 5 node dependency graph with the node
 ↪  representing a being the root, and isConstant( left(a) ) == true,
 ↪  because the node representing the 2 is a coefficient */
```

Example 4

```
symbol x;
num y;
num z;
y = 2;
z = 3
x = y + z;
/* the x node has a coefficient flag set, and has value set to 5. z and y
 ↪  are both still nums*/
```

3.2.4 Arrays

You cannot assign arrays to other identifiers; additionally arrays cannot be created by array literals. Each index has to be instantiated on its own. Array sizes must be defined from instantiation in the following manner:

```
symbol a[10];

arr[0] = 2;
```

The individual indices can be reassigned with individual members from the class.

3.3 Expressions and Operators

3.3.1 Expressions

Expressions are sequences of operators and operands. To see which sequences of operators and operands are valid, refer to the subsequent parts of this section. The following are example expressions:

```
42; //42
3 ^ 2 + 4 ^ 2; // 25
true or false; // true
```

In examples such as the last one, the order in which expressions are evaluated depends on the precedence of the operators. For additional clarity, the programmer can use parentheses to group expressions:

```
(3 ^ 2) + (4 ^ 2); // 25
(2 ^ (2 + 4)) ^ 2; // 531,441
```

3.3.2 Assignment Operator

The single equals sign (=) is the assignment operator in Damo, and allows values to be associated with identifiers. The value to the left of the assignment operator must be an identifier (not an expression) and the value on the right must be an expression. The following are examples of assignments:

```
int x = 2;
num y = 7.0 / 2.5;
```

There are no shorthand operators for assignments of the following form:

```
x = x + 1;
y = y - 5;
```

3.3.3 Mathematical Operators

We define operators for subtraction, exponentiation, multiplication, division, addition, logarithms, and modulation. Exponent and log operations always return nums.

```
2 _ 4; // 2 - this is the logarithm operator

4 ^ 2; // 16 - this is the exponentiation operator
```

```

3 * 2; // 6 - this is the multiplication operator

21 / 7; // 3 - this is the division operator

1 + 1; // 2 - this is the addition operator

1 - 3; // -2 - this is the subtraction operator

5 % 2 // 1 - this is the modulo operator
```

3.3.4 Comparison Operators

We define operators for comparing values that return boolean values, which are used in boolean expressions for control flow. All primitive types can be compared to another variable of the same type. Composite types are not comparable in that sense.

```
// less than operator
1 < 1; // false
1 < 2; // true

// less than or equal to operator
1 <= 1; // true

// greater than operator
1 > 1; // false
2 > 1; // true

// greater than or equal to
1 >= 1; // true

// equal to
1 == 1; // true
1 == 2; // false
"Hi" == "Hi"; // true

// not equal to
1 != 1; // false
1 != 2; // true
```

3.3.5 Boolean Operators

Boolean operators are used to create expressions where the operands are boolean values. We define boolean operators for **not**, **and**, and **or**.

3.3.6 Array Subscripts

Elements of arrays can be accessed using square brackets. Arrays are zero-index. Arrays must be instantiated before assigning indexes.

When on the left side of an assignment operator, the specified element of the array is mutated. Otherwise, when used in an expression, the element at the specified position is returned.

The following are examples:

```
int[4] arr;
arr[0] = 2;
print_int(arr[0]); // prints 2
```

3.3.7 Function Calls as Expressions

Function calls can be used in expressions, as long as the function returns a value. The following is an example of an expression containing a function:

```
def trivial int () : int {
    return 1;
}
trivial() + 2; // evaluates to 3
```

3.4 Statements

An expression is formed from a combination of operations and operands. A statement can be an assignment, a function call, or any other expression, including control flow statements (such as if-else, while loops, and for loops). Every statement must end with a semicolon, aside from control flow statements. Any expression that ends with a semicolon becomes a statement. Examples include the following:

```
int a = 5;
num b = 5.2;
```

Groups of statements and declarations can be grouped together in curly braces, and will be treated as an expression. It is important to note that in this case, no semicolon is needed after the curly brace. We use curly braces in if-else statements, while statements, and for statements, which we discuss below. Note that every expression in the parentheses below in our discussion of control flow elements (if-else, while, for) must be of type bool (for example, if (int a = 5) is not a valid expression, but if (int a == 5) is fine).

3.4.1 If-Else Statements

We allow for conditional statements with branches in the form of an if-elif-else statement. Formally, the syntax is as follows:

```
if (expression){
    // statement;
} elseif (/* expression */) {
    // statement;
} else {
    // statement;
}
```

3.4.2 While Loops

We also allow while loops. The syntax is as follows:

```
while (expression) {
    statement
}
```

3.4.3 For Loops

Another kind of loop supported by the language is the for loop. The syntax is as follows:

```
for(expression; expression; expression){
    statement;
}
```

The first statement is the initializer. It sets the initial state of the loop. The second expression is the boolean check. This is analogous to the expression in the parentheses of the while loop seen earlier, and ensures that the loop's current state still satisfy the conditions of the loop. The third statement in the parentheses updates the state of the loop. Below is an example:

```
int y = 4;
for(int x=5; x<6; x=x+1) {
    y=y+1;
}
```

The final value of y after this loop runs is 5.

Note that we declared a new variable in the for loop. We could have also declared it outside the for loop and either instantiated it or reassigned it to a new value in the loop. For example, we could have had the following:

```
int y = 4;
int x;
for(x=5; x<6; x=x+1) {
    y=y+1;
}
```

3.5 Functions

3.5.1 User-defined Functions

User-defined functions are declared in the following format:

```
def function_name(arg_type arg_name, arg_type arg_name, ...) : return_type
 ↪ {
    ....
}
```

Below is an example function that returns:

```
def increment (int a) : int {
    a = a+1;
    return a;
}
```

Below is an example function that does not return any value:

```
def print_string(string s) : void {
    print(s);
}
```

3.5.2 Calling Functions

Functions must be declared before they can be called. Functions can be called just by invoking the function name and supplying any parameters.

```
<function-name>(<parameter>)
```

We can also assign variables to the return types of functions, though only if the function has a non-void return type:

```
data_type var_name = function(<parameter>);
```

We can extend the increment example above.

```
int x = 5;
x = increment(5);
// x contains the value 6
```

3.5.3 Standard Library Functions

Our standard library comes with two functions that are used for evaluating and differentiating symbolic equations.

eval

This is used to evaluate a mathematical expression.

```
symbol a;
symbol b;
symbol c;

a = b + c;
b = 1.0;
c = 2;

print_num(eval(b));
print_num(eval(c));
print_num(eval(a));
```

gradient

There is also a function called partialDerivative that allows one to calculate the partial derivative of a mathematical expression.

3.6 Program Structure and Scope

3.6.1 Scope

All functions, data types, and primitives defined in a file are inside the scope of the file. We use lexical scoping. Every file has its own scope.

Variables declared in the file are accessible inside functions that are declared in the same scope.

main.damo

```
int int_value = 5;
int two = 2;
def multiply_by_two(int number) : int {
    return number * two;
}
print_int(multiply_by_two(int_value))
// prints '10'
```

3.6.2 Creation of Scope

We have two types of scopes: a global scope inside of a file, and a scope creating inside of a function. Variables declared inside functions leave their containing scopes when the statement ends. However, a variable defined in a function cannot have the same name as a variable defined in the global scope.

Chapter 4

Project Plan

4.1 Development Process

4.1.1 Project Timeline

- January 25 - begin initial language ideation
- February 7 - Complete Damo proposal
- February 15 - Start LRM
- February 21 - Complete LRM
- March 15 - Planning process begins
- March 20 - Planning process ends
- March 25 - Begin development process
- March 29 - Add numbers and strings to Damo
- April 1 - Create first version of scripting language, add in elseif and in-line assignments
- April 3 - Add in mathematical built-ins such as mod, exponent, and log functions
- April 9 - Fix initialization issues and begin adding arrays
- April 16 - Add in comments
- April 25 - Refactor parser to allow for arbitrary order of declarations and statements
- May 3 - Begin semantic checking
- May 7 - Finish semantic checking
- May 8 - Add in symbol functionality

4.1.2 Role Divisions

Ian provided the initial vision and motivation for the project and acted as the project manager. Abhiroop was our language guru who implemented our semantic checker, while Hari was our code generation expert. Alan managed testing.

4.1.3 Style Guide

We use 2 spaces for indentation, generally. To make our OCaml files readable, we use **ocp-indent**, a command-line tool that outputs a properly-indented copy of an OCaml source file. This, for the most part, kept our code more readable.

4.1.4 Project Environment

We used Vim, Sublime Text, and Visual Studio Code as our primary code editors. We all used virtual machines (VMs) to run versions of Ubuntu 14.04. This was done either in Google Cloud Platform using their Compute Engine instances, through Virtualbox, or using VMWare Fusion. We used OCaml version 4.02.

4.2 Git Log

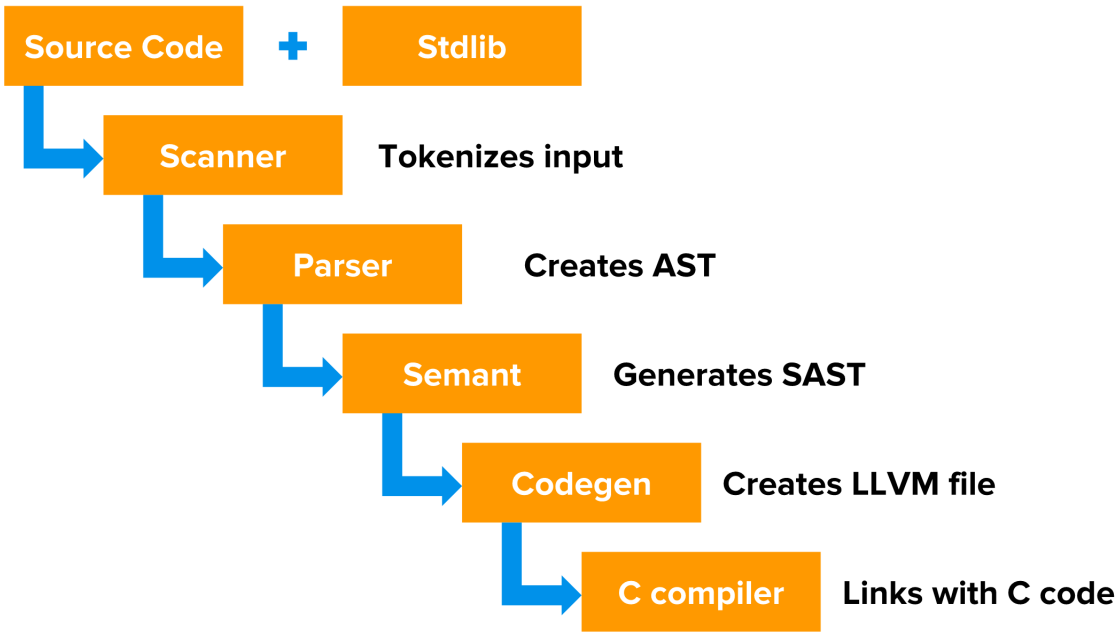
Chapter 5

Architectural Design

5.1 Compilation Process

The Damo compiler targets the LLVM intermediate representation as its target language. The steps proceed as follows:

1. (**stdlib.dm**) Standard library is pre-prended to the source code.
2. (**scanner.mll**) The scanner turns this intermediate **.dm** file into a token stream.
3. (**parser.mly, ast.ml**) The parser creates an Abstract Syntax Tree (AST) from this token stream.
4. (**semant.ml, sast.ml**) The semantic checker takes in the AST and outputs the Semantic AST (SAST), which adds type and scoping information.
5. (**codegen.ml**) The code generator takes this SAST and translates it into an LLVM program, which is a **.ll** file.
6. (**symbol.c, printbig.c**) The LLVM IR file is then turned into assembly by the C compiler, which can be turned into an executable as needed.



5.2 Implementation Responsibilities

Responsibilities were fragmented across different functionalities that extended across individual component boundaries. For example, to implement something like arrays, Abhiroop needed to modify code in the parser, codegen, as well as the semantic checker and the AST. In summary, Ian was responsible for much of the parser and scanner. Alan wrote an initial version of the parser and codegen that allowed for the arbitrary order scripting-language nature of Damo - this was later improved upon by Ian and Abhiroop. Abhiroop wrote the vast majority of our semantic checker and the code generation aspects of arrays. Hari managed much of the details of codegen, such as implicit type-casting, heap allocation for symbol structs, boolean comparisons, and more print functions. Ian wrote out the standard library and the functionality surrounding symbols.

Chapter 6

Test Plan

6.1 Example Programs

6.1.1 Program 1

This is a program to implement Newton's method for finding the root of a function.

Source

```
def univariateNewtonMethod(symbol out, symbol in[], num start, num threshold) : num {
        // Initial iteration
        num x1 = start;
        print(x1);
        in[0] = x1;
        num x2 = x1 - eval(out) / grad(out, in, 1)[0];
        // Repeat iteratively
        while(abs(x1 - x2) > threshold){
                x1 = x2;
                print(x1);
                in[0] = x1;
                x2 = x1 - eval(out) / grad(out, in, 1)[0];
        }
        return x1;
}

// Approximate square root of 10
symbol a; symbol b;
a = b ^ 2 - 10;
num partials1[1];
partials[0] = b;
num root = univariateNewtonMethod(a, partials1, 1, 0.00001);
print("Square root of 10");
print(root);

// Locate root of cubic polynomial
a = b ^ 3 - b + 1;
num root = univariateNewtonMethod(a, partials1, -1, 0.00001);
print("Root of polynomial");
print(root);
```

6.1.2 IR

```
; ModuleID = 'damo'

@exponentialConstant = global double 0.000000e+00
@a = global i8* null
@b = global i8* null
@root = global double 0.000000e+00
@fmtint = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmtstr = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@floatstr = private unnamed_addr constant [4 x i8] c"%f\0A\00"
@tmp = private unnamed_addr constant [4 x i8] c"EXP\00"
@tmp1 = private unnamed_addr constant [6 x i8] c"MINUS\00"
@tmp2 = private unnamed_addr constant [19 x i8] c"Square root of 10:\00"
@tmp3 = private unnamed_addr constant [4 x i8] c"EXP\00"
@tmp4 = private unnamed_addr constant [6 x i8] c"MINUS\00"
@tmp5 = private unnamed_addr constant [5 x i8] c"PLUS\00"
@tmp6 = private unnamed_addr constant [20 x i8] c"Root of polynomial:\00"
@fmtint7 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmtstr8 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@floatstr9 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
@fmtint10 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmtstr11 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@floatstr12 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
@tmp13 = private unnamed_addr constant [35 x i8] c"Evaluating an uninitialized symbol\00"
@tmp14 = private unnamed_addr constant [5 x i8] c"PLUS\00"
@tmp15 = private unnamed_addr constant [6 x i8] c"MINUS\00"
@tmp16 = private unnamed_addr constant [6 x i8] c"TIMES\00"
@tmp17 = private unnamed_addr constant [7 x i8] c"DIVIDE\00"
@tmp18 = private unnamed_addr constant [4 x i8] c"EXP\00"
@tmp19 = private unnamed_addr constant [4 x i8] c"LOG\00"
@tmp20 = private unnamed_addr constant [17 x i8] c"Unknown operator\00"
@fmtint21 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmtstr22 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@floatstr23 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
@tmp24 = private unnamed_addr constant [49 x i8] c"Attempting to differentiate uninitiali
@tmp25 = private unnamed_addr constant [5 x i8] c"PLUS\00"
@tmp26 = private unnamed_addr constant [6 x i8] c"MINUS\00"
@tmp27 = private unnamed_addr constant [6 x i8] c"TIMES\00"
@tmp28 = private unnamed_addr constant [7 x i8] c"DIVIDE\00"
@tmp29 = private unnamed_addr constant [4 x i8] c"EXP\00"
@tmp30 = private unnamed_addr constant [4 x i8] c"LOG\00"
@tmp31 = private unnamed_addr constant [65 x i8] c"Should crash here - invalid symbol ope
@fmtint32 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmtstr33 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@floatstr34 = private unnamed_addr constant [4 x i8] c"%f\0A\00"

declare i32 @printf(i8*, ...)

declare i32 @strcmp(i8*, i8*)

```

```
50  declare i32 @abs(i32)
51
52  declare double @fabs(double)
53
54  declare i8* @operator(i8*)
55
56  declare i8* @createSymbol()
57
58  declare i32 @isConstant(i8*)
59
60  declare i8* @createRoot(i8*, i8*, i8*)
61
62  declare i8* @setSymbolValue(i8*, double)
63
64  declare double @value(i8*)
65
66  declare i32 @isInitialized(i8*)
67
68  declare i8* @left(i8*)
69
70  declare i8* @right(i8*)
71
72  declare double @pow(double, double)
73
74  declare double @log(double)
75
76  declare i32 @printbig(i32)
77
78  define i32 @main() {
79  entry:
80    %symbolmal = call i8* @createSymbol()
81    store i8* %symbolmal, i8** @a
82    %symbolmal1 = call i8* @createSymbol()
83    store i8* %symbolmal1, i8** @b
84    store double 2.718280e+00, double* @exponentialConstant
85    %b = load i8** @b
86    %symbolmal2 = call i8* @createSymbol()
87    %symbolm = call i8* @setSymbolValue(i8* %symbolmal2, double 2.000000e+00)
88    %symbolm3 = call i8* @createRoot(i8* %b, i8* %symbolm, i8* getelementptr inbounds ([4 x
89    %symbolmal4 = call i8* @createSymbol()
90    %symbolm5 = call i8* @setSymbolValue(i8* %symbolmal4, double 1.000000e+01)
91    %symbolm6 = call i8* @createRoot(i8* %symbolm3, i8* %symbolm5, i8* getelementptr inboun
92    store i8* %symbolm6, i8** @a
93    %b7 = load i8** @b
94    %a = load i8** @a
95    %univariateNewtonMethod_result = call double @univariateNewtonMethod(i8* %a, i8* %b7, d
96    store double %univariateNewtonMethod_result, double* @root
97    %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmtstr, i
98    %root = load double* @root
99    %printf8 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatstr
100   %b9 = load i8** @b
```

```
101   %symbolmal10 = call i8* @createSymbol()
102   %symbolm11 = call i8* @setSymbolValue(i8* %symbolmal10, double 3.000000e+00)
103   %symbolm12 = call i8* @createRoot(i8* %b9, i8* %symbolm11, i8* getelementptr inbounds (
104   %b13 = load i8** @b
105   %symbolm14 = call i8* @createRoot(i8* %symbolm12, i8* %b13, i8* getelementptr inbounds
106   %symbolmal15 = call i8* @createSymbol()
107   %symbolm16 = call i8* @setSymbolValue(i8* %symbolmal15, double 1.000000e+00)
108   %symbolm17 = call i8* @createRoot(i8* %symbolm14, i8* %symbolm16, i8* getelementptr inb
109   store i8* %symbolm17, i8** @a
110   %b18 = load i8** @b
111   %a19 = load i8** @a
112   %univariateNewtonMethod_result20 = call double @univariateNewtonMethod(i8* %a19, i8* %b
113   store double %univariateNewtonMethod_result20, double* @root
114   %printf21 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmtstr,
115   %root22 = load double* @root
116   %printf23 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatst
117   ret i32 0
118 }
119
120 define i1 @streq(i8* %a, i8* %b) {
121 entry:
122   %a1 = alloca i8*
123   store i8* %a, i8** %a1
124   %b2 = alloca i8*
125   store i8* %b, i8** %b2
126   %b3 = load i8** %b2
127   %a4 = load i8** %a1
128   %strcmp = call i32 @strcmp(i8* %a4, i8* %b3)
129   %tmp = icmp eq i32 %strcmp, 0
130   ret i1 %tmp
131 }
132
133 define double @eval(i8* %a) {
134 entry:
135   %a1 = alloca i8*
136   store i8* %a, i8** %a1
137   %leftValue = alloca double
138   %rightValue = alloca double
139   %result = alloca double
140   %op = alloca i8*
141   %a2 = load i8** %a1
142   %symbol_const = call i32 @isConstant(i8* %a2)
143   %tmp = icmp eq i32 %symbol_const, 1
144   br i1 %tmp, label %then, label %else
145
146 merge:                                            ; preds = %merge6, %then
147   %result91 = load double* %result
148   ret double %result91
149
150 then:                                             ; preds = %entry
151   %a3 = load i8** %a1
```

```
152   %symbol_value = call double @value(i8* %a3)
153   store double %symbol_value, double* %result
154   br label %merge
155
156 else:                                             ; preds = %entry
157   %a4 = load i8** %a1
158   %symbol_init = call i32 @isInitialized(i8* %a4)
159   %tmp5 = icmp ne i32 %symbol_init, 1
160   br i1 %tmp5, label %then7, label %else8
161
162 merge6:                                           ; preds = %merge11, %then7
163   br label %merge
164
165 then7:                                            ; preds = %else
166   %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmtstr11,
167   br label %merge6
168
169 else8:                                            ; preds = %else
170   %a9 = load i8** %a1
171   %symbol_operator = call i8* @operator(i8* %a9)
172   store i8* %symbol_operator, i8** %op
173   %op10 = load i8** %op
174   %streq_result = call i1 @streq(i8* %op10, i8* getelementptr inbounds ([5 x i8]* @tmp14,
175   br i1 %streq_result, label %then12, label %else19
176
177 merge11:                                          ; preds = %merge22, %then12
178   br label %merge6
179
180 then12:                                           ; preds = %else8
181   %a13 = load i8** %a1
182   %symbol_left_call = call i8* @left(i8* %a13)
183   %eval_result = call double @eval(i8* %symbol_left_call)
184   store double %eval_result, double* %leftValue
185   %a14 = load i8** %a1
186   %symbol_right_call = call i8* @right(i8* %a14)
187   %eval_result15 = call double @eval(i8* %symbol_right_call)
188   store double %eval_result15, double* %rightValue
189   %leftValue16 = load double* %leftValue
190   %rightValue17 = load double* %rightValue
191   %tmp18 = fadd double %leftValue16, %rightValue17
192   store double %tmp18, double* %result
193   br label %merge11
194
195 else19:                                           ; preds = %else8
196   %op20 = load i8** %op
197   %streq_result21 = call i1 @streq(i8* %op20, i8* getelementptr inbounds ([6 x i8]* @tmp1
198   br i1 %streq_result21, label %then23, label %else33
199
200 merge22:                                          ; preds = %merge36, %then23
201   br label %merge11
202
```

```
203 then23:                                           ; preds = %else19
204   %a24 = load i8** %a1
205   %symbol_left_call25 = call i8* @left(i8* %a24)
206   %eval_result26 = call double @eval(i8* %symbol_left_call25)
207   store double %eval_result26, double* %leftValue
208   %a27 = load i8** %a1
209   %symbol_right_call28 = call i8* @right(i8* %a27)
210   %eval_result29 = call double @eval(i8* %symbol_right_call28)
211   store double %eval_result29, double* %rightValue
212   %leftValue30 = load double* %leftValue
213   %rightValue31 = load double* %rightValue
214   %tmp32 = fsub double %leftValue30, %rightValue31
215   store double %tmp32, double* %result
216   br label %merge22
217
218 else33:                                           ; preds = %else19
219   %op34 = load i8** %op
220   %streq_result35 = call i1 @streq(i8* %op34, i8* getelementptr inbounds ([6 x i8]* @tmp1
221   br i1 %streq_result35, label %then37, label %else47
222
223 merge36:                                          ; preds = %merge50, %then37
224   br label %merge22
225
226 then37:                                           ; preds = %else33
227   %a38 = load i8** %a1
228   %symbol_left_call39 = call i8* @left(i8* %a38)
229   %eval_result40 = call double @eval(i8* %symbol_left_call39)
230   store double %eval_result40, double* %leftValue
231   %a41 = load i8** %a1
232   %symbol_right_call42 = call i8* @right(i8* %a41)
233   %eval_result43 = call double @eval(i8* %symbol_right_call42)
234   store double %eval_result43, double* %rightValue
235   %leftValue44 = load double* %leftValue
236   %rightValue45 = load double* %rightValue
237   %tmp46 = fmul double %leftValue44, %rightValue45
238   store double %tmp46, double* %result
239   br label %merge36
240
241 else47:                                           ; preds = %else33
242   %op48 = load i8** %op
243   %streq_result49 = call i1 @streq(i8* %op48, i8* getelementptr inbounds ([7 x i8]* @tmp1
244   br i1 %streq_result49, label %then51, label %else61
245
246 merge50:                                          ; preds = %merge64, %then51
247   br label %merge36
248
249 then51:                                           ; preds = %else47
250   %a52 = load i8** %a1
251   %symbol_left_call53 = call i8* @left(i8* %a52)
252   %eval_result54 = call double @eval(i8* %symbol_left_call53)
253   store double %eval_result54, double* %leftValue
```

```
254   %a55 = load i8** %a1
255   %symbol_right_call56 = call i8* @right(i8* %a55)
256   %eval_result57 = call double @eval(i8* %symbol_right_call56)
257   store double %eval_result57, double* %rightValue
258   %leftValue58 = load double* %leftValue
259   %rightValue59 = load double* %rightValue
260   %tmp60 = fdiv double %leftValue58, %rightValue59
261   store double %tmp60, double* %result
262   br label %merge50
263
264 else61:                                           ; preds = %else47
265   %op62 = load i8** %op
266   %streq_result63 = call i1 @streq(i8* %op62, i8* getelementptr inbounds ([4 x i8]* @tmp1
267   br i1 %streq_result63, label %then65, label %else74
268
269 merge64:                                          ; preds = %merge77, %then65
270   br label %merge50
271
272 then65:                                           ; preds = %else61
273   %a66 = load i8** %a1
274   %symbol_left_call67 = call i8* @left(i8* %a66)
275   %eval_result68 = call double @eval(i8* %symbol_left_call67)
276   store double %eval_result68, double* %leftValue
277   %a69 = load i8** %a1
278   %symbol_right_call70 = call i8* @right(i8* %a69)
279   %eval_result71 = call double @eval(i8* %symbol_right_call70)
280   store double %eval_result71, double* %rightValue
281   %leftValue72 = load double* %leftValue
282   %rightValue73 = load double* %rightValue
283   %pow_func = call double @pow(double %leftValue72, double %rightValue73)
284   store double %pow_func, double* %result
285   br label %merge64
286
287 else74:                                           ; preds = %else61
288   %op75 = load i8** %op
289   %streq_result76 = call i1 @streq(i8* %op75, i8* getelementptr inbounds ([4 x i8]* @tmp1
290   br i1 %streq_result76, label %then78, label %else89
291
292 merge77:                                          ; preds = %else89, %then78
293   br label %merge64
294
295 then78:                                           ; preds = %else74
296   %a79 = load i8** %a1
297   %symbol_left_call80 = call i8* @left(i8* %a79)
298   %eval_result81 = call double @eval(i8* %symbol_left_call80)
299   store double %eval_result81, double* %leftValue
300   %a82 = load i8** %a1
301   %symbol_right_call83 = call i8* @right(i8* %a82)
302   %eval_result84 = call double @eval(i8* %symbol_right_call83)
303   store double %eval_result84, double* %rightValue
304   %leftValue85 = load double* %leftValue
```

```
305   %rightValue86 = load double* %rightValue
306   %log_func = call double @log(double %rightValue86)
307   %log_func87 = call double @log(double %leftValue85)
308   %tmp88 = fdiv double %log_func, %log_func87
309   store double %tmp88, double* %result
310   br label %merge77
311
312 else89:                                           ; preds = %else74
313   %printf90 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmtstr1
314   br label %merge77
315 }
316
317 define double @partialDerivative(i8* %out, i8* %in) {
318 entry:
319   %out1 = alloca i8*
320   store i8* %out, i8** %out1
321   %in2 = alloca i8*
322   store i8* %in, i8** %in2
323   %leftGrad = alloca double
324   %rightGrad = alloca double
325   %op = alloca i8*
326   %dadL = alloca double
327   %dadR = alloca double
328   %L = alloca double
329   %R = alloca double
330   %result = alloca double
331   store double 0.000000e+00, double* %dadL
332   store double 0.000000e+00, double* %dadR
333   %out3 = load i8** %out1
334   %symbol_const = call i32 @isConstant(i8* %out3)
335   %tmp = icmp eq i32 %symbol_const, 1
336   br i1 %tmp, label %then, label %else9
337
338 merge:                                            ; preds = %merge12, %merge7
339   %result150 = load double* %result
340   ret double %result150
341
342 then:                                             ; preds = %entry
343   %out4 = load i8** %out1
344   %in5 = load i8** %in2
345   %e2 = ptrtoint i8* %in5 to i32
346   %e1 = ptrtoint i8* %out4 to i32
347   %tmp6 = icmp eq i32 %e1, %e2
348   br i1 %tmp6, label %then8, label %else
349
350 merge7:                                           ; preds = %else, %then8
351   br label %merge
352
353 then8:                                            ; preds = %then
354   store double 1.000000e+00, double* %result
355   br label %merge7
```

```
356
357 else:                                           ; preds = %then
358   store double 0.000000e+00, double* %result
359   br label %merge7
360
361 else9:                                          ; preds = %entry
362   %out10 = load i8** %out1
363   %symbol_init = call i32 @isInitialized(i8* %out10)
364   %tmp11 = icmp ne i32 %symbol_init, 1
365   br i1 %tmp11, label %then13, label %else14
366
367 merge12:                                        ; preds = %merge16, %then13
368   br label %merge
369
370 then13:                                         ; preds = %else9
371   %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmtstr22,
372   br label %merge12
373
374 else14:                                         ; preds = %else9
375   %out15 = load i8** %out1
376   %symbol_operator = call i8* @operator(i8* %out15)
377   store i8* %symbol_operator, i8** %op
378   br i1 false, label %then17, label %else23
379
380 merge16:                                        ; preds = %merge32, %then17
381   br label %merge12
382
383 then17:                                         ; preds = %else14
384   %in18 = load i8** %in2
385   %out19 = load i8** %out1
386   %symbol_left_call = call i8* @left(i8* %out19)
387   %partialDerivative_result = call double @partialDerivative(i8* %symbol_left_call, i8* %
388   store double %partialDerivative_result, double* %leftGrad
389   store double -1.000000e+00, double* %dadL
390   %dadL20 = load double* %dadL
391   %leftGrad21 = load double* %leftGrad
392   %tmp22 = fmul double %dadL20, %leftGrad21
393   store double %tmp22, double* %result
394   br label %merge16
395
396 else23:                                         ; preds = %else14
397   %in24 = load i8** %in2
398   %out25 = load i8** %out1
399   %symbol_left_call26 = call i8* @left(i8* %out25)
400   %partialDerivative_result27 = call double @partialDerivative(i8* %symbol_left_call26, i
401   store double %partialDerivative_result27, double* %leftGrad
402   %in28 = load i8** %in2
403   %out29 = load i8** %out1
404   %symbol_right_call = call i8* @right(i8* %out29)
405   %partialDerivative_result30 = call double @partialDerivative(i8* %symbol_right_call, i8
406   store double %partialDerivative_result30, double* %rightGrad
```

```
407   %op31 = load i8** %op
408   %streq_result = call i1 @streq(i8* %op31, i8* getelementptr inbounds ([5 x i8]* @tmp25,
409   br i1 %streq_result, label %then33, label %else34
410
411 merge32:                                          ; preds = %merge37, %then33
412   %dadL143 = load double* %dadL
413   %leftGrad144 = load double* %leftGrad
414   %tmp145 = fmul double %dadL143, %leftGrad144
415   %dadR146 = load double* %dadR
416   %rightGrad147 = load double* %rightGrad
417   %tmp148 = fmul double %dadR146, %rightGrad147
418   %tmp149 = fadd double %tmp145, %tmp148
419   store double %tmp149, double* %result
420   br label %merge16
421
422 then33:                                           ; preds = %else23
423   store double 1.000000e+00, double* %dadL
424   store double 1.000000e+00, double* %dadR
425   br label %merge32
426
427 else34:                                           ; preds = %else23
428   %op35 = load i8** %op
429   %streq_result36 = call i1 @streq(i8* %op35, i8* getelementptr inbounds ([6 x i8]* @tmp2
430   br i1 %streq_result36, label %then38, label %else39
431
432 merge37:                                          ; preds = %merge42, %then38
433   br label %merge32
434
435 then38:                                           ; preds = %else34
436   store double 1.000000e+00, double* %dadL
437   store double -1.000000e+00, double* %dadR
438   br label %merge37
439
440 else39:                                           ; preds = %else34
441   %op40 = load i8** %op
442   %streq_result41 = call i1 @streq(i8* %op40, i8* getelementptr inbounds ([6 x i8]* @tmp2
443   br i1 %streq_result41, label %then43, label %else49
444
445 merge42:                                          ; preds = %merge52, %then43
446   br label %merge37
447
448 then43:                                           ; preds = %else39
449   %out44 = load i8** %out1
450   %symbol_right_call45 = call i8* @right(i8* %out44)
451   %eval_result = call double @eval(i8* %symbol_right_call45)
452   store double %eval_result, double* %dadL
453   %out46 = load i8** %out1
454   %symbol_left_call47 = call i8* @left(i8* %out46)
455   %eval_result48 = call double @eval(i8* %symbol_left_call47)
456   store double %eval_result48, double* %dadR
457   br label %merge42
```

```
458
459 else49:                                           ; preds = %else39
460   %op50 = load i8** %op
461   %streq_result51 = call i1 @streq(i8* %op50, i8* getelementptr inbounds ([7 x i8]* @tmp2
462   br i1 %streq_result51, label %then53, label %else66
463
464 merge52:                                          ; preds = %merge69, %then53
465   br label %merge42
466
467 then53:                                           ; preds = %else49
468   %out54 = load i8** %out1
469   %symbol_right_call55 = call i8* @right(i8* %out54)
470   %eval_result56 = call double @eval(i8* %symbol_right_call55)
471   %tmp57 = fdiv double 1.000000e+00, %eval_result56
472   store double %tmp57, double* %dadL
473   %out58 = load i8** %out1
474   %symbol_left_call59 = call i8* @left(i8* %out58)
475   %eval_result60 = call double @eval(i8* %symbol_left_call59)
476   %out61 = load i8** %out1
477   %symbol_right_call62 = call i8* @right(i8* %out61)
478   %eval_result63 = call double @eval(i8* %symbol_right_call62)
479   %pow_func = call double @pow(double %eval_result63, double 2.000000e+00)
480   %tmp64 = fdiv double %eval_result60, %pow_func
481   %tmp65 = fsub double 0.000000e+00, %tmp64
482   store double %tmp65, double* %dadR
483   br label %merge52
484
485 else66:                                           ; preds = %else49
486   %op67 = load i8** %op
487   %streq_result68 = call i1 @streq(i8* %op67, i8* getelementptr inbounds ([4 x i8]* @tmp2
488   br i1 %streq_result68, label %then70, label %else100
489
490 merge69:                                          ; preds = %merge103, %merge90
491   br label %merge52
492
493 then70:                                           ; preds = %else66
494   %out71 = load i8** %out1
495   %symbol_left_call72 = call i8* @left(i8* %out71)
496   %eval_result73 = call double @eval(i8* %symbol_left_call72)
497   store double %eval_result73, double* %L
498   %out74 = load i8** %out1
499   %symbol_right_call75 = call i8* @right(i8* %out74)
500   %eval_result76 = call double @eval(i8* %symbol_right_call75)
501   store double %eval_result76, double* %R
502   %leftGrad77 = load double* %leftGrad
503   %tmp78 = fcmp one double %leftGrad77, 0.000000e+00
504   br i1 %tmp78, label %then80, label %else87
505
506 merge79:                                          ; preds = %else87, %then80
507   %rightGrad88 = load double* %rightGrad
508   %tmp89 = fcmp one double %rightGrad88, 0.000000e+00
```

```
509   br i1 %tmp89, label %then91, label %else99
510
511 then80:                                             ; preds = %then70
512   %R81 = load double* %R
513   %L82 = load double* %L
514   %R83 = load double* %R
515   %tmp84 = fsub double %R83, 1.000000e+00
516   %pow_func85 = call double @pow(double %L82, double %tmp84)
517   %tmp86 = fmul double %R81, %pow_func85
518   store double %tmp86, double* %dadL
519   br label %merge79
520
521 else87:                                             ; preds = %then70
522   br label %merge79
523
524 merge90:                                            ; preds = %else99, %then91
525   br label %merge69
526
527 then91:                                             ; preds = %merge79
528   %exponentialConstant = load double* @exponentialConstant
529   %L92 = load double* %L
530   %log_func = call double @log(double %L92)
531   %log_func93 = call double @log(double %exponentialConstant)
532   %tmp94 = fdiv double %log_func, %log_func93
533   %L95 = load double* %L
534   %R96 = load double* %R
535   %pow_func97 = call double @pow(double %L95, double %R96)
536   %tmp98 = fmul double %tmp94, %pow_func97
537   store double %tmp98, double* %dadR
538   br label %merge90
539
540 else99:                                             ; preds = %merge79
541   br label %merge90
542
543 else100:                                            ; preds = %else66
544   %op101 = load i8** %op
545   %streq_result102 = call i1 @streq(i8* %op101, i8* getelementptr inbounds ([4 x i8]* @tm
546   br i1 %streq_result102, label %then104, label %else141
547
548 merge103:                                           ; preds = %else141, %merge131
549   br label %merge69
550
551 then104:                                            ; preds = %else100
552   %out105 = load i8** %out1
553   %symbol_left_call106 = call i8* @left(i8* %out105)
554   %eval_result107 = call double @eval(i8* %symbol_left_call106)
555   store double %eval_result107, double* %L
556   %out108 = load i8** %out1
557   %symbol_right_call109 = call i8* @right(i8* %out108)
558   %eval_result110 = call double @eval(i8* %symbol_right_call109)
559   store double %eval_result110, double* %R
```

```
560   %leftGrad111 = load double* %leftGrad
561   %tmp112 = fcmp one double %leftGrad111, 0.000000e+00
562   br i1 %tmp112, label %then114, label %else128
563
564 merge113:                                         ; preds = %else128, %then114
565   %rightGrad129 = load double* %rightGrad
566   %tmp130 = fcmp one double %rightGrad129, 0.000000e+00
567   br i1 %tmp130, label %then132, label %else140
568
569 then114:                                          ; preds = %then104
570   %L115 = load double* %L
571   %R116 = load double* %R
572   %log_func117 = call double @log(double %R116)
573   %log_func118 = call double @log(double %L115)
574   %tmp119 = fdiv double %log_func117, %log_func118
575   %L120 = load double* %L
576   %exponentialConstant121 = load double* @exponentialConstant
577   %log_func122 = call double @log(double %exponentialConstant121)
578   %log_func123 = call double @log(double %L120)
579   %tmp124 = fdiv double %log_func122, %log_func123
580   %tmp125 = fmul double %tmp119, %tmp124
581   %L126 = load double* %L
582   %tmp127 = fdiv double %tmp125, %L126
583   store double %tmp127, double* %dadL
584   br label %merge113
585
586 else128:                                          ; preds = %then104
587   br label %merge113
588
589 merge131:                                         ; preds = %else140, %then132
590   br label %merge103
591
592 then132:                                          ; preds = %merge113
593   %L133 = load double* %L
594   %exponentialConstant134 = load double* @exponentialConstant
595   %log_func135 = call double @log(double %exponentialConstant134)
596   %log_func136 = call double @log(double %L133)
597   %tmp137 = fdiv double %log_func135, %log_func136
598   %R138 = load double* %R
599   %tmp139 = fdiv double %tmp137, %R138
600   store double %tmp139, double* %dadR
601   br label %merge131
602
603 else140:                                          ; preds = %merge113
604   br label %merge131
605
606 else141:                                          ; preds = %else100
607   %printf142 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmtstr
608   br label %merge103
609 }
610
```

```
611 define double @univariateNewtonMethod(i8* %out, i8* %in, double %start, double %threshold
612 entry:
613   %out1 = alloca i8*
614   store i8* %out, i8** %out1
615   %in2 = alloca i8*
616   store i8* %in, i8** %in2
617   %start3 = alloca double
618   store double %start, double* %start3
619   %threshold4 = alloca double
620   store double %threshold, double* %threshold4
621   %x1 = alloca double
622   %result = alloca double
623   %deriv = alloca double
624   %x2 = alloca double
625   %start5 = load double* %start3
626   store double %start5, double* %x1
627   %x16 = load double* %x1
628   %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatstr3
629   %x17 = load double* %x1
630   %in8 = load i8** %in2
631   %symbolm = call i8* @setSymbolValue(i8* %in8, double %x17)
632   store i8* %symbolm, i8** %in2
633   %out9 = load i8** %out1
634   %eval_result = call double @eval(i8* %out9)
635   store double %eval_result, double* %result
636   %in10 = load i8** %in2
637   %out11 = load i8** %out1
638   %partialDerivative_result = call double @partialDerivative(i8* %out11, i8* %in10)
639   store double %partialDerivative_result, double* %deriv
640   %result12 = load double* %result
641   %printf13 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatst
642   %deriv14 = load double* %deriv
643   %printf15 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatst
644   %x116 = load double* %x1
645   %result17 = load double* %result
646   %deriv18 = load double* %deriv
647   %tmp = fdiv double %result17, %deriv18
648   %tmp19 = fsub double %x116, %tmp
649   store double %tmp19, double* %x2
650   br label %while
651
652 while:                                            ; preds = %while_body, %entry
653   %x136 = load double* %x1
654   %x237 = load double* %x2
655   %tmp38 = fsub double %x136, %x237
656   %absnum = call double @fabs(double %tmp38)
657   %threshold39 = load double* %threshold4
658   %tmp40 = fcmp ogt double %absnum, %threshold39
659   br i1 %tmp40, label %while_body, label %merge
660
661 while_body:                                       ; preds = %while
```

```
662   %x220 = load double* %x2
663   store double %x220, double* %x1
664   %x121 = load double* %x1
665   %printf22 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatst
666   %x123 = load double* %x1
667   %in24 = load i8** %in2
668   %symbolm25 = call i8* @setSymbolValue(i8* %in24, double %x123)
669   store i8* %symbolm25, i8** %in2
670   %out26 = load i8** %out1
671   %eval_result27 = call double @eval(i8* %out26)
672   store double %eval_result27, double* %result
673   %in28 = load i8** %in2
674   %out29 = load i8** %out1
675   %partialDerivative_result30 = call double @partialDerivative(i8* %out29, i8* %in28)
676   store double %partialDerivative_result30, double* %deriv
677   %x131 = load double* %x1
678   %result32 = load double* %result
679   %deriv33 = load double* %deriv
680   %tmp34 = fdiv double %result32, %deriv33
681   %tmp35 = fsub double %x131, %tmp34
682   store double %tmp35, double* %x2
683   br label %while
684
685 merge:                                            ; preds = %while
686   %x141 = load double* %x1
687   ret double %x141
688 }
```

6.1.3 Program 2

This is a function to showcase gradient descent.

Source

```
1  // Example 1
2  symbol a;
3
4  symbol partials1[1];
5  partials1[0] = a;
6  num grad[];
7  grad = gradient(a, partials, 1);
8
9  print(grad[0]);
10
11 // Example 2
12 symbol b;
13 symbol c;
14
15 a = b * c;
16 b = 4;
17 c = 5;
```

```
18
19 symbol partials[2];
20 partials2[0] = b; partials2[1] = c;
21 grad = gradient(a, partials2, 2);
22
23 print(grad[0]); print(grad[1]);
24
25 // Example 3
26 symbol d; symbol e; symbol f; symbol g;
27
28 g = a / d;
29 d = e _ f;
30 e = 5;
31 f = 25;
32
33 symbol partials3[4];
34 partials3[0] = b; partials3[1] = c; partials3[2] = e; partials3[3] = f;
35 grad = gradient(g, partials3, 4);
36
37 print(grad[0]); print(grad[1]); print(grad[2]); print(grad[3]);
```

IR

```
1  ; ModuleID = 'damo'
2
3  @exponentialConstant = global double 0.000000e+00
4  @a = global i8* null
5  @b = global i8* null
6  @c = global i9* null
7  @d = global i8* null
8  @e = global i8* null
9  @f = global i8* null
10 @g = global i8* null
11 @fmtint = private unnamed_addr constant [4 x i8] c"%d\0A\00"
12 @fmtstr = private unnamed_addr constant [4 x i8] c"%s\0A\00"
13 @floatstr = private unnamed_addr constant [4 x i8] c"%f\0A\00"
14 @tmp = private unnamed_addr constant [5 x i8] c"PLUS\00"
15 @tmp1 = private unnamed_addr constant [6 x i8] c"TIMES\00"
16 @tmp2 = private unnamed_addr constant [4 x i8] c"EXP\00"
17 @fmtint3 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
18 @fmtstr4 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
19 @floatstr5 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
20 @fmtint6 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
21 @fmtstr7 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
22 @floatstr8 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
23 @tmp9 = private unnamed_addr constant [35 x i8] c"Evaluating an uninitialized symbol\00"
24 @tmp10 = private unnamed_addr constant [5 x i8] c"PLUS\00"
25 @tmp11 = private unnamed_addr constant [6 x i8] c"MINUS\00"
26 @tmp12 = private unnamed_addr constant [6 x i8] c"TIMES\00"
27 @tmp13 = private unnamed_addr constant [7 x i8] c"DIVIDE\00"
28 @tmp14 = private unnamed_addr constant [4 x i8] c"EXP\00"
```

```
29 @tmp15 = private unnamed_addr constant [4 x i8] c"LOG\00"
30 @tmp16 = private unnamed_addr constant [17 x i8] c"Unknown operator\00"
31 @fmtint17 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
32 @fmtstr18 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
33 @floatstr19 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
34 @tmp20 = private unnamed_addr constant [49 x i8] c"Attempting to differentiate uninitiali
35 @tmp21 = private unnamed_addr constant [5 x i8] c"PLUS\00"
36 @tmp22 = private unnamed_addr constant [6 x i8] c"MINUS\00"
37 @tmp23 = private unnamed_addr constant [6 x i8] c"TIMES\00"
38 @tmp24 = private unnamed_addr constant [7 x i8] c"DIVIDE\00"
39 @tmp25 = private unnamed_addr constant [4 x i8] c"EXP\00"
40 @tmp26 = private unnamed_addr constant [4 x i8] c"LOG\00"
41 @tmp27 = private unnamed_addr constant [65 x i8] c"Should crash here - invalid symbol ope
42
43 declare i32 @printf(i8*, ...)
44
45 declare i32 @strcmp(i8*, i8*)
46
47 declare i32 @abs(i32)
48
49 declare double @fabs(double)
50
51 declare i8* @operator(i8*)
52
53 declare i8* @createSymbol()
54
55 declare i32 @isConstant(i8*)
56
57 declare i8* @createRoot(i8*, i8*, i8*)
58
59 declare i8* @setSymbolValue(i8*, double)
60
61 declare double @value(i8*)
62
63 declare i32 @isInitialized(i8*)
64
65 declare i8* @left(i8*)
66
67 declare i8* @right(i8*)
68
69 declare double @pow(double, double)
70
71 declare double @log(double)
72
73 declare i32 @printbig(i32)
74
75 define i32 @main() {
76 entry:
77   %symbolmal = call i8* @createSymbol()
78   store i8* %symbolmal, i8** @a
79   %symbolmal1 = call i8* @createSymbol()
```

```
80  store i8* %symbolmal1, i8** @b
81  %symbolmal2 = call i8* @createSymbol()
82  store i8* %symbolmal2, i8** @c
83  %symbolmal3 = call i8* @createSymbol()
84  store i8* %symbolmal3, i8** @d
85  %symbolmal4 = call i8* @createSymbol()
86  store i8* %symbolmal4, i8** @e
87  %symbolmal5 = call i8* @createSymbol()
88  store i8* %symbolmal5, i8** @f
89  %symbolmal6 = call i8* @createSymbol()
90  store i8* %symbolmal6, i8** @g
91  store double 2.718280e+00, double* @exponentialConstant
92  %b = load i8** @b
93  %c = load i8** @c
94  %symbolm = call i8* @createRoot(i8* %b, i8* %c, i8* getelementptr inbounds ([5 x i8]* @
95  store i8* %symbolm, i8** @a
96  %b7 = load i8** @b
97  %symbolm8 = call i8* @setSymbolValue(i8* %b7, double 1.000000e+00)
98  store i8* %symbolm8, i8** @b
99  %c9 = load i8** @c
100 %symbolm10 = call i8* @setSymbolValue(i8* %c9, double 2.000000e+00)
101 store i8* %symbolm10, i8** @c
102 %b11 = load i8** @b
103 %eval_result = call double @eval(i8* %b11)
104 %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatstr,
105 %c12 = load i8** @c
106 %eval_result13 = call double @eval(i8* %c12)
107 %printf14 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatst
108 %a = load i8** @a
109 %eval_result15 = call double @eval(i8* %a)
110 %printf16 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatst
111 %e = load i8** @e
112 %f = load i8** @f
113 %symbolm17 = call i8* @createRoot(i8* %e, i8* %f, i8* getelementptr inbounds ([6 x i8]*
114 store i8* %symbolm17, i8** @d
115 %e18 = load i8** @e
116 %symbolm19 = call i8* @setSymbolValue(i8* %e18, double 3.000000e+00)
117 store i8* %symbolm19, i8** @e
118 %f20 = load i8** @f
119 %symbolm21 = call i8* @setSymbolValue(i8* %f20, double 4.000000e+00)
120 store i8* %symbolm21, i8** @f
121 %e22 = load i8** @e
122 %eval_result23 = call double @eval(i8* %e22)
123 %printf24 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatst
124 %f25 = load i8** @f
125 %eval_result26 = call double @eval(i8* %f25)
126 %printf27 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatst
127 %d = load i8** @d
128 %eval_result28 = call double @eval(i8* %d)
129 %printf29 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatst
130 %a30 = load i8** @a
```

```
131   %d31 = load i8** @d
132   %symbolm32 = call i8* @createRoot(i8* %a30, i8* %d31, i8* getelementptr inbounds ([4 x
133   store i8* %symbolm32, i8** @g
134   %g = load i8** @g
135   %eval_result33 = call double @eval(i8* %g)
136   %printf34 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @floatst
137   ret i32 0
138 }
139
140 define i1 @streq(i8* %a, i8* %b) {
141 entry:
142   %a1 = alloca i8*
143   store i8* %a, i8** %a1
144   %b2 = alloca i8*
145   store i8* %b, i8** %b2
146   %b3 = load i8** %b2
147   %a4 = load i8** %a1
148   %strcmp = call i32 @strcmp(i8* %a4, i8* %b3)
149   %tmp = icmp eq i32 %strcmp, 0
150   ret i1 %tmp
151 }
152
153 define double @eval(i8* %a) {
154 entry:
155   %a1 = alloca i8*
156   store i8* %a, i8** %a1
157   %leftValue = alloca double
158   %rightValue = alloca double
159   %result = alloca double
160   %op = alloca i8*
161   %a2 = load i8** %a1
162   %symbol_const = call i32 @isConstant(i8* %a2)
163   %tmp = icmp eq i32 %symbol_const, 1
164   br i1 %tmp, label %then, label %else
165
166 merge:                                            ; preds = %merge6, %then
167   %result91 = load double* %result
168   ret double %result91
169
170 then:                                             ; preds = %entry
171   %a3 = load i8** %a1
172   %symbol_value = call double @value(i8* %a3)
173   store double %symbol_value, double* %result
174   br label %merge
175
176 else:                                             ; preds = %entry
177   %a4 = load i8** %a1
178   %symbol_init = call i32 @isInitialized(i8* %a4)
179   %tmp5 = icmp ne i32 %symbol_init, 1
180   br i1 %tmp5, label %then7, label %else8
181
```

```
182 merge6:                                           ; preds = %merge11, %then7
183   br label %merge
184
185 then7:                                            ; preds = %else
186   %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmtstr7,
187   br label %merge6
188
189 else8:                                            ; preds = %else
190   %a9 = load i8** %a1
191   %symbol_operator = call i8* @operator(i8* %a9)
192   store i8* %symbol_operator, i8** %op
193   %op10 = load i8** %op
194   %streq_result = call i1 @streq(i8* %op10, i8* getelementptr inbounds ([5 x i8]* @tmp10,
195   br i1 %streq_result, label %then12, label %else19
196
197 merge11:                                          ; preds = %merge22, %then12
198   br label %merge6
199
200 then12:                                           ; preds = %else8
201   %a13 = load i8** %a1
202   %symbol_left_call = call i8* @left(i8* %a13)
203   %eval_result = call double @eval(i8* %symbol_left_call)
204   store double %eval_result, double* %leftValue
205   %a14 = load i8** %a1
206   %symbol_right_call = call i8* @right(i8* %a14)
207   %eval_result15 = call double @eval(i8* %symbol_right_call)
208   store double %eval_result15, double* %rightValue
209   %leftValue16 = load double* %leftValue
210   %rightValue17 = load double* %rightValue
211   %tmp18 = fadd double %leftValue16, %rightValue17
212   store double %tmp18, double* %result
213   br label %merge11
214
215 else19:                                           ; preds = %else8
216   %op20 = load i8** %op
217   %streq_result21 = call i1 @streq(i8* %op20, i8* getelementptr inbounds ([6 x i8]* @tmp1
218   br i1 %streq_result21, label %then23, label %else33
219
220 merge22:                                          ; preds = %merge36, %then23
221   br label %merge11
222
223 then23:                                           ; preds = %else19
224   %a24 = load i8** %a1
225   %symbol_left_call25 = call i8* @left(i8* %a24)
226   %eval_result26 = call double @eval(i8* %symbol_left_call25)
227   store double %eval_result26, double* %leftValue
228   %a27 = load i8** %a1
229   %symbol_right_call28 = call i8* @right(i8* %a27)
230   %eval_result29 = call double @eval(i8* %symbol_right_call28)
231   store double %eval_result29, double* %rightValue
232   %leftValue30 = load double* %leftValue
```

```
233   %rightValue31 = load double* %rightValue
234   %tmp32 = fsub double %leftValue30, %rightValue31
235   store double %tmp32, double* %result
236   br label %merge22
237
238 else33:                                           ; preds = %else19
239   %op34 = load i8** %op
240   %streq_result35 = call i1 @streq(i8* %op34, i8* getelementptr inbounds ([6 x i8]* @tmp1
241   br i1 %streq_result35, label %then37, label %else47
242
243 merge36:                                          ; preds = %merge50, %then37
244   br label %merge22
245
246 then37:                                           ; preds = %else33
247   %a38 = load i8** %a1
248   %symbol_left_call39 = call i8* @left(i8* %a38)
249   %eval_result40 = call double @eval(i8* %symbol_left_call39)
250   store double %eval_result40, double* %leftValue
251   %a41 = load i8** %a1
252   %symbol_right_call42 = call i8* @right(i8* %a41)
253   %eval_result43 = call double @eval(i8* %symbol_right_call42)
254   store double %eval_result43, double* %rightValue
255   %leftValue44 = load double* %leftValue
256   %rightValue45 = load double* %rightValue
257   %tmp46 = fmul double %leftValue44, %rightValue45
258   store double %tmp46, double* %result
259   br label %merge36
260
261 else47:                                           ; preds = %else33
262   %op48 = load i8** %op
263   %streq_result49 = call i1 @streq(i8* %op48, i8* getelementptr inbounds ([7 x i8]* @tmp1
264   br i1 %streq_result49, label %then51, label %else61
265
266 merge50:                                          ; preds = %merge64, %then51
267   br label %merge36
268
269 then51:                                           ; preds = %else47
270   %a52 = load i8** %a1
271   %symbol_left_call53 = call i8* @left(i8* %a52)
272   %eval_result54 = call double @eval(i8* %symbol_left_call53)
273   store double %eval_result54, double* %leftValue
274   %a55 = load i8** %a1
275   %symbol_right_call56 = call i8* @right(i8* %a55)
276   %eval_result57 = call double @eval(i8* %symbol_right_call56)
277   store double %eval_result57, double* %rightValue
278   %leftValue58 = load double* %leftValue
279   %rightValue59 = load double* %rightValue
280   %tmp60 = fdiv double %leftValue58, %rightValue59
281   store double %tmp60, double* %result
282   br label %merge50
283
```

```
284 else61:                                           ; preds = %else47
285   %op62 = load i8** %op
286   %streq_result63 = call i1 @streq(i8* %op62, i8* getelementptr inbounds ([4 x i8]* @tmp1
287   br i1 %streq_result63, label %then65, label %else74
288
289 merge64:                                          ; preds = %merge77, %then65
290   br label %merge50
291
292 then65:                                           ; preds = %else61
293   %a66 = load i8** %a1
294   %symbol_left_call67 = call i8* @left(i8* %a66)
295   %eval_result68 = call double @eval(i8* %symbol_left_call67)
296   store double %eval_result68, double* %leftValue
297   %a69 = load i8** %a1
298   %symbol_right_call70 = call i8* @right(i8* %a69)
299   %eval_result71 = call double @eval(i8* %symbol_right_call70)
300   store double %eval_result71, double* %rightValue
301   %leftValue72 = load double* %leftValue
302   %rightValue73 = load double* %rightValue
303   %pow_func = call double @pow(double %leftValue72, double %rightValue73)
304   store double %pow_func, double* %result
305   br label %merge64
306
307 else74:                                           ; preds = %else61
308   %op75 = load i8** %op
309   %streq_result76 = call i1 @streq(i8* %op75, i8* getelementptr inbounds ([4 x i8]* @tmp1
310   br i1 %streq_result76, label %then78, label %else89
311
312 merge77:                                          ; preds = %else89, %then78
313   br label %merge64
314
315 then78:                                           ; preds = %else74
316   %a79 = load i8** %a1
317   %symbol_left_call80 = call i8* @left(i8* %a79)
318   %eval_result81 = call double @eval(i8* %symbol_left_call80)
319   store double %eval_result81, double* %leftValue
320   %a82 = load i8** %a1
321   %symbol_right_call83 = call i8* @right(i8* %a82)
322   %eval_result84 = call double @eval(i8* %symbol_right_call83)
323   store double %eval_result84, double* %rightValue
324   %leftValue85 = load double* %leftValue
325   %rightValue86 = load double* %rightValue
326   %log_func = call double @log(double %rightValue86)
327   %log_func87 = call double @log(double %leftValue85)
328   %tmp88 = fdiv double %log_func, %log_func87
329   store double %tmp88, double* %result
330   br label %merge77
331
332 else89:                                           ; preds = %else74
333   %printf90 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmtstr7
334   br label %merge77
```

```
335 }
336
337 define double @partialDerivative(i8* %out, i8* %in) {
338 entry:
339   %out1 = alloca i8*
340   store i8* %out, i8** %out1
341   %in2 = alloca i8*
342   store i8* %in, i8** %in2
343   %leftGrad = alloca double
344   %rightGrad = alloca double
345   %op = alloca i8*
346   %dadL = alloca double
347   %dadR = alloca double
348   %L = alloca double
349   %R = alloca double
350   %result = alloca double
351   store double 0.000000e+00, double* %dadL
352   store double 0.000000e+00, double* %dadR
353   %out3 = load i8** %out1
354   %symbol_const = call i32 @isConstant(i8* %out3)
355   %tmp = icmp eq i32 %symbol_const, 1
356   br i1 %tmp, label %then, label %else9
357
358 merge:                                            ; preds = %merge12, %merge7
359   %result150 = load double* %result
360   ret double %result150
361
362 then:                                             ; preds = %entry
363   %out4 = load i8** %out1
364   %in5 = load i8** %in2
365   %e2 = ptrtoint i8* %in5 to i32
366   %e1 = ptrtoint i8* %out4 to i32
367   %tmp6 = icmp eq i32 %e1, %e2
368   br i1 %tmp6, label %then8, label %else
369
370 merge7:                                           ; preds = %else, %then8
371   br label %merge
372
373 then8:                                            ; preds = %then
374   store double 1.000000e+00, double* %result
375   br label %merge7
376
377 else:                                             ; preds = %then
378   store double 0.000000e+00, double* %result
379   br label %merge7
380
381 else9:                                            ; preds = %entry
382   %out10 = load i8** %out1
383   %symbol_init = call i32 @isInitialized(i8* %out10)
384   %tmp11 = icmp ne i32 %symbol_init, 1
385   br i1 %tmp11, label %then13, label %else14
```

```
386
387 merge12:                                          ; preds = %merge16, %then13
388   br label %merge
389
390 then13:                                          ; preds = %else9
391   %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmtstr18,
392   br label %merge12
393
394 else14:                                          ; preds = %else9
395   %out15 = load i8** %out1
396   %symbol_operator = call i8* @operator(i8* %out15)
397   store i8* %symbol_operator, i8** %op
398   br i1 false, label %then17, label %else23
399
400 merge16:                                          ; preds = %merge32, %then17
401   br label %merge12
402
403 then17:                                          ; preds = %else14
404   %in18 = load i8** %in2
405   %out19 = load i8** %out1
406   %symbol_left_call = call i8* @left(i8* %out19)
407   %partialDerivative_result = call double @partialDerivative(i8* %symbol_left_call, i8* %
408   store double %partialDerivative_result, double* %leftGrad
409   store double -1.000000e+00, double* %dadL
410   %dadL20 = load double* %dadL
411   %leftGrad21 = load double* %leftGrad
412   %tmp22 = fmul double %dadL20, %leftGrad21
413   store double %tmp22, double* %result
414   br label %merge16
415
416 else23:                                          ; preds = %else14
417   %in24 = load i8** %in2
418   %out25 = load i8** %out1
419   %symbol_left_call26 = call i8* @left(i8* %out25)
420   %partialDerivative_result27 = call double @partialDerivative(i8* %symbol_left_call26, i
421   store double %partialDerivative_result27, double* %leftGrad
422   %in28 = load i8** %in2
423   %out29 = load i8** %out1
424   %symbol_right_call = call i8* @right(i8* %out29)
425   %partialDerivative_result30 = call double @partialDerivative(i8* %symbol_right_call, i8
426   store double %partialDerivative_result30, double* %rightGrad
427   %op31 = load i8** %op
428   %streq_result = call i1 @streq(i8* %op31, i8* getelementptr inbounds ([5 x i8]* @tmp21,
429   br i1 %streq_result, label %then33, label %else34
430
431 merge32:                                          ; preds = %merge37, %then33
432   %dadL143 = load double* %dadL
433   %leftGrad144 = load double* %leftGrad
434   %tmp145 = fmul double %dadL143, %leftGrad144
435   %dadR146 = load double* %dadR
436   %rightGrad147 = load double* %rightGrad
```

```
437   %tmp148 = fmul double %dadR146, %rightGrad147
438   %tmp149 = fadd double %tmp145, %tmp148
439   store double %tmp149, double* %result
440   br label %merge16
441
442 then33:                                           ; preds = %else23
443   store double 1.000000e+00, double* %dadL
444   store double 1.000000e+00, double* %dadR
445   br label %merge32
446
447 else34:                                           ; preds = %else23
448   %op35 = load i8** %op
449   %streq_result36 = call i1 @streq(i8* %op35, i8* getelementptr inbounds ([6 x i8]* @tmp2
450   br i1 %streq_result36, label %then38, label %else39
451
452 merge37:                                          ; preds = %merge42, %then38
453   br label %merge32
454
455 then38:                                           ; preds = %else34
456   store double 1.000000e+00, double* %dadL
457   store double -1.000000e+00, double* %dadR
458   br label %merge37
459
460 else39:                                           ; preds = %else34
461   %op40 = load i8** %op
462   %streq_result41 = call i1 @streq(i8* %op40, i8* getelementptr inbounds ([6 x i8]* @tmp2
463   br i1 %streq_result41, label %then43, label %else49
464
465 merge42:                                          ; preds = %merge52, %then43
466   br label %merge37
467
468 then43:                                           ; preds = %else39
469   %out44 = load i8** %out1
470   %symbol_right_call45 = call i8* @right(i8* %out44)
471   %eval_result = call double @eval(i8* %symbol_right_call45)
472   store double %eval_result, double* %dadL
473   %out46 = load i8** %out1
474   %symbol_left_call47 = call i8* @left(i8* %out46)
475   %eval_result48 = call double @eval(i8* %symbol_left_call47)
476   store double %eval_result48, double* %dadR
477   br label %merge42
478
479 else49:                                           ; preds = %else39
480   %op50 = load i8** %op
481   %streq_result51 = call i1 @streq(i8* %op50, i8* getelementptr inbounds ([7 x i8]* @tmp2
482   br i1 %streq_result51, label %then53, label %else66
483
484 merge52:                                          ; preds = %merge69, %then53
485   br label %merge42
486
487 then53:                                           ; preds = %else49
```

```
488   %out54 = load i8** %out1
489   %symbol_right_call55 = call i8* @right(i8* %out54)
490   %eval_result56 = call double @eval(i8* %symbol_right_call55)
491   %tmp57 = fdiv double 1.000000e+00, %eval_result56
492   store double %tmp57, double* %dadL
493   %out58 = load i8** %out1
494   %symbol_left_call59 = call i8* @left(i8* %out58)
495   %eval_result60 = call double @eval(i8* %symbol_left_call59)
496   %out61 = load i8** %out1
497   %symbol_right_call62 = call i8* @right(i8* %out61)
498   %eval_result63 = call double @eval(i8* %symbol_right_call62)
499   %pow_func = call double @pow(double %eval_result63, double 2.000000e+00)
500   %tmp64 = fdiv double %eval_result60, %pow_func
501   %tmp65 = fsub double 0.000000e+00, %tmp64
502   store double %tmp65, double* %dadR
503   br label %merge52
504
505 else66:                                           ; preds = %else49
506   %op67 = load i8** %op
507   %streq_result68 = call i1 @streq(i8* %op67, i8* getelementptr inbounds ([4 x i8]* @tmp2
508   br i1 %streq_result68, label %then70, label %else100
509
510 merge69:                                          ; preds = %merge103, %merge90
511   br label %merge52
512
513 then70:                                           ; preds = %else66
514   %out71 = load i8** %out1
515   %symbol_left_call72 = call i8* @left(i8* %out71)
516   %eval_result73 = call double @eval(i8* %symbol_left_call72)
517   store double %eval_result73, double* %L
518   %out74 = load i8** %out1
519   %symbol_right_call75 = call i8* @right(i8* %out74)
520   %eval_result76 = call double @eval(i8* %symbol_right_call75)
521   store double %eval_result76, double* %R
522   %leftGrad77 = load double* %leftGrad
523   %tmp78 = fcmp one double %leftGrad77, 0.000000e+00
524   br i1 %tmp78, label %then80, label %else87
525
526 merge79:                                          ; preds = %else87, %then80
527   %rightGrad88 = load double* %rightGrad
528   %tmp89 = fcmp one double %rightGrad88, 0.000000e+00
529   br i1 %tmp89, label %then91, label %else99
530
531 then80:                                           ; preds = %then70
532   %R81 = load double* %R
533   %L82 = load double* %L
534   %R83 = load double* %R
535   %tmp84 = fsub double %R83, 1.000000e+00
536   %pow_func85 = call double @pow(double %L82, double %tmp84)
537   %tmp86 = fmul double %R81, %pow_func85
538   store double %tmp86, double* %dadL
```

```
539   br label %merge79
540
541 else87:                                           ; preds = %then70
542   br label %merge79
543
544 merge90:                                          ; preds = %else99, %then91
545   br label %merge69
546
547 then91:                                           ; preds = %merge79
548   %exponentialConstant = load double* @exponentialConstant
549   %L92 = load double* %L
550   %log_func = call double @log(double %L92)
551   %log_func93 = call double @log(double %exponentialConstant)
552   %tmp94 = fdiv double %log_func, %log_func93
553   %L95 = load double* %L
554   %R96 = load double* %R
555   %pow_func97 = call double @pow(double %L95, double %R96)
556   %tmp98 = fmul double %tmp94, %pow_func97
557   store double %tmp98, double* %dadR
558   br label %merge90
559
560 else99:                                           ; preds = %merge79
561   br label %merge90
562
563 else100:                                          ; preds = %else66
564   %op101 = load i8** %op
565   %streq_result102 = call i1 @streq(i8* %op101, i8* getelementptr inbounds ([4 x i8]* @tm
566   br i1 %streq_result102, label %then104, label %else141
567
568 merge103:                                         ; preds = %else141, %merge131
569   br label %merge69
570
571 then104:                                          ; preds = %else100
572   %out105 = load i8** %out1
573   %symbol_left_call106 = call i8* @left(i8* %out105)
574   %eval_result107 = call double @eval(i8* %symbol_left_call106)
575   store double %eval_result107, double* %L
576   %out108 = load i8** %out1
577   %symbol_right_call109 = call i8* @right(i8* %out108)
578   %eval_result110 = call double @eval(i8* %symbol_right_call109)
579   store double %eval_result110, double* %R
580   %leftGrad111 = load double* %leftGrad
581   %tmp112 = fcmp one double %leftGrad111, 0.000000e+00
582   br i1 %tmp112, label %then114, label %else128
583
584 merge113:                                         ; preds = %else128, %then114
585   %rightGrad129 = load double* %rightGrad
586   %tmp130 = fcmp one double %rightGrad129, 0.000000e+00
587   br i1 %tmp130, label %then132, label %else140
588
589 then114:                                          ; preds = %then104
```

```
590   %L115 = load double* %L
591   %R116 = load double* %R
592   %log_func117 = call double @log(double %R116)
593   %log_func118 = call double @log(double %L115)
594   %tmp119 = fdiv double %log_func117, %log_func118
595   %L120 = load double* %L
596   %exponentialConstant121 = load double* @exponentialConstant
597   %log_func122 = call double @log(double %exponentialConstant121)
598   %log_func123 = call double @log(double %L120)
599   %tmp124 = fdiv double %log_func122, %log_func123
600   %tmp125 = fmul double %tmp119, %tmp124
601   %L126 = load double* %L
602   %tmp127 = fdiv double %tmp125, %L126
603   store double %tmp127, double* %dadL
604   br label %merge113
605
606 else128:                                          ; preds = %then104
607   br label %merge113
608
609 merge131:                                         ; preds = %else140, %then132
610   br label %merge103
611
612 then132:                                          ; preds = %merge113
613   %L133 = load double* %L
614   %exponentialConstant134 = load double* @exponentialConstant
615   %log_func135 = call double @log(double %exponentialConstant134)
616   %log_func136 = call double @log(double %L133)
617   %tmp137 = fdiv double %log_func135, %log_func136
618   %R138 = load double* %R
619   %tmp139 = fdiv double %tmp137, %R138
620   store double %tmp139, double* %dadR
621   br label %merge131
622
623 else140:                                          ; preds = %merge113
624   br label %merge131
625
626 else141:                                          ; preds = %else100
627   %printf142 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmtstr
628   br label %merge103
629 }
```

6.2 Test Summary

6.2.1 Test Choices

We selected tests that cover basic functionality of the language, such as variable assignments and initialization. We also have extensive tests that cover symbols - returning them as values, using them, initializing them, and running our standard library functions on them.

6.2.2 Automation

We defined a file called **testall.sh**, which is a shell script that runs all our test files. It does this by going through a multi-step process, given an existing **damo.native** file created by our **MAKEFILE**:

1. Prepend standard library into a a **.dml** file
2. Append the actual code into that file
3. Compile the file into an LLVM file (**.ll**)
4. Turn that into assembly
5. Create an executable from that assembly
6. Runs the executable and diffs its output with the expected output

This generates a **testall.log** file that contains a summary of each of the tests and their success or failure. In the case of failure, the log will contain the command that produced the failure as well as the error message.

For failed tests, all the relevant files, which includes the diff, the assembly, the LLVM code, the executable, and the actual output will be saved to disk, where we can view it.

Overall, this let us implement an Edit-Make-Test development process, where after every commit we could immediately test out the language and watch for any breaking changes.

6.2.3 Testing Responsibilities

For each part of the project that we were responsible for, we wrote tests. For example, since Abhiroop implemented array functionality for 1-dimensional arrays, he wrote the tests that covered the general usage of those arrays. This goes for the rest of the team as well. Alan managed the tests for the structural parts of the language, such as some function declarations and variable initializations.

6.3 Test Suite

6.3.1 fail-if1

Source

```
def test() : int {
        if (true) {}
        if (false) {} else {}
        if (42) {}
}
```

Error

```
Fatal error: exception Failure("expected Boolean expression in 42")
```

6.3.2 fail-local

Source

```
def foo(bool i): void{
    int i;
    i = 42;
    print_int(i + i);
}

foo(true);
```

Error

```
Fatal error: exception Failure("duplicate found")
```

6.3.3 test-absint

Source

```
int i = 3;
print_int(absInt(i));
i = - 3;
print_int(absInt(i));


symbol a;
symbol b;
symbol c;
```

Output

```
3
3
```

6.3.4 test-absnum

Source

```
num i = 1.;
if (absNum(i) == 1.){
        print("Right");
}
else{
        print("Wrong");
}

i = 0-1.;
if (absNum(i) == 1.){
```

```
        print("Right");
}
else{
        print("Wrong");
}
```

Output

```
Right
Right
```

6.3.5 test-array

Source

```
def arr() : void{

        int array[5];
        array[0]=1;
        string string_array[5];
        string_array[0]="string";
        print(string_array[0]);
        print_int(array[0]);
        symbol s_array[1];
        print_int(1+array[0]);
}
arr();
```

Output

```
string
1
2
```

6.3.6 test-bool-ops

Source

```
if (true && true){
        print("Right");
}

if (true && false){
        print("Wrong");
}

if (false && true){
        print("Wrong");
}
```

```

if (true || true){
        print("Right");
}

if (true || false){
        print("Right");
}

if (false || true){
        print("Right");
}

if (false || false){
        print("Wrong");
}

if (true){
        print("Right");
}

if (false){
        print("Wrong");
}

if (! true){
        print("Wrong");
}

if (! false){
        print("Right");
}
```

Output

```
Right
Right
Right
Right
Right
Right
```

6.3.7 test-elseif

Source

```
if (true) {
        print_int(0);
}
elseif (true) {
```

```
        print_int(1);
}
else {
        print_int(2);
}

if (false) {
        print_int(3);
}
elseif (true) {
        print_int(4);
}
else {
        print_int(5);
}

if (false) {
        print_int(6);
}
elseif (false) {
        print_int(7);
}
else {
        print_int(8);
}
```

Output

```
0
4
8
```

6.3.8 test-empty

Source

```

```

Output

```

```

6.3.9 test-escape-print

Source

```
def test() : int
{
    print("Hello, literal world! \n\n Jello World");
    return 0;
```

```
}

test();
```

Output

```
Hello, literal world!

 Jello World
```

6.3.10 test-fib

Source

```
def fib(int x): int {
    if (x < 2) {
        return 1;
    } else {
        return fib(x - 1) + fib(x - 2);
    }
}

  print_int(fib(0));
  print_int(fib(1));
  print_int(fib(2));
  print_int(fib(3));
  print_int(fib(4));
  print_int(fib(5));
```

Output

```
1
1
2
3
5
8
```

6.3.11 test-for1

Source

```
int i;
for (i = 0; i < 5; i = i + 1) {
    print_int(i);
}
print_int(42);
```

Output

```
0
1
2
3
4
42
```

Output

```
0
1
2
3
4
42
```

6.3.12 test-func

Source

```
print_int(15);

def test(): int {
    int x;

    x = 15;
    print_int(x);

    return 0;
}

test();
```

Output

```
15
15
```

6.3.13 test-func2

Source

```
def fun(int x, int y): int {
    return 0;
}

int i;
```

```
i = 1;
fun(i, 2);
print_int(i);
```

Output

```
1
```

6.3.14 test-func3

Source

```
def printem(int a, int b, int c, int d): void
{
  print_int(a);
  print_int(b);
  print_int(c);
  print_int(d);
}

printem(42,17,192,8);
```

Output

```
42
17
192
8
```

Output

```
62
```

6.3.15 test-func5

Source

```
def foo(int a): int {
    return a;
}

print_int(foo(5));
```

Output

```
5
```

6.3.16 test-func6

Source

```
def foo(): void {

}

def bar(int a, bool b, int c): int {
    return a + c;
}

print_int(bar(17, false, 25));
```

Output

```
42
```

6.3.17 test-func7

Source

```
int a;

def foo(int c): void {
  a = c + 42;
}

foo(73);
print_int(a);
```

Output

```
115
```

6.3.18 test-func8

Source

```
def foo(int a) : void {
    print_int(a + 3);
}

foo(40);
```

Output

```
43
```

6.3.19 test-gcd

Source

```
def gcd(int a, int b): int {
    while (a != b) {
        if (a > b)  {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}

print_int(gcd(2, 14));
print_int(gcd(3, 15));
print_int(gcd(99, 121));
```

Output

```
2
3
11
```

6.3.20 test-if1

Source

```
// Here's a comment
def test() : int
{
  //This is another one
  if (false) {
        print_int(42);
  }
  else if(true){
        print_int(41);
  }

  print_int(17);
  return 0;
}

test();
```

Output

```
41
17
```

6.3.21 test-if2

Source

```
if (true) {
    print_int(42);
}
print_int(17);
```

Output

```
42
17
```

6.3.22 test-if3

Source

```
if (true) {
    print_int(42);
} else {
    print_int(8);
}
print_int(17);
```

Output

```
42
17
```

6.3.23 test-if4

Source

```
if (false) {
    print_int(42);
} else {
    print_int(8);
}
print_int(17);
```

Output

```
8
17
```

6.3.24 test-if5

Source

```
if (false) {
    print_int(42);
}
print_int(17);
```

Output

```
17
```

6.3.25 test-if6

Source

```
def cond(bool b): int {
    int x;
    if (b) {
        x = 42;
    } else {
        x = 17;
    }
    return x;
}

print_int(cond(true));
print_int(cond(false));
```

Output

```
42
17
```

6.3.26 test-ifblocks

Source

```
if (true) {
        print_int(0);
        print_int(1);
}
else {
        print_int(2);
}
if (false) {
        print_int(0);
        print_int(1);
```

```
}
else {
        print_int(2);
}

if (true) {
        print_int(3);
}

else {
        print_int(4);
        print_int(5);
}

if (false) {
        print_int(3);
}

else {
        print_int(4);
        print_int(5);
}

if (true) {
        print_int(6);
        print_int(7);
}

else {
        print_int(8);
}

if (false) {
        print_int(6);
        print_int(7);
}
else {
        print_int(8);
}
```

Output

```
0
1
2
3
4
5
6
7
8
```

6.3.27 test-init-int

Source

```
        // Variable declarations come first
        int i = 3;
        print_int(i);
```

Output

```
3
```

6.3.28 test-init

Source

```
num x = 5.0;
string y;
y = "hello";
bool a = true;
print_num(x+3.0);
print(y);
print_bool(a);

/*def print_boolean() : int {

        string y;
        y = "hello";
        return 0;

}

print_boolean();*/
```

Output

```
8.000000
hello
1
```

6.3.29 test-int-add

Source

```
def test() : int {
        // Variable declarations come first
        int i;
```

```
        int j;
        int k;

        // Statements come second
        j = 2;
        i = 1;
        k = j + i;
        print_int(k);
        print_int(3);
        return 0;
}

test();
```

Output

```
3
3
```

6.3.30 test-int

Source

```
int a=3;
def test() : int {
        // Variable declarations come first
        int i;
        int j;

        // Statements come second
        j = 2;
        i = 1;
        print_int(i);
        print_int(j);
        print_int(a+j);
        return 0;
}

test();
```

Output

```
1
2
5
```

6.3.31 test-integration-bop

Source

```
def test() : int {
        // Variable declarations come first
        int i;
        int j;
        num k;
        num l;

        k = 1.0;
        l = 2.0;
        j = 2;
        i = 4;

        print_int( i/j );
        print_int( i + j );
        print_num(k*l);
        print_num(l-k);


        return 0;
}

test();
```

Output

```
2
6
2.000000
1.000000
```

6.3.32 test-local

Source

```
def foo(int a, bool b): int {
    int c;
    bool d;
    c = a;
    return c + 10;
}

print_int(foo(37, false));
```

Output

```
47
```

6.3.33 test-log-fun

Source

```
def pow_test() : int {
        // Variable declarations come first
        num i;
        num j;
        int f;
        int k;

        // Statements come second
        j = 4.0;
        i = 2.0;
        f = 4;
        k = 2;


        print_num( i _ j);
        print_int( k _ f );
        return 0;
}

pow_test();
```

Output

```
2.000000
2
```

6.3.34 test-mod-bop

Source

```
def test_mod_bop() : int {
        // Variable declarations come first

        int i;
        int j;
        num k;
        num h;
        k = 4.0;
        h = 2.0;
        i = 4;
        j = 2;

        print_int( i % j );
        print_num( k % h );

        return 0;
}
```

```
18
19
20 test_mod_bop();
```

Output

```
1 0
2 0.000000
```

6.3.35 test-num-add

Source

```
1  def test() : int {
2          // Variable declarations come first
3          num i;
4          num j;
5
6          // Statements come second
7          j = 2.0;
8          i = 1.0;
9          print_num(i+j);
10         return 0;
11 }
12
13 test();
```

Output

```
1 3.000000
```

6.3.36 test-num-int-add

Source

```
1  def test() : int {
2          // Variable declarations come first
3          int i;
4          num j;
5
6          // Statements come second
7          j = 1.7;
8          i = 1;
9          print_num(1.7+1);
10         print_num(j + i);
11         return 0;
12 }
13
14 test();
```

Output

```
2.700000
2.700000
```

6.3.37 test-num

Source

```
def test() : int {
        // Variable declarations come first
        num i;
        num j;

        // Statements come second
        j = 1.7;
        i = .4;
        print_num(i);
        print_num(j);
        return 0;
}

test();
```

Output

```
0.400000
1.700000
```

6.3.38 test-ops1

Source

```
print_int(1 + 2);
print_int(1 - 2);
print_int(1 * 2);
print_int(100 / 2);
print_int(99);
print_bool(1 == 2);
print_bool(1 == 1);
print_int(99);
print_bool(1 != 2);
print_bool(1 != 1);
print_int(99);
print_bool(1 < 2);
print_bool(2 < 1);
print_int(99);
print_bool(1 <= 2);
print_bool(1 <= 1);
print_bool(2 <= 1);
```

```
print_int(99);
print_bool(1 > 2);
print_bool(2 > 1);
print_int(99);
print_bool(1 >= 2);
print_bool(1 >= 1);
print_bool(2 >= 1);
```

Output

```
3
-1
2
50
99
0
1
99
1
0
99
1
0
99
1
1
0
99
0
1
99
0
1
1
```

6.3.39 test-ops2

Source

```
print_bool(true);
print_bool(false);
print_bool(true && true);
print_bool(true && false);
print_bool(false && true);
print_bool(false && false);
print_bool(true || true);
print_bool(true || false);
print_bool(false || true);
print_bool(false || false);
print_bool(!false);
```

```
print_bool(!true);
print_int(-10);
print_int(--42);
```

Output

```
1
0
1
0
0
0
1
1
1
0
1
0
-10
42
```

6.3.40 test-pow-fun

Source

```
def pow_test() : int {
        // Variable declarations come first
        num i;
        num j;
        int f;
        int k;

        // Statements come second
        j = 2.0;
        i = 1.0;
        f = 2;
        k = 2;


        print_num(i^j);
        print_int( f^k );
        return 0;
}

pow_test();
```

Output

```
1.000000
4
```

6.3.41 test-print-bool

Source

```
def print_boolean() : int {
        // Variable declarations come first

        bool i;
        i = true;

        //Statements come second
        3 < 2;
        print_bool( i );
        print_bool( 3 < 2 );
        return 0;
}


print_boolean();
```

Output

```
1
0
```

6.3.42 test-print-variable

Source

```
def test() : int
{
    string test;
    int test_int;

    test = "Hello, world!";
    test_int = 15;

    print(test);
    print_int(test_int);

    return 0;
}

test();
```

Output

```
Hello, world!
15
```

6.3.43 test-print1

Source

```
def test() : int
{
    print("Hello, literal world!");
    return 0;
}

test();
```

Output

```
Hello, literal world!
```

6.3.44 test-print2

Source

```
def test() : int
{
    print("Hello, literal world!");
    print("And here's another one");
    return 0;
}

test();
```

Output

```
Hello, literal world!
And here's another one
```

6.3.45 test-recursion

Source

```
def test(int x): int {
   if(x==1){
        return 1;
    }

    else{
```

```
        return x * test(x-1);
    }
}

print_int(test(5));
```

Output

```
120
```

6.3.46 test-simple-int

Source

```
print("hi");
def test () : void {
  int a;
}
test();
```

Output

```
hi
```

6.3.47 test-simple

Source

```
print_int(1);
def test_print(string a): void {
        int b = 1;
        print(a);
}
test_print("hi");
```

Output

```
1
hi
```

6.3.48 test-strcompare

Source

```
string myString1 = "Hello";
string myString2 = "world";
string myString3;
myString3 = "world";
```

```

if (strcompare(myString1, myString2) != 0){
        print("Right");
}
else{
        print("Wrong");
}

if (strcompare(myString2, myString3) == 0){
        print("Right");
}
else{
        print("Wrong");
}
```

Output

```
Right
Right
```

6.3.49 test-symbol-all-ops

Source

```
symbol a;
symbol b;
symbol c;
a = b + c;
a = b - c;
a = b * c;
a = b / c;
a = b ^ c;
a = b _ c;
```

Output

```

```

6.3.50 test-symbol-assign-int

Source

```
symbol a;
a = 1;
a = 2;
```

Output

```

```

6.3.51 test-symbol-assign-num

Source

```
symbol b;
b = 1.0;
b = 2.0;
```

Output

```

```

6.3.52 test-symbol-assign-var

Source

```
symbol a;
int i = 1;
num j = 2.0;
a = i;
a = j;
```

Output

```

```

6.3.53 test-symbol-const

Source

```
symbol a;
symbol b;
a = 3.0;
a = 2;
b = 1;
b = 5.;
```

Output

```

```

6.3.54 test-symbol-decl

Source

```
symbol a;
symbol b;
```

```
3 symbol c;
```

Output

```
1
```

6.3.55 test-symbol-eval-const

Source

```
1 symbol a;
2 a = 1;
3 if (eval(a) == 1.0){
4         print("Right");
5 }
6 else {
7         print("Wrong");
8 }
```

Output

```
1 Right
```

6.3.56 test-symbol-eval1

Source

```
1  symbol a;
2  symbol b;
3  a = b + 1;
4  b = 1;
5  if (eval(a) == 2.0){
6          print("Right");
7  }
8  else{
9          print("Wrong");
10 }
11
12 b = 2;
13 if (eval(a) == 3.0){
14         print("Right");
15 }
16 else{
17         print("Wrong");
18 }
```

Output

```
Right
Right
```

6.3.57 test-symbol-eval2

Source

```
symbol a;
symbol b;
symbol c;
symbol d;
symbol e;
symbol f;
symbol g;

b = c * d;
e = f / g;

c = 2;
d = 3;

f = 8;
g = 4;

a = e ^ b;

if (eval(a) == 64.0){
        print("Right");
}
else{
        print("Wrong");
}

f = 7.5;
g = 2.5;

if (eval(a) == 729.0){
        print("Right");
}
else{
        print("Wrong");
}
```

Output

```
Right
Right
```

6.3.58 test-symbol-exp

Source

```
symbol a;
symbol b;
symbol c;

a = 5;
b = 2;
c = a ^ b;

if (eval(c) == 25.){
        print("Right");
}
else{
        print("Wrong");
}

c = 4 ^ b;

if (eval(c) == 16.){
        print("Right");
}
else{
        print("Wrong");
}

c = b ^ 3;

if (eval(c) == 8.){
        print("Right");
}
else{
        print("Wrong");
}
```

Output

```
Right
Right
Right
```

6.3.59 test-symbol-expr1

Source

```
symbol a;
symbol b;
symbol c;
a = b + c;
```

Output

```
1
```

6.3.60 test-symbol-expr2

Source

```
symbol a;
symbol b;
a = 2 * b;
a = 3.0 * b;
```

Output

```
1
```

6.3.61 test-symbol-fancy

Source

```
symbol a;
symbol b;
symbol c;
symbol d;

b = 2 + d;
c = 3 + d;
d = 1;
a = b + c;

if (eval(a) == 7.){
        print("Right");
}
else{
        print("Wrong");
}
```

Output

```
Right
```

6.3.62 test-symbol-gradient

Source

```
symbol a;
symbol b;
num result;
num deriv;

b = -2;
a = b ^ 3;

result = eval(a);
deriv = partialDerivative(a, b);

if (result == 0.0-8.0  && deriv == 12.0){
        print("Right");
}
else{
        print("Wrong");
}

b = 3;

result = eval(a);
deriv = partialDerivative(a, b);

if (result == 27.0 && deriv == 27.0){
        print("Right");
}
else{
        print("Wrong");
}

b = 5;
a = exponentialConstant ^ b;

result = eval(a);
deriv = partialDerivative(a, b);

if (result == exponentialConstant ^ 5  && deriv == exponentialConstant ^ 5){
        print("Right");
}
else{
        print("Wrong");
}

b = 3;
a = b * 2;

result = eval(a);
deriv = partialDerivative(a, b);
```

```
49
50 if (result == 6.0 && deriv == 2.0){
51         print("Right");
52 }
53 else{
54         print("Wrong");
55 }
56
57 a = b + 1;
58
59 result = eval(a);
60 deriv = partialDerivative(a, b);
61
62 if (result == 4.0 && deriv == 1.0){
63         print("Right");
64 }
65 else{
66         print("Wrong");
67 }
68
69 a = b * b;
70
71 result = eval(a);
72 deriv = partialDerivative(a, b);
73
74 if (result == 9.0 && deriv == 6.0){
75         print("Right");
76 }
77 else{
78         print("Wrong");
79 }
80
81 b = 4;
82 a = 2 _ b;
83
84 result = eval(a);
85 deriv = partialDerivative(a, b);
86
87 if (result == 2.0 && deriv == 2 _ exponentialConstant / 4){
88         print("Right");
89 }
90 else{
91         print("Wrong");
92 }
93
94 b = 2;
95 a = b _ 4;
96
97 result = eval(a);
98 deriv = partialDerivative(a, b);
99
```

```
100 if (result == 2.0 && deriv == 2 _ exponentialConstant){
101         print("Right");
102 }
103 else{
104         print("Wrong");
105 }
```

Output

```
1 Right
2 Right
3 Right
4 Right
5 Right
6 Right
7 Right
8 Right
```

6.3.63 test-symbol-in-and-return

Source

```
1  symbol a;
2
3  def test(symbol tmp): symbol {
4      symbol x;
5      symbol z;
6      x = 15;
7      z = x + 2.0;
8      tmp = z + 2;
9
10     return tmp;
11 }
12
13 a = test(a);
14
15 print_num( value(right(a)));
```

Output

```
1 2.000000
```

6.3.64 test-symbol-in-function

Source

```
1  print_int(15);
2
3  def test(): int {
```

```
    symbol x;

    x = 15;
    print_num(value(x));

    return 0;
}

test();
```

Output

```
15
15.000000
```

6.3.65 test-symbol-init

Source

```
symbol a;
symbol b;
symbol c;
symbol d;
d = 1;

a = 1 + b;
print_int( isInitialized(c) );
print_int( isInitialized(a) );
```

Output

```
0
1
```

6.3.66 test-symbol-isConst

Source

```
symbol a;
symbol b;
symbol c;
symbol d;
d = 1;

a = 1 + b;
print_int( isConstant(d) );
print_int( isConstant(a) );
```

Output

```
1
0
```

6.3.67 test-symbol-left-right-val

Source

```
symbol a;
symbol b;
symbol c;

b = 1;
c = 3.0;

a = b + c;

print_num( value(left(a)) );
print_num( value(right(a)));
```

Output

```
1.000000
3.000000
```

6.3.68 test-symbol-return-function

Source

```
symbol a;

def test(): symbol {
    symbol x;
    symbol z;
    x = 15;
    z = x + 2.0;

    return z;
}

a = test( );

print_num( value(right(a)));
```

Output

```
2.000000
```

6.3.69 test-toplvl-print

Source

```
int x;
x = 15;
print_int(x);
```

Output

```
15
```

6.3.70 test-toplvl

Source

```
def test(): int {
    num i;
    i = 0.4;
    print_num(i);
    return 0;
}

test();
```

Output

```
0.400000
```

6.3.71 test-while1

Source

```
def foo(int a): int {
  int j;
  j = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}

print_int(foo(7));
```

Output

```
14
```

6.3.72 test-while2

Source

```
int i;
i = 5;
while (i > 0) {
    print_int(i);
    i = i - 1;
}
print_int(42);
```

Output

```
5
4
3
2
1
42
```

Chapter 7

Lessons Learned

7.1 Important Learnings

Ian Covert

Ian learned that project management is difficult - a team of four is difficult enough, let alone scaling software projects to hundreds or thousands of engineers.

Hari Devaraj

Hari has come to appreciate the complexities and small joys of debugging LLVM code and understanding the code generation process that turns all the programs we write into some form of assembly that eventually runs on our computers' CPUs. Also scoping is a non trivial problem and should be assumed to work. Its pretty cool to see how seamlessly LLVM calls c functions. He also experienced the thrill of coding a program that he could never test. Integration took more than a few hours

Abhiroop Gangopadhyay

Abhiroop learned both the power and the bite of type inference. Many an hour was spent debugging obscure type errors, though he has gained an increased appreciation for the power of algebraic data types and pattern matching. He also experienced the otherworldly bliss of integrating a program that he had not written and had never tested. He still has flash backs to this day.

Alan Gou

Alan has learned the importance of thoroughly testing code, and that even if you might think you have tested enough, there will always be more to test.

7.2 Future Advice

Here are some short tips we would like to give to future teams.

Iterative Development

Iterative development makes life easier. This means always committing, merging, and running your test suite on basically every commit. This helps a lot when you are merging branches, since there will be tons of conflicts introduced that may break large parts of your code.

Specialization is a Double-edged Sword

When one person specializes in some area, such as building the SAST, it means they can be more effective at making changes in that area, but it also carries the risk of bottle-necking progress behind that one person's workload and speed at making changes.

Realistic Deadlines are Necessary

We had a million things we wanted to implement, but in the end, we had to be realistic about how much we could reasonably accomplish. Without setting harsh and realistic deadlines, leaving ample room for unforeseen obstacles, it is incredibly easy to find yourself drawn into a rabbit-hole that does not actually end up making a large impact on the central functionality of the project.

Never Assume that Something Works

This is almost obvious. But it bears repeating. Test. Test. And test. Assuming something will work and that you can test later can bite you hard later on.

Appendix A

Appendix

A.1 ast.ml

```
(* Abstract Syntax Tree and functions for printing it *)

(* NEW mathematical operators *)
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
          And | Or | Exp | Log | Mod

type uop = Neg | Not

(* NEW types *)
type typ = Int | Bool | Num | String | Symbol | Void

type lvalue =
    Idl of string
  | ArrIdl of string * expr list

and expr =
    IntLit of int
  | BoolLit of bool
  | StringLit of string
  | NumLit of float
  | Id of string
  | ArrId of string * expr list
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of lvalue * expr
  | Call of string * expr list
  | Noexpr

type bind =
    Decl of typ * string
  | ArrDecl of typ * string * expr list

type stmt =
    Expr of expr
  | Block of stmt list
```

```
36    | Return of expr
37    | If of expr * stmt * stmt
38    | For of expr * expr * expr * stmt
39    | While of expr * stmt
40
41  (*type function_unit =
42      VarFunit of bind
43    | StmtFunit of stmt
44  *)
45
46  type func_decl = {
47      typ : typ;
48      fname : string;
49      formals : bind list;
50      body : program_sequence list;
51    }
52
53  and program_sequence =
54      VarUnit of bind
55    | FuncUnit of func_decl
56    | StmtUnit of stmt
57
58  type program = program_sequence list
59
60  (* Pretty-printing functions *)
61
62  (* NEW printing mathematical operators *)
63  let string_of_op = function
64      Add -> "+"
65    | Sub -> "-"
66    | Mult -> "*"
67    | Div -> "/"
68    | Exp -> "^"
69    | Log -> "_"
70    | Mod -> "%"
71    | Equal -> "=="
72    | Neq -> "!="
73    | Less -> "<"
74    | Leq -> "<="
75    | Greater -> ">"
76    | Geq -> ">="
77    | And -> "&&"
78    | Or -> "||"
79
80  let string_of_uop = function
81      Neg -> "-"
82    | Not -> "!"
83
84  (* NEW printing strings with quotes *)
85  (*let rec string_of_expr = function
86      IntLit(l) -> string_of_int l
```

```
87    | BoolLit(true) -> "true"
88    | BoolLit(false) -> "false"
89    | NumLit(n) -> string_of_float n
90    | StringLit(s) -> "\"" ^ s ^ "\""
91    | Id(s) -> s
92    | Binop(e1, o, e2) ->
93        string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
94    | Unop(o, e) -> string_of_uop o ^ string_of_expr e
95    | Call(f, el) ->
96        f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
97    | Noexpr -> ""
98  *)
99  (* NEW print string, print num *)
100 let string_of_typ = function
101     Int -> "int"
102   | Bool -> "bool"
103   | Void -> "void"
104   | String -> "string"
105   | Num -> "num"
106   | Symbol -> "symbol"
107
108 (*let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"
109
110 let rec string_of_stmt = function
111     Block(stmts) ->
112       "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
113   | Expr(expr) -> string_of_expr expr ^ ";\n";
114   | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
115   | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
116   | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
117       string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
118   | For(e1, e2, e3, s) ->
119       "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
120       string_of_expr e3  ^ ") " ^ string_of_stmt s
121   | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s*)
122 (*  | Bind(t, i) -> string_of_vdecl (t, i)
123
124 let string_of_fdecl fdecl =
125   string_of_typ fdecl.typ ^ " " ^
126   fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
127   ")\n{\n" ^
128   String.concat "" (List.map string_of_vdecl fdecl.locals) ^
129   String.concat "" (List.map string_of_stmt fdecl.body) ^
130   "}\n"
131
132 let string_of_topstmts topstmts =
133   "int main() { \n" ^
134   String.concat "\n" (List.map string_of_stmt topstmts) ^ "}"
135
136 let string_of_program (topstmts, funcs) =
137   String.concat "" (List.map string_of_fdecl funcs) ^ "\n" ^
```

```
138   string_of_topstmts topstmts ^ "\n"
139 *)
```

A.2 sast.ml

```
1  module AST = Ast
2
3  type t =
4      Int
5    | Bool
6    | Num
7    | String
8    | Symbol
9    | Void
10
11 type lvalue =
12     Idl of t * string
13   | ArrIdl of t * string * s_expr list
14
15 and s_expr =
16     IntLit of t * int
17   | BoolLit of t * bool
18   | StringLit of t * string
19   | NumLit of t * float
20   | Id of t * string
21   | ArrId of t * string * s_expr list
22   | Binop of t * s_expr * AST.op * s_expr
23   | Unop of t * AST.uop * s_expr
24   | Assign of lvalue * s_expr
25   | Call of t * string * s_expr list
26   | Noexpr of t
27
28
29 type s_bind =
30     Decl of t * string
31   | ArrDecl of t * string * s_expr list
32
33
34 type s_stmt =
35     Expr of s_expr
36   | Block of s_stmt list
37   | Return of s_expr
38   | If of s_expr * s_stmt * s_stmt
39   | For of s_expr * s_expr * s_expr * s_stmt
40   | While of s_expr * s_stmt
41
42
43 (*type s_function_unit =
44     VarFunit of s_bind
45   | StmtFunit of s_stmt
```

```
46  *)
47  type s_func_decl = {
48      s_typ : t;
49      s_fname : string;
50      s_formals : s_bind list;
51      s_body : s_program_sequence list;
52    }
53
54  and s_program_sequence =
55      VarUnit of s_bind
56    | FuncUnit of s_func_decl
57    | StmtUnit of s_stmt
58
59  type s_program = s_program_sequence list
```

A.3 codegen.ml

```
1   module L = Llvm
2   module AST = Ast
3   module A = Sast
4
5   module StringMap = Map.Make(String)
6   exception NotImplemented
7   exception IllegalType
8
9
10  let translate (program_unit_list) =
11    let context = L.global_context () in
12    let the_module = L.create_module context "damo"
13    and i32_t  = L.i32_type  context
14    and num_t = L.double_type context
15    and i8_t   = L.i8_type context
16    and str_t = L.pointer_type (L.i8_type context)
17    and i1_t   = L.i1_type   context
18    and void_t = L.void_type context
19    (*and void_ptr = L.pointer_type (L.i8_type context)*)
20    and symbol_t = L.pointer_type (L.i8_type context) in
21
22    let ltype_of_typ = function
23        A.Int -> i32_t
24      | A.Bool -> i1_t
25      | A.Num -> num_t
26      | A.String -> str_t
27      | A.Symbol -> symbol_t
28      | A.Void -> void_t in
29
30
31    (* Declare each global variable; remember its value in a map *)
32
33    (*
```

```
34      The SAST comes in as a list of variable declarations, statements and
35      function declarations. The following function is there to parse this
36      list into its three component lists:
37      a list for main fxn statements,
38      a list for function declarations,
39      and a list for declarations
40    *)
41    let rec combine_prog_units prog_stmts main_stmts gvars func_unit = match prog_stmts wit
42          [] -> (main_stmts, gvars, func_unit)
43        | A.VarUnit(b)::tail -> (combine_prog_units tail main_stmts (gvars@[b]) func_unit)
44        | A.FuncUnit(f)::tail -> (combine_prog_units tail main_stmts gvars (func_unit@[f]))
45        | A.StmtUnit(s)::tail -> (combine_prog_units tail (main_stmts@[s]) gvars func_unit)
46
47        in
48
49    let topstmts, gvars, functions = combine_prog_units program_unit_list [] [] [] in
50
51    let main_function = {
52      A.s_typ = A.Int;
53      A.s_fname = "main";
54      A.s_formals = [];
55      A.s_body = List.map (fun x -> A.StmtUnit(x)) topstmts;
56    } in
57
58    let g_alloc = Hashtbl.create 20 in
59
60
61    (* Now build up global variables by parsing the list of globals
62        First parse the decls in s_bind:
63        Here is where I'm making an assumption, I assume that InitDecl's
64        expr will be a Literal (not ID) *)
65
66    let global_vars =
67      let global_var m decl =
68        (match decl with
69            A.Decl( t, s ) ->
70                (match t with
71                     A.Int -> let init = L.const_int (ltype_of_typ t) 0 in
72                       StringMap.add s ((L.define_global s init the_module),t, [A.Noexpr(t)]
73                   | A.Num -> let init = L.const_float (ltype_of_typ t) 0.0 in
74                       StringMap.add s ((L.define_global s init the_module),t, [A.Noexpr(t)]
75                   | A.String -> let init = L.const_pointer_null (ltype_of_typ t) in
76                       StringMap.add s ((L.define_global s init the_module),t, [A.Noexpr(t)]
77                   | A.Bool -> let init = L.const_int (ltype_of_typ t) 0 in
78                       StringMap.add s ((L.define_global s init the_module),t, [A.Noexpr(t)]
79                   | A.Symbol -> let init = L.const_pointer_null symbol_t in
80                       StringMap.add s ((L.define_global s init the_module), t, [A.Noexpr(t)
81                   | _ -> let init = L.const_int (ltype_of_typ t) 0 in
82                       StringMap.add s ((L.define_global s init the_module),t, [A.Noexpr(t)]
83          | A.ArrDecl (t, s, el) -> let init = L.const_pointer_null (L.i8_type context) in
84                       StringMap.add s ((L.define_global s init the_module),t, el) m
```

```
85        )
86    in List.fold_left global_var StringMap.empty gvars in
87  (*
88    let _ = Printf.printf "%s" (String.concat "\n" (List.map A.string_of_stmt main_function
89    *)
90
91    let functions = main_function :: functions in
92
93    (* Declare C functions that can be called *)
94    let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
95    let printf_func = L.declare_function "printf" printf_t the_module in
96    let string_compare = L.declare_function "strcmp"
97              ( Llvm.function_type i32_t [| str_t; str_t |]) the_module in
98    let abs_int = L.declare_function "abs"
99              ( Llvm.function_type i32_t [| i32_t |]) the_module in
100   let abs_num = L.declare_function "fabs"
101             ( Llvm.function_type num_t [| num_t |]) the_module in
102   let symbol_operator = Llvm.declare_function "operator"
103             ( Llvm.function_type str_t [| symbol_t |] ) the_module in
104   let symbol_malloc = Llvm.declare_function "createSymbol"
105             ( Llvm.function_type symbol_t [|  |] ) the_module in
106   let symbol_const_check = Llvm.declare_function "isConstant"
107             ( Llvm.function_type i32_t [| symbol_t |] ) the_module in
108   let root_symbol = Llvm.declare_function "createRoot"
109             ( Llvm.function_type symbol_t [| symbol_t; symbol_t; str_t; |])  the_module i
110   let const_symbol = Llvm.declare_function "setSymbolValue"
111             ( Llvm.function_type symbol_t [| symbol_t; num_t |] ) the_module in
112   let value_symbol = Llvm.declare_function "value"
113             ( Llvm.function_type num_t [| symbol_t |] ) the_module in
114   let symbol_init_check = Llvm.declare_function "isInitialized"
115             ( Llvm.function_type i32_t [| symbol_t |] ) the_module in
116   let symbol_left = Llvm.declare_function "left"
117             ( Llvm.function_type symbol_t [| symbol_t |] ) the_module in
118   let symbol_right = Llvm.declare_function "right"
119             ( Llvm.function_type symbol_t [| symbol_t |] ) the_module in
120   (*let malloc = LLvm.declare_function "malloc"
121             ( Llvm.function_type void_ptr [| i_32 |] ) the_module in
122   *)
123   let pwr_fxn_num = Llvm.declare_function "pow" ( Llvm.function_type num_t [| num_t; num_
124   let log_fxn_num = Llvm.declare_function "log" ( Llvm.function_type num_t [|  num_t |] )
125   let printbig_t = L.function_type i32_t [| i32_t |] in
126   let printbig_func = L.declare_function "printbig" printbig_t the_module in
127
128   (* Define each function (arguments and return type) so we can call it *)
129   let function_decls =
130     let function_decl m fdecl =
131       let check_type = function
132         A.Decl(t,_) -> ltype_of_typ t
133       | A.ArrDecl(_,_,_) -> i8_t
134       in
135       let name = fdecl.A.s_fname
```

```
136       and formal_types =
137         Array.of_list (List.map (fun x -> check_type x) fdecl.A.s_formals)
138       in let ftype = L.function_type (ltype_of_typ fdecl.A.s_typ) formal_types in
139       StringMap.add name (L.define_function name ftype the_module, fdecl) m in
140     List.fold_left function_decl StringMap.empty functions in
141
142   (* Fill in the body of the given function *)
143   let build_function_body fdecl =
144     let (the_function, _) = StringMap.find fdecl.A.s_fname function_decls in
145     let builder = L.builder_at_end context (L.entry_block the_function) in
146
147     (* NEW formatting string for using printf on strings *)
148     let int_format_str = L.build_global_stringptr "%d\n" "fmtint" builder in
149     let str_format_str = L.build_global_stringptr "%s\n" "fmtstr" builder in
150     let float_format_str = L.build_global_stringptr "%f\n" "floatstr" builder in
151
152     (*
153       PREPROCESSING STEP: PARSE FUNCTION BODY INTO VARFUNIT AND STMTFUNIT,
154         - GET LIST OF VARFUNIT TO BE PROCESSED BY LOCAL VARS TO GET LOCALS
155     *)
156     let rec fxn_body_decouple f_body decl_l stmt_l  = (match f_body with
157         [] -> (decl_l, stmt_l)
158       | A.VarUnit(s) :: tail -> fxn_body_decouple tail (decl_l@[s]) stmt_l
159       | A.StmtUnit(sf) :: tail -> fxn_body_decouple tail decl_l (stmt_l@[sf])
160       | _ :: _ -> raise(Failure("Nothing else should be in a function body"))
161     )
162     in
163
164     let locals, stmt_list = fxn_body_decouple fdecl.A.s_body [] [] in
165     let lookup_global n =
166           try StringMap.find n global_vars
167           with Not_found -> raise(Failure("In add main, variable not defined")) in
168     (* Construct the function's "locals": formal arguments and locally
169        declared variables.  Allocate each on the stack, initialize their
170        value, if appropriate, and remember their values in the "locals" map *)
171     let local_vars =
172       let add_formal m x p = (match x with
173           A.Decl(t, n) -> (match t with
174             A.Symbol -> L.set_value_name n p;
175                         let local = L.build_alloca (ltype_of_typ t) n builder in
176                         ignore (L.build_store p local builder);
177                         StringMap.add n (local, t, [Sast.Noexpr(t)]) m
178           | _ -> L.set_value_name n p; let local = L.build_alloca (ltype_of_typ t) n bu
179                         ignore (L.build_store p local builder);
180                         StringMap.add n (local, t, [Sast.Noexpr(t)]) m )
181         | A.ArrDecl(t,n,el) -> L.set_value_name n p;
182         let local = L.build_alloca (ltype_of_typ t) n builder in
183         ignore (L.build_store p local builder);
184         StringMap.add n (local, t, el) m )
185     in
186       let add_local m x= (match x with
```

```
187             A.Decl(t, n) -> (match t with
188                 A.Symbol -> let variable = L.build_call symbol_malloc [| |] "symbolmal" bui
189                   let v_ptr = symbol_t in let s_ptr= L.build_alloca v_ptr n builder
190                   in ignore(L.build_store variable s_ptr builder);
191                   StringMap.add n (s_ptr, t, [Sast.Noexpr(t)]) m
192
193                 | _ -> let local_var = L.build_alloca (ltype_of_typ t) n builder
194                 in StringMap.add n (local_var, t, [Sast.Noexpr(t)]) m
195               )
196             | A.ArrDecl(t, n, el) -> let local_var = L.build_alloca (ltype_of_typ t) n buil
197               in StringMap.add n (local_var, t, el) m
198         ) in
199         (* Most global variables were declared at the top, but side i didn't have a builder
200           the global context, this fuction mallocs all the global symbols and adds back to
201           global map. all other variables are left unchanged *)
202         let add_main_local m x = (match x with
203             A.Decl(t, n) -> (match t with
204               A.Symbol -> let global_variable = L.build_call symbol_malloc [| |] "symbolmal
205                   let (s_v, _, _) = lookup_global n in ignore(L.build_store global_variable
206                   m
207               | _ -> m
208             )
209             | A.ArrDecl(t, n, _) -> let new_el' = L.const_int (ltype_of_typ t) 100 in let a
210         in
211
212         let formals = List.fold_left2 add_formal StringMap.empty fdecl.A.s_formals
213             (Array.to_list (L.params the_function)) in
214         if fdecl.A.s_fname = "main" then
215             List.fold_left add_main_local global_vars gvars
216         else
217             List.fold_left add_local formals locals in
218           (* Return the value for a variable or formal argument *)
219       let lookup n =
220         if fdecl.A.s_fname = "main" then
221             try StringMap.find n global_vars
222             with Not_found -> raise(Failure("Global variable not defined"))
223         else
224           try StringMap.find n local_vars
225           with Not_found -> try StringMap.find n global_vars
226           with Not_found -> raise (Failure ("not defined yet"))
227       in
228         let get_type node = (match node with
229                 A.IntLit (t , _) -> t
230               | A.BoolLit (t , _) -> t
231               | A.StringLit (t ,_) -> t
232               | A.NumLit (t,_ ) -> t
233               | A.Id (t, _) -> t
234               | A.ArrId (t, _, _) -> t
235               | A.Binop (t, _, _, _) -> t
236               | A.Unop (t, _, _) -> t
237               | A.Assign (lvalue, _) -> (match lvalue with
```

```
238                A.Idl(t, _) -> t
239              | A.ArrIdl(t, _, _) -> t)
240            | A.Call (t, _ , _) -> t
241            | A.Noexpr (t) -> t
242        ) in
243       let get_str_op op_t = (match op_t with
244              AST.Add     -> "PLUS"
245            | AST.Sub     -> "MINUS"
246            | AST.Mult    -> "TIMES"
247            | AST.Div     -> "DIVIDE"
248            | AST.Exp     -> "EXP"
249            | AST.Log     -> "LOG"
250            | _ -> raise(Failure("Not supported operator"))
251        ) in
252
253     (* Construct code for an expression; return its value *)
254     let rec expr builder = function
255         A.IntLit(_, i) -> L.const_int i32_t i
256       | A.BoolLit(_, b) -> L.const_int i1_t (if b then 1 else 0)
257       | A.StringLit(_, st) -> L.build_global_stringptr st "tmp" builder
258       | A.NumLit(_, num) -> L.const_float num_t num
259       | A.Noexpr(_) -> L.const_int i32_t 0
260       | A.Id(_, s) -> let (s_v,_, _) = lookup s in L.build_load s_v s builder
261       | A.ArrId(_, s, el) -> let pointer = Hashtbl.find g_alloc s in let el' = expr build
262       | A.Binop (t, e1, op, e2) ->
263         let e1' = expr builder e1
264         and e2' = expr builder e2 in
265         let t_1 = get_type( e1 )
266         and t_2 = get_type( e2 ) in
267         let binop_type_check ( t, e1, e2 ) =
268            (match t with
269               A.Int -> (e1', e2')
270             | A.Bool -> (match t_1, t_2 with
271                   A.Num, A.Num -> (e1', e2')
272                 | A.Num, A.Int -> (e1',
273                     L.build_sitofp e2' num_t "cast" builder)
274                 | A.Int, A.Num -> (L.build_sitofp e1' num_t "cast" builder,
275                     e2')
276                 | A.Bool, A.Bool -> (e1', e2')
277                 | A.Int, A.Int -> (e1', e2')
278                 | A.Symbol, A.Symbol -> (e1', e2')
279                 | _ -> raise(Failure("not matched")))
280             | A.Num -> (match t_1, t_2 with
281                   A.Num, A.Num -> (e1', e2')
282                 | A.Num, A.Int -> (e1', L.build_sitofp e2' num_t "cast" builder)
283                 | A.Int, A.Num -> (L.build_sitofp e1' num_t "cast" builder, e2')
284                 | A.Bool, A.Bool -> (e1', e2')
285                 | _ -> raise(Failure("not matched num")))
286             | A.Symbol -> let t1 = get_type e1 and t2 = (get_type e2) in ( match t1, t2 w
287                   A.Symbol, A.Symbol -> e1', e2'
288                 | A.Num, A.Symbol -> let num_node = (L.build_call symbol_malloc [| |] "sy
```

```
289                     let e' = L.build_call const_symbol [| num_node; e1' |] "symbolm" bu
290                 | A.Symbol, A.Num -> let num_node = (L.build_call symbol_malloc [| |] "sy
291                     let e' = L.build_call const_symbol [| num_node; e2' |] "symbolm" bu
292                 | A.Int, A.Symbol -> let num_node = (L.build_call symbol_malloc [| |] "sy
293                     let e' = L.build_call const_symbol [|num_node; (L.build_sitofp e1' 
294                     "symbolm" builder in (e', e2')
295                 | A.Symbol, A.Int -> let num_node = (L.build_call symbol_malloc [| |] "sy
296                     let e' = L.build_call const_symbol [|num_node; (L.build_sitofp e2' 
297                     "symbolm" builder in (e1', e')
298                 | _, _ -> raise(Failure("Symbol used improperly"))
299                 )
300             | _ -> raise(Failure("Binop evaluates to an unexpected type"))
301           )
302            in
303         let e1_new', e2_new' = binop_type_check(t, e1, e2) in
304
305         if t = A.Symbol then let sym_type =
306             let operat = L.build_global_stringptr (get_str_op(op)) "tmp" builder in
307             let e' = L.build_call root_symbol [| e1_new'; e2_new'; operat |] "symbolm" bu
308             in sym_type
309         (* Exp invokes C function *)
310         else if op = AST.Exp then
311             let exp_types = (match t with
312               A.Num -> Llvm.build_call pwr_fxn_num [| e1_new'; e2_new' |] "pow_func" buil
313             | A.Int ->
314                let e1_cast = L.build_sitofp e1_new' num_t "cast" builder in
315                let e2_cast = L.build_sitofp e2_new' num_t "cast" builder in
316                let pow_cast =  Llvm.build_call pwr_fxn_num [| e1_cast; e2_cast |] "pow_fu
317                L.build_fptosi pow_cast i32_t "cast" builder
318             | _ -> raise( NotImplemented )
319             ) in exp_types
320         (* Log invokes C function*)
321         else if op = AST.Log then
322          let log_types = (match t with
323               A.Num -> let top_log = Llvm.build_call log_fxn_num [| e2_new' |] "log_func"
324                        let bottom_log =  Llvm.build_call log_fxn_num [| e1_new' |] "log_f
325                        Llvm.build_fdiv top_log bottom_log "tmp" builder
326             | A.Int ->
327                        let e1_cast = L.build_sitofp e1_new' num_t "cast" builder in
328                        let e2_cast = L.build_sitofp e2_new' num_t "cast" builder in
329                        let top_log = Llvm.build_call log_fxn_num [| e2_cast |] "log_func"
330                        let bottom_log =  Llvm.build_call log_fxn_num [| e1_cast |] "log_f
331                        let eval_l = Llvm.build_fdiv top_log bottom_log "tmp" builder in
332                        L.build_fptosi eval_l i32_t "cast" builder
333             | _ -> raise( NotImplemented )
334             ) in log_types
335         else
336         let int_bop op =
337           (match op with
338              AST.Add     -> L.build_add
339            | AST.Sub     -> L.build_sub
```

```
340             | AST.Mult    -> L.build_mul
341             | AST.Div     -> L.build_sdiv
342             | AST.And     -> L.build_and
343             | AST.Or      -> L.build_or
344             | AST.Mod     -> L.build_srem
345             | AST.Equal   -> L.build_icmp L.Icmp.Eq
346             | AST.Neq     -> L.build_icmp L.Icmp.Ne
347             | AST.Less    -> L.build_icmp L.Icmp.Slt
348             | AST.Leq     -> L.build_icmp L.Icmp.Sle
349             | AST.Greater -> L.build_icmp L.Icmp.Sgt
350             | AST.Geq     -> L.build_icmp L.Icmp.Sge
351             | _ -> raise( IllegalType )
352           ) e1_new' e2_new' "tmp" builder in
353         let num_bop op =
354           (match op with
355              AST.Add     -> L.build_fadd
356             | AST.Sub     -> L.build_fsub
357             | AST.Mult    -> L.build_fmul
358             | AST.Div     -> L.build_fdiv
359             | AST.And     -> L.build_and
360             | AST.Mod     -> L.build_frem
361             | AST.Or      -> L.build_or
362             | AST.Equal   -> L.build_fcmp L.Fcmp.Oeq
363             | AST.Neq     -> L.build_fcmp L.Fcmp.One
364             | AST.Less    -> L.build_fcmp L.Fcmp.Olt
365             | AST.Leq     -> L.build_fcmp L.Fcmp.Ole
366             | AST.Greater -> L.build_fcmp L.Fcmp.Ogt
367             | AST.Geq     -> L.build_fcmp L.Fcmp.Oge
368             | _ -> raise( IllegalType )
369           ) e1_new' e2_new' "tmp" builder in
370         let build_ops_with_types t =
371           (match (t) with
372             A.Int -> int_bop op
373           | A.Num -> num_bop op
374           | A.Bool -> (match t_1, t_2 with
375                A.Num, A.Num -> num_bop op
376              | A.Int, A.Num -> num_bop op
377              | A.Num, A.Int -> num_bop op
378              | A.Int, A.Int -> int_bop op
379              | A.Bool, A.Bool -> int_bop op
380              | A.Symbol, A.Symbol -> L.build_icmp L.Icmp.Eq (L.build_pointercast e1_new'
381              | _, _ -> raise(Failure("Unsupported usage of comparison operator")))
382           | _ -> raise(Failure("binops can only take symbols, ints and nums"))
383           )
384              in (build_ops_with_types t)
385
386       | A.Unop(_, op, e) ->
387         let e' = expr builder e in
388         (match op with
389            AST.Neg -> L.build_neg
390          | AST.Not -> L.build_not) e' "tmp" builder
```

```
391       | A.Assign (A.Idl(t, s), e) -> (match t with
392                       A.Symbol -> let t_1 = get_type e in
393                         (match t_1 with
394                           A.Num -> let e_val = expr builder e in
395                             let (s_v, _, _) = (lookup s) in let s_v1 = L.build_load s_v s
396                             let e' = L.build_call const_symbol [| s_v1; e_val |] "symbolm
397                             ignore( L.build_store e' s_v builder); e'
398                         | A.Int -> let e_val = L.build_sitofp (expr builder e) num_t "cas
399                             let (s_v, _, _) = (lookup s) in let s_v1  =  L.build_load s_v
400                             let e' = L.build_call const_symbol [| s_v1; e_val |] "symbolm
401                             ignore( L.build_store e' s_v builder); e'
402                         | _ -> let e_val = expr builder e in
403                             let (s_v, _, _) = lookup s in
404                             ignore( L.build_store e_val s_v builder ); e_val )
405                       | A.Num -> let t_1 = get_type e in
406                             (match t_1 with
407                               A.Num -> let e_val = expr builder e in ignore( let (s_v, _,
408                                                                   L.build_store e_val s_v bui
409                             | A.Int -> let e_val = (L.build_sitofp (expr builder e) num_t
410                                           ignore( let (s_v, _, _) = (lookup s) in
411                                           L.build_store e_val s_v builder); e_val
412                             | _ -> raise(Failure("Unexpected type assigned to num"))
413                           )
414                       | _ -> let e' = expr builder e in ignore( let (s_v, _, _) = (lookup
415                                                                   L.build_store e' s_v builde
416
417
418       | A.Assign (A.ArrIdl(t, s, el), e) -> if Hashtbl.mem g_alloc s then (let pointer =
419       | A.Call (_, "print_int", [e]) | A.Call (_, "print_bool", [e]) ->
420         L.build_call printf_func [| int_format_str ; (expr builder e) |]
421           "printf" builder
422       | A.Call (_, "strcompare", [e1; e2]) -> L.build_call string_compare [| (expr builde
423           "strcmp" builder
424       | A.Call (_, "absInt", [e]) -> L.build_call abs_int [| (expr builder e) |]
425           "absint" builder
426       | A.Call (_, "absNum", [e]) -> L.build_call abs_num [| (expr builder e) |]
427           "absnum" builder
428       | A.Call (_, "value", [e]) ->
429         L.build_call value_symbol [|  (expr builder e) |]
430           "symbol_value" builder
431       | A.Call (_, "isConstant", [e]) ->
432         L.build_call symbol_const_check [|  (expr builder e) |]
433           "symbol_const" builder
434       | A.Call (_, "isInitialized", [e]) ->
435         L.build_call symbol_init_check [|  (expr builder e) |]
436           "symbol_init" builder
437       | A.Call (_, "left", [e]) ->
438         L.build_call symbol_left [|  (expr builder e) |]
439           "symbol_left_call" builder
440       | A.Call (_, "right", [e]) ->
441         L.build_call symbol_right [|  (expr builder e) |]
```

```
442           "symbol_right_call" builder
443       | A.Call (_, "operator", [e]) ->
444         L.build_call symbol_operator [|  (expr builder e) |]
445           "symbol_operator" builder
446       | A.Call (_, "print_num", [e]) ->
447         L.build_call printf_func [| float_format_str ; (expr builder e) |]
448           "printf" builder
449       | A.Call (_, "print", [e]) ->
450         L.build_call printf_func [| str_format_str ; (expr builder e) |]
451           "printf" builder
452       | A.Call (_, "printbig", [e]) ->
453         L.build_call printbig_func [| (expr builder e) |] "printbig" builder
454       | A.Call (_, f, act) ->
455         let (fdef, fdecl) = StringMap.find f function_decls in
456         let actuals = List.rev (List.map (expr builder) (List.rev act)) in
457         let result = (match fdecl.A.s_typ with A.Void -> ""
458                                              | _ -> f ^ "_result") in
459         L.build_call fdef (Array.of_list actuals) result builder
460     in
461
462      (* Invoke "f builder" if the current block doesn't already
463         have a terminal (e.g., a branch). *)
464      let add_terminal builder f =
465        match L.block_terminator (L.insertion_block builder) with
466          Some _ -> ()
467        | None -> ignore (f builder) in
468
469      (* Build the code for the given statement; return the builder for
470         the statement's successor *)
471      let rec stmt builder = function
472          A.Block(sl) -> List.fold_left stmt builder sl
473        | A.Expr(e) -> ignore (expr builder e); builder
474        | A.Return(e) ->
475          ignore (match fdecl.A.s_typ with
476              A.Void -> L.build_ret_void builder
477            | _ -> L.build_ret (expr builder e) builder
478          ); builder
479
480        | A.If (predicate, then_stmt, else_stmt) ->
481          let bool_val = expr builder predicate in
482          let merge_bb = L.append_block context "merge" the_function in
483
484          let then_bb = L.append_block context "then" the_function in
485          add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
486            (L.build_br merge_bb);
487
488          let else_bb = L.append_block context "else" the_function in
489          add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
490            (L.build_br merge_bb);
491
492          ignore (L.build_cond_br bool_val then_bb else_bb builder);
```

```
493         L.builder_at_end context merge_bb
494
495       | A.While (predicate, body) ->
496         let pred_bb = L.append_block context "while" the_function in
497         ignore (L.build_br pred_bb builder);
498
499         let body_bb = L.append_block context "while_body" the_function in
500         add_terminal (stmt (L.builder_at_end context body_bb) body)
501           (L.build_br pred_bb);
502
503         let pred_builder = L.builder_at_end context pred_bb in
504         let bool_val = expr pred_builder predicate in
505
506         let merge_bb = L.append_block context "merge" the_function in
507         ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
508         L.builder_at_end context merge_bb
509
510       | A.For(e1, e2, e3, body) -> stmt builder
511                                       ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body
512
513     in
514     (* Build the code for each statement in the function *)
515     let builder = stmt builder (A.Block stmt_list) in
516         add_terminal builder (match fdecl.A.s_typ with
517           A.Void -> L.build_ret_void
518         | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
519   in
520
521   List.iter build_function_body functions; the_module
```

A.4 damo.ml

```
1  (* Top-level of the Damo compiler: scan & parse the input,
2     check the resulting AST, generate LLVM IR, and dump the module *)
3
4  type action = Ast | LLVM_IR | Compile
5
6  let _ =
7    let action = if Array.length Sys.argv > 1 then
8      List.assoc Sys.argv.(1) [ (*("-a", Ast);*)        (* Print the AST only *)
9                                ("-l", LLVM_IR);  (* Generate LLVM, don't check *)
10                               ("-c", Compile) ] (* Generate, check LLVM IR *)
11   else Compile in
12   let lexbuf = Lexing.from_channel stdin in
13   let ast = Parser.program Scanner.token lexbuf in
14   (*Semant.check ast;*)
15   match action with
16     (*Ast -> print_string (Ast.string_of_program ast)*)
17     Ast -> ignore()
18   | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate (Semant.convert a
```

```
19    | Compile -> let m = Codegen.translate (Semant.convert ast) in
20      Llvm_analysis.assert_valid_module m;
21      print_string (Llvm.string_of_llmodule m)
```

A.5 semant.ml

```
1  (* Semantic checking for the damo compiler *)
2
3  open Ast
4  open Sast
5
6  module StringMap = Map.Make(String)
7
8  (* Semantic checking of a program. Returns void if successful,
9     throws an exception if something is wrong.
10
11    Check each global variable, then check each function *)
12 (*type env_var = {v_type: Sast.t; v_expr: Sast.expr; v_name: string}
13 type env_global = {sym_table: env_var StringMap.t;}
14 type env_function = {env_name: string; sym_table: env_var StringMap.t; env_ret_type: Sast
15 *)
16 let function_map = Hashtbl.create 10;;
17   (*maps for function name and type info, such as return type and length of paramter list
18 let function_map_type = Hashtbl.create 10;;
19 let function_map_length = Hashtbl.create 10;;
20 let function_map_formals = Hashtbl.create 10;;
21 let global_scope = Hashtbl.create 15;;
22
23 let convert program_list =
24   (*map for function name and list of variables, including formals and locals*)
25   let convert_type_tuple x = let (t, _) = x in match t with
26         Ast.Int    -> Sast.Int
27       | Ast.Bool   -> Sast.Bool
28       | Ast.Num    -> Sast.Num
29       | Ast.Symbol -> Sast.Symbol
30       | Ast.Void   -> Sast.Void
31       | Ast.String -> Sast.String
32
33   in
34
35   let convert_type = function
36         Ast.Int    -> Sast.Int
37       | Ast.Bool   -> Sast.Bool
38       | Ast.Num    -> Sast.Num
39       | Ast.Symbol -> Sast.Symbol
40       | Ast.Void   -> Sast.Void
41       | Ast.String -> Sast.String
42
43   in
44
```

```
45  let extract_type s_expr = match s_expr with
46        Sast.Id(t, _)              -> t
47      | Sast.Noexpr(t)             -> t
48      | Sast.Binop(t, _, _ ,_)     -> t
49      | Sast.IntLit(t, _)          -> t
50      | Sast.BoolLit(t, _)         -> t
51      | Sast.StringLit(t, _)       -> t
52      | Sast.NumLit(t, _)          -> t
53      | Sast.ArrId(t, _, _)        -> t
54      | Sast.Unop(t, _, _)         -> t
55      | Sast.Call(t, _, _)         -> t
56      | _                          -> raise(Failure("cannot extract type for this"))
57  in
58
59  (*Hashtbl.add global_scope "test" (Ast.String, 0);*)
60
61  Hashtbl.add function_map_formals "print" [(Ast.String, "x")];
62  Hashtbl.add function_map_formals "print_int" [(Ast.Int, "x")];
63  Hashtbl.add function_map_formals "print_bool" [(Ast.Bool, "x")];
64  Hashtbl.add function_map_formals "print_num" [(Ast.Num, "x")];
65  Hashtbl.add function_map_formals "left" [(Ast.Symbol, "x")];
66  Hashtbl.add function_map_formals "right" [(Ast.Symbol, "x")];
67  Hashtbl.add function_map_formals "operator" [(Ast.Symbol, "x")];
68  Hashtbl.add function_map_formals "isInitialized" [(Ast.Symbol, "x")];
69  Hashtbl.add function_map_formals "value" [(Ast.Symbol, "x")];
70  Hashtbl.add function_map_formals "strcompare" [(Ast.String, "x"); (Ast.String, "y")];
71  Hashtbl.add function_map_formals "absInt" [(Ast.Int, "x")];
72  Hashtbl.add function_map_formals "absNum" [(Ast.Num, "x")];
73  Hashtbl.add function_map_formals "isConstant" [(Ast.Symbol, "x")];
74  Hashtbl.add function_map_formals "operator" [(Ast.Symbol, "x")];
75
76  Hashtbl.add function_map_length "print" 1;
77  Hashtbl.add function_map_length "print_int" 1;
78  Hashtbl.add function_map_length "print_bool" 1;
79  Hashtbl.add function_map_length "print_num" 1;
80  Hashtbl.add function_map_length "left" 1;
81  Hashtbl.add function_map_length "right" 1;
82  Hashtbl.add function_map_length "operator" 1;
83  Hashtbl.add function_map_length "isInitialized" 1;
84  Hashtbl.add function_map_length "value" 1;
85  Hashtbl.add function_map_length "strcompare" 2;
86  Hashtbl.add function_map_length "absInt" 1;
87  Hashtbl.add function_map_length "absNum" 1;
88  Hashtbl.add function_map_length "operator" 1;
89  Hashtbl.add function_map_length "isConstant" 1;
90
91  Hashtbl.add function_map_type "print" Ast.Void;
92  Hashtbl.add function_map_type "print_int" Ast.Void;
93  Hashtbl.add function_map_type "print_bool" Ast.Void;
94  Hashtbl.add function_map_type "print_num" Ast.Void;
95  Hashtbl.add function_map_type "left" Ast.Symbol;
```

```
96    Hashtbl.add function_map_type "right" Ast.Symbol;
97    Hashtbl.add function_map_type "operator" Ast.String;
98    Hashtbl.add function_map_type "isInitialized" Ast.Int;
99    Hashtbl.add function_map_type "value" Ast.Num;
100   Hashtbl.add function_map_type "strcompare" Ast.Int;
101   Hashtbl.add function_map_type "absInt" Ast.Int;
102   Hashtbl.add function_map_type "absNum" Ast.Num;
103   Hashtbl.add function_map_type "isConstant" Ast.Int;
104   Hashtbl.add function_map_type "operator" Ast.String;
105
106   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.String, 0); Hashtbl.add
107   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.Int, 0); Hashtbl.add fu
108   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.Bool, 0); Hashtbl.add f
109   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.Num, 0); Hashtbl.add fu
110   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.Symbol, 0); Hashtbl.add
111   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.Symbol, 0); Hashtbl.add
112   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.Symbol, 0); Hashtbl.add
113   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.Symbol, 0); Hashtbl.add
114   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.String, 0); Hashtbl.add
115   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.Int, 0); Hashtbl.add fu
116   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.Num, 0); Hashtbl.add fu
117   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.Symbol, 0); Hashtbl.add
118   let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" (Ast.Symbol, 0); Hashtbl.add
119
120
121
122   (*  let printing_functions = ["print" ; "print_int" ; "print_num" ; "print_bool"] in
123     let symbol_functions = ["left" ; "right" ; "operator" ; "isConstant" ] in
124     let built_in_functions = printing_functions @ symbol_functions in
125     let add_builtin fname = match fname with
126           "print" -> Hashtbl.add function_map_formals fname [(Ast.String, "x")];
127                      let new_map = Hashtbl.create 10;
128                      Hashtbl.add new_map "x" Ast.String;
129                      Hashtbl.add function_map fname new_map;
130                      Hashtbl.add function_map_length fname 1;
131                      Hashtbl.add function_map_type fname (Ast.Void)
132         | "print_int" ->
133                  ignore(Hashtbl.add function_map_formals fname [(Ast.Int, "x")]);
134                  ignore(let new_map = Hashtbl.create 10 in Hashtbl.add new_map "x" Ast.Int
135                  ignore(Hashtbl.add function_map_length fname 1);
136                  Hashtbl.add function_map_type fname (Ast.Void)
137
138         | "print_num" -> ignore(Hashtbl.add function_map_formals fname [(Ast.Num, "x")]); i
139
140         | "print_bool" -> ignore(Hashtbl.add function_map_formals fname [(Ast.Bool, "x")]);
141
142         | "left" -> ignore(Hashtbl.add function_map_formals fname [(Ast.Symbol, "x")]); ign
143
144         | "right" -> ignore(Hashtbl.add function_map_formals fname [(Ast.Symbol, "x")]); ig
145
146         | "operator" -> ignore(Hashtbl.add function_map_formals fname [(Ast.Symbol, "x")]);
```

```
147
148         | "isConstant" -> ignore(Hashtbl.add function_map_formals fname [(Ast.Symbol, "x")]
149
150     in List.iter (fun x -> add_builtin x) built_in_functions in
151     *)
152     let extract_type_lvalue lvalue = match lvalue with
153           Sast.Idl(t, _)            -> t
154         | Sast.ArrIdl(t, _, _)      -> t
155     in
156
157     let check_assign lvaluet rvaluet err = match lvaluet, rvaluet with
158         (Sast.Symbol, Sast.Int) -> Sast.Symbol
159       | (Sast.Symbol, Sast.Num) -> Sast.Symbol
160       | (Sast.Num, Sast.Int) -> Sast.Num
161       | (_, _) -> if lvaluet = rvaluet then lvaluet else raise err in
162
163     (* global scope record for all global variables *)
164
165     let type_of_identifier s env =
166
167         try Hashtbl.find env s
168         with Not_found -> try Hashtbl.find global_scope s with Not_found -> raise (Failure(
169
170     in
171
172     let find_func fname =
173         try Hashtbl.find function_map fname
174         with Not_found -> raise (Failure ("undeclared function"))
175     in
176
177     let rec check_expr_legal e env = let e' = expr env e in if extract_type e' = Sast.Int t
178
179
180     and check_lvalue env lvalue  = match lvalue with
181           Ast.Idl(s) -> Sast.Idl(convert_type_tuple(type_of_identifier s env), s)
182         | Ast.ArrIdl(s, el) -> List.iter (fun a -> ignore(check_expr_legal a env)) el; let
183
184     and expr env e = match e with
185           Ast.IntLit(i)    -> Sast.IntLit(Sast.Int, i)
186         | Ast.BoolLit(b)   -> Sast.BoolLit(Sast.Bool, b)
187         | Ast.NumLit(n)    -> Sast.NumLit(Sast.Num, n)
188         | Ast.StringLit(s) -> Sast.StringLit(Sast.String, s)
189         | Ast.Id(s) -> Sast.Id(convert_type_tuple (type_of_identifier s env), s)
190         | Ast.ArrId(a, el) -> List.iter (fun x -> ignore(check_expr_legal x env)) el; let e
191         | Ast.Binop(e1, op, e2) -> let e1' = expr env e1 and e2' = expr env e2 in let t1 =
192                 (match op with
193                 Ast.Add | Ast.Sub | Ast.Mult | Ast.Div | Ast.Mod | Ast.Exp | Ast.Log when (
194               | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div | Ast.Mod | Ast.Exp | Ast.Log when (
195               | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div | Ast.Mod | Ast.Exp | Ast.Log when t
196               | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div | Ast.Mod | Ast.Exp | Ast.Log when t
197               | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div | Ast.Exp | Ast.Log  when t1 = Sast.
```

```
198             | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div | Ast.Exp | Ast.Log  when t1 = Sast.
199             | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div | Ast.Exp | Ast.Log  when t1 = Sast.
200              | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div | Ast.Exp | Ast.Log  when t1 = Sast
201               | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div | Ast.Exp | Ast.Log  when t1 = Sas
202                   | Ast.Equal | Ast.Neq when t1 = t2 && (t1 = Sast.Int || t1 = Sast.Num)
203                   | Ast.Less | Ast.Leq | Ast.Greater | Ast.Geq when t1 = Sast.Int && t2 =
204                 | Ast.Less | Ast.Leq | Ast.Greater | Ast.Geq when t1 = Sast.Num && t2 = S
205                  | Ast.Equal when t1 = Sast.Symbol && t2 = Sast.Symbol -> Sast.Binop(Sast
206
207                   | Ast.And | Ast.Or when t1 = Sast.Bool && t2 = Sast.Bool -> Sast.Binop(
208                   | _ -> raise (Failure ("illegal binary operator "))
209         )
210       | Ast.Unop(op, e) -> let e' = expr env e in let t = extract_type e' in
211         (match op with
212            Ast.Neg when t = Sast.Int -> Sast.Unop(Sast.Int, op, e')
213          | Ast.Not when t = Sast.Bool -> Sast.Unop(Sast.Bool, op, e')
214          | _ -> raise (Failure ("illegal unary operator ")))
215       | Ast.Noexpr -> Sast.Noexpr(Sast.Void)
216       | Ast.Assign(lvalue, e) -> let new_lvalue = check_lvalue env lvalue in let lt = ext
217       | Ast.Call(fname, actuals) -> ignore(find_func fname); if List.length actuals != Ha
218           raise (Failure ("incorrect number of arguments"))
219         else
220           let new_actuals = List.map (fun e -> expr env e) actuals in
221           List.iter2 (fun (ft, _) e -> let et = extract_type e in
222                       ignore (check_assign (convert_type ft) et
223                                 (Failure ("illegal actual argument found, wrong type"))))
224             (try Hashtbl.find function_map_formals fname with
225              | Not_found -> raise(Failure("function not defined in formals map"))) new_ac
226         Sast.Call(convert_type (try Hashtbl.find function_map_type fname with Not_found -
227
228   in
229
230
231   let report_duplicate_map var env =
232     if Hashtbl.mem env var = false then true else raise(Failure("duplicate found"));
233
234   in
235   (* Raise an exception if the given list has a duplicate *)
236   let report_duplicate exceptf list =
237     let rec helper = function
238         n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
239       | _ :: t -> helper t
240       | [] -> ()
241     in helper (List.sort compare list)
242   in
243
244   (* Raise an exception if a given binding is to a void type *)
245   let check_not_void t = match t with
246       Ast.Void -> raise (Failure ("can't have void variable"))
247     | _ -> ()
248   in
```

```
249
250
251   let check_not_void_map _ argTwo  = match argTwo with
252       (Ast.Void, _) -> raise(Failure("cannot have void variable type"))
253     | (_, _) -> ()
254   in
255
256     let check_vdecl = function
257       Ast.Decl(t,name) ->  ignore(report_duplicate_map name global_scope); ignore(Hashtbl
258     | Ast.ArrDecl(t, n, l) -> ignore(report_duplicate_map n global_scope); ignore(Hashtbl
259
260   in
261
262   let check_vdecl_function function_name function_line = match function_line with
263          Ast.Decl(t,name) -> let f_map = (try Hashtbl.find function_map function_name wit
264     | Ast.ArrDecl(t, n, l) -> let f_map = (try Hashtbl.find function_map function_name wi
265
266   in
267
268   let check_bool_expr env e = if extract_type (expr env e) <> Sast.Bool
269       then raise (Failure ("expected boolean expression "))
270       else () in
271
272     (* verify a statement or throw an exception *)
273   let rec stmt env fname s = match s with
274         Ast.Expr(e) -> Sast.Expr(expr env e)
275       | Ast.Block(sl) -> let sl' = List.map (fun a -> stmt env fname a) sl in Sast.Block(
276       | Ast.Return(e) -> if fname = "" then raise(Failure("can't have return type outside
277       | Ast.If(p, b1, b2) -> check_bool_expr env p; Sast.If(expr env p, stmt env fname b1
278       | Ast.For(e1, e2, e3, st) -> check_bool_expr env e2; Sast.For(expr env e1, expr env
279       | Ast.While(p, s) -> check_bool_expr env p; Sast.While(expr env p, stmt env fname s
280
281   in
282   let check_stmt function_name program_unit =
283         if function_name = "" then stmt global_scope "" program_unit else stmt (try Hasht
284
285   in
286
287   let get_function_type function_name function_line = match function_line with
288        Ast.VarUnit(s) -> Sast.VarUnit(check_vdecl_function function_name s)
289      | Ast.StmtUnit(st) -> Sast.StmtUnit(check_stmt function_name st)
290      | _ -> raise (Failure("this declaration is not defined"))
291   in
292
293   (*let add_formals formal fname = match formal with
294         Ast.Decl(t, n) -> let f_map = (try Hashtbl.find function_map fname with Not_found
295       | Ast.ArrDecl(t, n, el) -> let f_map = (try Hashtbl.find function_map fname with No
296
297   let add_hash x = match x with
298         Ast.Decl(t, n) -> (t, n)
299       | Ast.ArrDecl(t, n, _) -> (t, n)
```

```
300   in
301
302   let update_maps fd =
303         ignore(Hashtbl.add function_map_type fd.fname fd.typ);
304         (*ignore(Hashtbl.add function_map_formals fd.fname fd.formals;*)
305         let map = List.map (fun x -> add_hash x) fd.formals in Hashtbl.add function_map_f
306         ignore(Hashtbl.add function_map fd.fname (Hashtbl.create 20));
307         (*List.iter (fun n -> ignore(add_formals n fd.fname)) fd.formals;*)
308         Hashtbl.add function_map_length fd.fname (List.length fd.formals)
309   in
310
311   let extract_name formal = match formal with
312         Ast.Decl(_, n) -> n
313       | Ast.ArrDecl(_, n, _) -> n
314   in
315
316   let rec resolve_body name body = match body with
317           [] -> []
318         | head::tail -> let r = get_function_type name head in r::(resolve_body name tail)
319   in
320
321   let check_not_void_general formal = match formal with
322        Ast.Decl(t, _) -> check_not_void t
323      | Ast.ArrDecl(t, _, _) -> check_not_void t
324
325   in
326
327   let check_fdecl fd =
328         ignore(if Hashtbl.mem function_map fd.fname = false then update_maps fd else rais
329
330         report_duplicate (fun _ -> "duplicate formal") (List.map (fun x -> extract_name x
331         List.iter (fun n -> check_not_void_general n) fd.formals;
332         (*resolve all formals, resolve function body, return a Sast type of FuncUnit with
333         let func_formals = List.map (fun a -> check_vdecl_function fd.fname a) fd.formals
334         new_body = resolve_body fd.fname fd.body in let new_fdecl = {s_typ=convert_type f
335
336   in
337
338   let get_type = function
339         Ast.VarUnit(s)  -> Sast.VarUnit(check_vdecl s)
340       | Ast.FuncUnit(fd) -> check_fdecl fd
341       | Ast.StmtUnit(st) -> Sast.StmtUnit(check_stmt "" st)
342   in
343
344   List.map get_type program_list;
345   (*let rec check_types program_list = match program_list with
346         [] -> []
347       | head::tail -> let r = get_type head in r::(check_types tail)
348   in
349
350   check_types program_list *)
```

A.6 parser.mly

```
/* Ocamlyacc parser for MicroC */

%{
open Ast
%}

%token SEMI COMMA
%token LBRACKET RBRACKET LPAREN RPAREN LBRACE RBRACE
%token PLUS MINUS TIMES DIVIDE EXP LOG MOD
%token NOT AND OR
%token ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token IF ELSEIF ELSE FOR WHILE
%token INT BOOL NUM STRING SYMBOL VOID
%token DEF RETURN COLON
%token TRUE FALSE
%token <string> STRING_LITERAL
%token <float> NUM_LITERAL
%token <int> INT_LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc ELSEIF
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right EXP
%left LOG
%right MOD
%right NOT NEG

%start program
%type <Ast.program> program

%%

program:
  program_sequence EOF { List.rev $1 }

program_sequence:
  /* nothing */ { [] }
  | program_sequence vdecl { $2 @ $1 }
  | program_sequence fdecl { FuncUnit($2) :: $1 }
```

```
50    | program_sequence stmt { StmtUnit($2) :: $1 }
51
52  fdecl:
53     DEF ID LPAREN formals_opt RPAREN COLON typ LBRACE program_sequence RBRACE
54      {
55        {
56          typ = $7;
57          fname = $2;
58          formals = $4;
59          body = List.rev $9
60        }
61      }
62
63  formals_opt:
64      /* nothing */ { [] }
65    | formal_list   { List.rev $1 }
66
67  formal_list:
68      typ ID                   { [Decl($1,$2)] }
69    | formal_list COMMA typ ID { Decl($3,$4) :: $1 }
70
71  typ:
72      INT { Int }
73    | NUM { Num }
74    | BOOL { Bool }
75    | STRING { String }
76    | SYMBOL { Symbol }
77    | VOID { Void }
78
79  vdecl:
80      typ ID SEMI { [VarUnit(Decl($1, $2))] }
81    | typ ID ASSIGN expr SEMI { [StmtUnit(Expr(Assign(Idl($2), $4))) ; VarUnit(Decl($1, $2)
82    | typ ID LBRACKET brackets RBRACKET SEMI { [VarUnit(ArrDecl($1, $2, List.rev $4))] }
83
84  brackets:
85      expr { [$1] }
86    | brackets RBRACKET LBRACKET expr { $4 :: $1 }
87
88  stmt_list:
89      /* nothing */  { [] }
90    | stmt_list stmt { $2 :: $1 }
91
92  stmt:
93      expr SEMI { Expr $1 }
94    | RETURN SEMI { Return Noexpr }
95    | RETURN expr SEMI { Return $2 }
96    | LBRACE stmt_list RBRACE { Block(List.rev $2) }
97    | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
98    | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
99    | IF LPAREN expr RPAREN stmt else_stmt    { If($3, $5, $6) }
100   | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
```

```
101       { For($3, $5, $7, $9) }
102     | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
103
104   else_stmt:
105     ELSEIF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
106     | ELSEIF LPAREN expr RPAREN stmt else_stmt { If($3, $5, $6) }
107     | ELSEIF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
108   expr_opt:
109       /* nothing */ { Noexpr }
110     | expr          { $1 }
111
112   expr:
113       INT_LITERAL      { IntLit($1) }
114     | NUM_LITERAL      { NumLit($1) }
115     | STRING_LITERAL   { StringLit($1) }
116     | TRUE             { BoolLit(true) }
117     | FALSE            { BoolLit(false) }
118     | ID               { Id($1) }
119     | arrid            { $1 }
120     | expr PLUS   expr { Binop($1, Add,   $3) }
121     | expr MINUS  expr { Binop($1, Sub,   $3) }
122     | expr TIMES  expr { Binop($1, Mult,  $3) }
123     | expr DIVIDE expr { Binop($1, Div,   $3) }
124     | expr EXP    expr { Binop($1, Exp, $3) }
125     | expr LOG    expr { Binop($1, Log, $3) }
126     | expr MOD    expr { Binop($1, Mod, $3) }
127     | expr EQ     expr { Binop($1, Equal, $3) }
128     | expr NEQ    expr { Binop($1, Neq,   $3) }
129     | expr LT     expr { Binop($1, Less,  $3) }
130     | expr LEQ    expr { Binop($1, Leq,   $3) }
131     | expr GT     expr { Binop($1, Greater, $3) }
132     | expr GEQ    expr { Binop($1, Geq,   $3) }
133     | expr AND    expr { Binop($1, And,   $3) }
134     | expr OR     expr { Binop($1, Or,    $3) }
135     | MINUS expr %prec NEG { Unop(Neg, $2) }
136     | NOT expr         { Unop(Not, $2) }
137     | ID ASSIGN expr   { Assign(Idl($1), $3) }
138     | larrid ASSIGN expr { Assign($1, $3) }
139     | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
140     | LPAREN expr RPAREN { $2 }
141
142   larrid:
143     ID LBRACKET brackets RBRACKET { ArrIdl($1, List.rev $3) }
144
145   arrid:
146     ID LBRACKET brackets RBRACKET { ArrId($1, List.rev $3) }
147
148   actuals_opt:
149       /* nothing */ { [] }
150     | actuals_list  { List.rev $1 }
151
```

```
152 actuals_list:
153     expr                    { [$1] }
154   | actuals_list COMMA expr { $3 :: $1 }
```

A.7 scanner.mll

```
1  (* Ocamllex scanner for MicroC *)
2
3  { open Parser
4
5  (* TODO is this necessary? *)
6  let unescape s =
7          Scanf.sscanf ( "\"" ^s ^ "\"" ) "%S%!" (fun x -> x)
8  }
9
10 (* Complex regular expressions *) (* TODO are any of these unnecessary? *)
11 let alpha = ['a'-'z' 'A'-'Z']
12 let ascii = ([' '-'!' '#'-'[' ']'-'~' ])
13 let digit = ['0'-'9']
14 let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
15 let escape_char = ''' (escape) '''
16 let id = (alpha | '_') (alpha | digit | '_')*
17 let string_re = '"' ((ascii|escape)* as s) '"'
18 let whitespace = [' ' '\t' '\r']
19 let num_re = (digit+ '.' digit*) | ('.' digit+)
20 let int_re = digit+
21
22 rule token = parse
23   [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
24 | "/*"     { multi_comment lexbuf }           (* Multiline comments *)
25 | "//"                { single_comment lexbuf }        (* Single line comments *)
26 | '('      { LPAREN }
27 | ')'      { RPAREN }
28 | '{'      { LBRACE }
29 | '}'      { RBRACE }
30 | ';'      { SEMI }
31 | ':'                { COLON } (* NEW for our function syntax *)
32 | ','      { COMMA }
33 | '['           { LBRACKET }
34 | ']'           { RBRACKET }
35
36 (* Operators *)
37 | '+'      { PLUS }
38 | '-'      { MINUS }
39 | '*'      { TIMES }
40 | '/'      { DIVIDE }
41 | '^'                { EXP }
42 | '_'                { LOG }
43 | '%'                { MOD }
44 | '='      { ASSIGN }
```

```
45  | "=="      { EQ }
46  | "!="      { NEQ }
47  | '<'       { LT }
48  | "<="      { LEQ }
49  | ">"       { GT }
50  | ">="      { GEQ }
51  | "&&"      { AND }
52  | "||"      { OR }
53  | "!"       { NOT }
54
55  (* Control flow, functions *)
56  | "def"                { DEF }
57  | "if"      { IF }
58  | "else"    { ELSE }
59  | "elseif" { ELSEIF }
60  | "for"     { FOR }
61  | "while"   { WHILE }
62  | "return" { RETURN }
63
64  (* Data types *)
65  | "int"     { INT }
66  | "bool"    { BOOL }
67  | "void"    { VOID }
68  | "string"         { STRING }
69  | "num"                 { NUM }
70  | "symbol"         { SYMBOL }
71
72  (* Literals *)  (* TODO can we replace all instances of Literal with Int_Literal? *)
73  | "true"    { TRUE }
74  | "false"   { FALSE }
75  | int_re as lxm { INT_LITERAL(int_of_string lxm) }
76  | num_re as lxm { NUM_LITERAL(float_of_string lxm) }
77  | string_re         { STRING_LITERAL(unescape s) }
78
79  (* Others *)
80  | id as lxm { ID(lxm) }
81  | eof { EOF }
82  | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
83
84  and multi_comment = parse
85    "*/" { token lexbuf }
86  | _    { multi_comment lexbuf }
87
88  and single_comment = parse
89    ['\n' '\r'] { token lexbuf }
90  | _ {single_comment lexbuf }
```

A.8 stdlib.dm

```
def streq(string a, string b) : bool {
        return strcompare(a, b) == 0;
}

def eval(symbol a) : num {
        num leftValue;
        num rightValue;
        num result;
        string op;
        if (isConstant(a) == 1){
                result = value(a);
        }
        elseif (isInitialized(a) != 1){
                print("Evaluating an uninitialized symbol");
        }
        else {
                op = operator(a);
                if (streq(op, "PLUS")){
                        leftValue = eval(left(a));
                        rightValue = eval(right(a));
                        result = leftValue + rightValue;
                }
                elseif (streq(op, "MINUS")){
                        leftValue = eval(left(a));
                        rightValue = eval(right(a));
                        result = leftValue - rightValue;
                }
                elseif (streq(op, "TIMES")){
                        leftValue = eval(left(a));
                        rightValue = eval(right(a));
                        result = leftValue * rightValue;
                }
                elseif (streq(op, "DIVIDE")){
                        leftValue = eval(left(a));
                        rightValue = eval(right(a));
                        result = leftValue / rightValue;
                }
                elseif (streq(op, "EXP")){
                        leftValue = eval(left(a));
                        rightValue = eval(right(a));
                        result = leftValue ^ rightValue;
                }
                elseif (streq(op, "LOG")){
                        leftValue = eval(left(a));
                        rightValue = eval(right(a));
                        result = leftValue _ rightValue;
                }
                /*elseif (streq(op, "NEGATIVE")){
                        // TODO currently, we won't reach here
```

```
50                          leftValue = eval(left(a));
51                          result = - leftValue;
52                  }*/
53                  else {
54                          // Crash the program
55                          print("Unknown operator");
56                  }
57          }
58          return result;
59  }
60
61  num exponentialConstant = 2.71828;
62
63  def partialDerivative(symbol out, symbol in) : num {
64          num leftGrad;
65          num rightGrad;
66          string op;
67          num dadL = 0;
68          num dadR = 0;
69          num L;
70          num R;
71          num result;
72
73          if (isConstant(out) == 1){
74                  if (out == in){
75                          result = 1.0;
76                  }
77                  else{
78                          result = 0.0;
79                  }
80          }
81          elseif(isInitialized(out) != 1){
82                  print("Attempting to differentiate uninitialized symbol");
83          }
84          else{
85                  op = operator(out);
86                  if (0 == 1 /*streq(op, "NEGATIVE")*/){
87                          leftGrad = partialDerivative(left(out), in);
88                          dadL = -1;
89                          result = dadL * leftGrad;
90                  }
91                  else {
92                          leftGrad = partialDerivative(left(out), in);
93                          rightGrad = partialDerivative(right(out), in);
94
95                          if (streq(op, "PLUS")){
96                                  dadL = 1;
97                                  dadR = 1;
98                          }
99                          elseif (streq(op, "MINUS")){
100                                 dadL = 1;
```

```
                                dadR = 0.0 - 1.0;
                        }
                        elseif (streq(op, "TIMES")){
                                dadL = eval(right(out));
                                dadR = eval(left(out));
                        }
                        elseif (streq(op, "DIVIDE")){
                                dadL = 1 / eval(right(out));
                                dadR = 0 - eval(left(out)) / (eval(right(out)) ^ 2);
                        }
                        elseif (streq(op, "EXP")){
                                L = eval(left(out));
                                R = eval(right(out));
                                if (leftGrad != 0.0){
                                        dadL = R * (L ^ (R - 1));
                                }
                                if (rightGrad != 0.0){
                                        dadR = (exponentialConstant _ L) * (L ^ R);
                                }
                        }
                        elseif (streq(op, "LOG")){
                                L = eval(left(out));
                                R = eval(right(out));
                                if (leftGrad != 0.0){
                                        dadL = (L _ R) * (L _ exponentialConstant) / L;
                                }
                                if (rightGrad != 0.0){
                                        dadR = (L _ exponentialConstant) / R;
                                }
                        }
                        else {
                                // Crash the program
                                print("Should crash here - invalid symbol operation (part
                        }
                        result = dadL * leftGrad + dadR * rightGrad;
                }
        }
        return result;
}
```

A.9 printbig.c

```
/*
 *  A function illustrating how to link C code to code generated from LLVM
 */

#include <stdio.h>
#include <math.h>

```

```
8  /*
9   * Font information: one byte per row, 8 rows per character
10  * In order, space, 0-9, A-Z
11  */
12 static const char font[] = {
13   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
14   0x1c, 0x3e, 0x61, 0x41, 0x43, 0x3e, 0x1c, 0x00,
15   0x00, 0x40, 0x42, 0x7f, 0x7f, 0x40, 0x40, 0x00,
16   0x62, 0x73, 0x79, 0x59, 0x5d, 0x4f, 0x46, 0x00,
17   0x20, 0x61, 0x49, 0x4d, 0x4f, 0x7b, 0x31, 0x00,
18   0x18, 0x1c, 0x16, 0x13, 0x7f, 0x7f, 0x10, 0x00,
19   0x27, 0x67, 0x45, 0x45, 0x45, 0x7d, 0x38, 0x00,
20   0x3c, 0x7e, 0x4b, 0x49, 0x49, 0x79, 0x30, 0x00,
21   0x03, 0x03, 0x71, 0x79, 0x0d, 0x07, 0x03, 0x00,
22   0x36, 0x4f, 0x4d, 0x59, 0x59, 0x76, 0x30, 0x00,
23   0x06, 0x4f, 0x49, 0x49, 0x69, 0x3f, 0x1e, 0x00,
24   0x7c, 0x7e, 0x13, 0x11, 0x13, 0x7e, 0x7c, 0x00,
25   0x7f, 0x7f, 0x49, 0x49, 0x49, 0x7f, 0x36, 0x00,
26   0x1c, 0x3e, 0x63, 0x41, 0x41, 0x63, 0x22, 0x00,
27   0x7f, 0x7f, 0x41, 0x41, 0x63, 0x3e, 0x1c, 0x00,
28   0x00, 0x7f, 0x7f, 0x49, 0x49, 0x49, 0x41, 0x00,
29   0x7f, 0x7f, 0x09, 0x09, 0x09, 0x09, 0x01, 0x00,
30   0x1c, 0x3e, 0x63, 0x41, 0x49, 0x79, 0x79, 0x00,
31   0x7f, 0x7f, 0x08, 0x08, 0x08, 0x7f, 0x7f, 0x00,
32   0x00, 0x41, 0x41, 0x7f, 0x7f, 0x41, 0x41, 0x00,
33   0x20, 0x60, 0x40, 0x40, 0x40, 0x7f, 0x3f, 0x00,
34   0x7f, 0x7f, 0x18, 0x3c, 0x76, 0x63, 0x41, 0x00,
35   0x00, 0x7f, 0x7f, 0x40, 0x40, 0x40, 0x40, 0x00,
36   0x7f, 0x7f, 0x0e, 0x1c, 0x0e, 0x7f, 0x7f, 0x00,
37   0x7f, 0x7f, 0x0e, 0x1c, 0x38, 0x7f, 0x7f, 0x00,
38   0x3e, 0x7f, 0x41, 0x41, 0x41, 0x7f, 0x3e, 0x00,
39   0x7f, 0x7f, 0x11, 0x11, 0x11, 0x1f, 0x0e, 0x00,
40   0x3e, 0x7f, 0x41, 0x51, 0x71, 0x3f, 0x5e, 0x00,
41   0x7f, 0x7f, 0x11, 0x31, 0x79, 0x6f, 0x4e, 0x00,
42   0x26, 0x6f, 0x49, 0x49, 0x4b, 0x7a, 0x30, 0x00,
43   0x00, 0x01, 0x01, 0x7f, 0x7f, 0x01, 0x01, 0x00,
44   0x3f, 0x7f, 0x40, 0x40, 0x40, 0x7f, 0x3f, 0x00,
45   0x0f, 0x1f, 0x38, 0x70, 0x38, 0x1f, 0x0f, 0x00,
46   0x1f, 0x7f, 0x38, 0x1c, 0x38, 0x7f, 0x1f, 0x00,
47   0x63, 0x77, 0x3e, 0x1c, 0x3e, 0x77, 0x63, 0x00,
48   0x00, 0x03, 0x0f, 0x78, 0x78, 0x0f, 0x03, 0x00,
49   0x61, 0x71, 0x79, 0x5d, 0x4f, 0x47, 0x43, 0x00
50 };
51
52 void printbig(int c)
53 {
54   int index = 0;
55   int col, data;
56   if (c >= '0' && c <= '9') index = 8 + (c - '0') * 8;
57   else if (c >= 'A' && c <= 'Z') index = 88 + (c - 'A') * 8;
58   do {
```

```
59      data = font[index++];
60      for (col = 0 ; col < 8 ; data <<= 1, col++) {
61        char d = data & 0x80 ? 'X' : ' ';
62        putchar(d); putchar(d);
63      }
64      putchar('\n');
65    } while (index & 0x7);
66  }
67
68
69  #ifdef BUILD_TEST
70  int main()
71  {
72    char s[] = "HELLO WORLD09AZ";
73    char *c;
74    for ( c = s ; *c ; c++) printbig(*c);
75  }
76  #endif
```

A.10 symbol.c

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   struct symbol {
5           struct symbol *left;
6           struct symbol *right;
7           int isConstant;
8           int isInitialized;
9           double value;
10          char *operator;
11  };
12
13  char *operator(struct symbol *a){
14          return a->operator;
15  }
16
17  struct symbol *left(struct symbol *a){
18          return a->left;
19  }
20
21  struct symbol *right(struct symbol *a){
22          return a->right;
23  }
24
25  int isConstant(struct symbol *a){
26          return a->isConstant;
27  }
28
29  int isInitialized(struct symbol *a){
```

```
30          return a->isInitialized;
31  }
32
33  double value(struct symbol *a){
34          return a->value;
35  }
36
37  struct symbol *createSymbol(){
38          struct symbol *a = malloc(sizeof(struct symbol));
39          if (a == 0){
40                  printf("Malloc failed in createSymbol");
41                  exit(1);
42          }
43          a->left = 0;
44          a->right = 0;
45          a->isConstant = 0;
46          a->isInitialized = 0;
47          a->value = 0.0;
48          return a;
49  }
50
51  struct symbol *createRoot(struct symbol *l, struct symbol *r, char *op){
52          struct symbol *a = createSymbol();
53          a->left = l;
54          a->right = r;
55          a->operator = op;
56          a->isInitialized = 1;
57          return a;
58  }
59
60  struct symbol *setSymbolValue(struct symbol *a, double val){
61          a->value = val;
62          a->isConstant = 1;
63          a->isInitialized = 1;
64          a->left = 0;
65          a->right = 0;
66          return a;
67  }
```

A.11 testall.sh

```
1  #!/bin/sh
2
3  # Regression testing script for DAMO
4  # Step through a list of files
5  #  Compile, run, and check the output of each expected-to-work test
6  #  Compile and check the error of each expected-to-fail test
7
8  # Path to the LLVM interpreter
9  LLI="lli"
```

```
10 #LLI="/usr/local/opt/llvm/bin/lli"
11
12 # Path to the LLVM compiler
13 LLC="llc"
14
15 # Path to the C compiler
16 CC="cc"
17
18 # Path to the damo compiler.  Usually "./damo.native"
19 # Try "_build/damo.native" if ocamlbuild was unable to create a symbolic
   ↪ link.
20 DAMO="./damo.native"
21 #DAMO="_build/damo.native"
22
23 # Set time limit for all operations
24 ulimit -t 30
25
26 globallog=testall.log
27 rm -f $globallog
28 error=0
29 globalerror=0
30
31 keep=0
32
33 Usage() {
34     echo "Usage: testall.sh [options] [.dm files]"
35     echo "-k    Keep intermediate files"
36     echo "-h    Print this help"
37     exit 1
38 }
39
40 SignalError() {
41     if [ $error -eq 0 ] ; then
42         echo "FAILED"
43         error=1
44     fi
45     echo "  $1"
46 }
47
48 # Compare <outfile> <reffile> <difffile>
49 # Compares the outfile with reffile.  Differences, if any, written to
   ↪ difffile
50 Compare() {
51     generatedfiles="$generatedfiles $3"
52     echo diff -b $1 $2 ">" $3 1>&2
53     diff -b "$1" "$2" > "$3" 2>&1 || {
54         SignalError "$1 differs"
55         echo "FAILED $1 differs from $2" 1>&2
56     }
57 }
58
```

```
59 # Run <args>
60 # Report the command, run it, and report any errors
61 Run() {
62     echo $* 1>&2
63     eval $* || {
64         SignalError "$1 failed on $*"
65         return 1
66     }
67 }
68
69 # RunFail <args>
70 # Report the command, run it, and expect an error
71 RunFail() {
72     echo $* 1>&2
73     eval $* && {
74         SignalError "failed: $* did not report an error"
75         return 1
76     }
77     return 0
78 }
79
80 Check() {
81     error=0
82     basename=`echo $1 | sed 's/.*\\///
83                               s/.dm//'`
84     reffile=`echo $1 | sed 's/.dm$//'`
85     basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
86
87     echo -n "$basename..."
88
89     echo 1>&2
90     echo "###### Testing $basename" 1>&2
91
92     generatedfiles=""
93
94     generatedfiles="$generatedfiles ${basename}.dml ${basename}.ll
       ↪ ${basename}.s ${basename}.exe ${basename}.out" &&
95     # Prepend standard library
96     Run "cat" "stdlib.dm" ">" "${basename}.dml" &&
97     Run "cat" $1 ">>" "${basename}.dml" &&
98     # Proceed with compilation
99     Run "$DAMO" "<" "${basename}.dml" ">" "${basename}.ll" &&
100    #Run "$DAMO" "<" $1 ">" "${basename}.ll"
101    Run "$LLC" "${basename}.ll" ">" "${basename}.s" &&
102    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "printbig.o"
       ↪ "symbol.o" "-lm"&&
103    Run "./${basename}.exe" > "${basename}.out" &&
104    Compare ${basename}.out ${reffile}.out ${basename}.diff
105
106    # Report the status and clean up the generated files
107
```

```
108     if [ $error -eq 0 ] ; then
109         if [ $keep -eq 0 ] ; then
110             rm -f $generatedfiles
111         fi
112         echo "OK"
113         echo "###### SUCCESS" 1>&2
114     else
115         echo "###### FAILED" 1>&2
116         globalerror=$error
117     fi
118 }
119
120
121 CheckFail() {
122     error=0
123     basename=`echo $1 | sed 's/.*\\///
124                               s/.dm//'`
125     reffile=`echo $1 | sed 's/.dm$//'`
126     basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
127
128     echo -n "$basename..."
129
130     echo 1>&2
131     echo "###### Testing $basename" 1>&2
132
133     generatedfiles=""
134
135     generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
136     RunFail "$DAMO" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
137     Compare ${basename}.err ${reffile}.err ${basename}.diff
138
139     # Report the status and clean up the generated files
140
141     if [ $error -eq 0 ] ; then
142         if [ $keep -eq 0 ] ; then
143             rm -f $generatedfiles
144         fi
145         echo "OK"
146         echo "###### SUCCESS" 1>&2
147     else
148         echo "###### FAILED" 1>&2
149         globalerror=$error
150     fi
151 }
152
153 while getopts kdpsh c; do
154     case $c in
155         k) # Keep intermediate files
156             keep=1
157             ;;
158         h) # Help
```

```
159             Usage
160             ;;
161     esac
162 done
163
164 shift `expr $OPTIND - 1`
165
166 LLIFail() {
167   echo "Could not find the LLVM interpreter \"$LLI\"."
168   echo "Check your LLVM installation and/or modify the LLI variable in
    ↪  testall.sh"
169   exit 1
170 }
171
172 which "$LLI" >> $globallog || LLIFail
173
174 if [ ! -f printbig.o ]
175 then
176     echo "Could not find printbig.o"
177     echo "Try \"make printbig.o\""
178     exit 1
179 fi
180
181 if [ $# -ge 1 ]
182 then
183     files=$@
184 else
185     # TODO new directory for tests
186     files="tests/test-*.dm tests/fail-*.dm"
187 fi
188
189 for file in $files
190 do
191     case $file in
192         *test-*)
193             Check $file 2>> $globallog
194             ;;
195         *fail-*)
196             CheckFail $file 2>> $globallog
197             ;;
198         *)
199             echo "unknown file type $file"
200             globalerror=1
201             ;;
202     esac
203 done
204
205 exit $globalerror
```