DCL

Project Manager: William Essilfie (wke2102)
System Architect: Craig Rhodes (cdr2139)
Language Guru: Ashutosh Nanda (an2655)
Tester: Chang Liu (cl3043)

Chapter 1: Introduction

Dynamic Callback Language (DCL) is an event-driven programming language that can be used for a variety of purposes. This language was designed to make programming both easy and educational for new programmers as well as to make handling special cases even easier for more advanced developers. Syntactically, DCL is a mix of Java and C. This allows users to access many of the low-level systems of the C language while also having the nice additions of easier ways for string concatenation as well as easier to read code. A lot of this is made possible via DCL's compilation to LLVM IR. LLVM IR, a cross-platform runtime environment, is useful because it will allow this language to run on any system (ex. Windows, Linux, MacOS, etc.) that has an LLVM port. The main feature of DCL is its event-driven callback system. Essentially, callbacks allows variables to have global sets of instructions that need to be executed depending on the state of themselves and other variables; these are introduced by the **buteverytime** keyword (see LRM).

Programs written in DCL have a formatting close to that of a typical scripting language such as Python. This means that instead of using classes and objects typical in Java, DCL consists of various methods and at the end of the file a user can specify which methods to call. In addition, DCL has access to a few C functions such as file I/O.

Background

Scripting languages are generally considered to be languages meant for writing code generally expected to be only a few thousand lines of code on the high end. Some examples of this include Python, AWK (co-created by former Columbia Professor Alfred Aho), and Bash. While many scripting languages are built for a singular specific purpose, many are created to be for general purpose use. DCL can be considered to be a general-purpose language.

Chapter 2: Language Tutorial

Language Compilation

To get started with compiling and running DCL files, there is an initial setup process one must go through. DCL's compiler was written on a Docker Image using Ubuntu 15.10. For an easy installation, a user must first download Docker from https://www.docker.com/community-edition Then, the user can clone the DCL repository from

https://github.com/PLT-DCL/dcl. Once downloaded, the user should change directories into the src using "cd src" to write any .dcl files they want to run. Then they should change directories back to the docker folder by typing "cd .. && cd docker" into their command line system. Once in the file, they should type "run [name of file].dcl." This will pull the DCL Docker image and then after compiling the compiler code, will run the user's file. The output will be displayed and visible from the user's command line system. Any changes made in the src directory within the docker container are reflected outside the container and vice versa.

Using the Compiler

To run a file in DCL is quite easy when using the docker image. To explain this, an example is provided below:

```
./dcl.native < float.dcl > float.ll
llc < float.ll > float.s
cc -o float float.s externalcalls.o -lm ./float
```

Chapter 3: Language Resource Manual

Introduction

DCL is an event-driven, non object-oriented language. The language is syntactically similar to Java and compiles down to to LLVM. The benefit to using LLVM is that it will allow DCL to include automatic garbage collection. DCL is a strongly typed programming language. As a result, at compile time, the language will prevent runtime errors based on variable type.

Lexical Conventions

Comments

DCL supports the comment style: /* comments */ for both one-line and multiline comments. /* presents the beginning of a coment and */ terminates that comment. All the content inside /*...*/ is treated as comments instead of part of the code that is compiled.

Examples

```
/* this is a one-line comment */
```

```
/* and now
   This is Multiline comment */
```

Identifiers

Identifiers are characters that can be used to name variables and functions in DCL. Those characters can be alphabetic letters, decimal digits, and the underscore character. The first character is not allowed to be a decimal digit. In DCL, uppercase and lowercase letters are different, so var1 and var1 are considered two identifiers. The identifiers used for variables and functions cannot be the same as the reserved keywords, otherwise an error will be given.

Keywords

The following identifiers are reserved as keywords and cannot be used for variables and functions, or any other purpose:

buteverytime for while return int double void string else true false print_line print read write # exp_int exp_dbl add_str

Constants

Constants are the fixed values that the program cannot alter during the execution. They can be of any primitive data type defined in DCL, such as integer constants, double constants, character constants. String constants are also included in DCL.

Integer Constant

An integer constant is a sequence of digits in decimal representation. Integer representations using other bases are not supported in DCL. Integer constants following a negation operator - are negative integers.

Examples:

```
30      /* decimal number 30 */
-30    /*  decimal number -30 */
```

Double Constant

A double constant represents a floating-point number and consists of integer part, decimal point, fraction part, and exponent part. Both integer part and fraction part are sequences of numbers. Exponent part includes e and the exponent. In DCL, either

integer part or fraction part can be missing from a double constant, but they cannot be missing at the same time; either decimal point or the exponent part can be missing, but they cannot be missing together. Our definition is equivalent to that of the C language, which defines them as:

> "A floating constant consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part, or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing."

Examples:

```
30.5          /*  floating-point number 30.5 */
-30.0         /*  floating-point number -30.0 */
3.253         /*  floating-point number 3.253 */
3253e-3       /*  floating-point number 3.253*/
3253          /*  floating point number 3253*/
.5            /*  floating point still valid even if integer part is missing */
5.            /*  Allowed even if the fraction part is missing */
```

String Constant

String constants are sequences of characters enclosed in double quotes. If double quotes need to be used in string constants, they need to be prefixed by a backslash.

```
"a"                 /* string a */
"ab"                /* string ab */
"abc\"de"           /* string abc"de */
"Hello! World"      /* string Hello! World */
```

Types

Integer (int)

Integers in DCL are stored in 32 bits (4 bytes). Integers should be used in this language when working with whole numbers (i.e., countable quantities). Integers can be any values between -2,147,483,648 and 2,147,483,647.
An example of working with integers in the correct DCL syntax is as follows:

```
int <function_name>(int <parameters> ) {
        <code placed here>
```

```
        return <int value>;
}
```

One-line example
int x = 1;

Double (double)

Doubles are 64 bits (8 bytes). Doubles should be used to store fractional numbers or numbers too large to be saved into the integer primitive data type. All values will be represented in binary so doubles may be approximated. They can range in value from 1e-37 to 1e37.

An example of working with doubles in the correct DCL syntax is as follows:

```
double <function_name>(double <parameters> ) {
        <code placed here>
          return <double value>;
}
```

One-line example:
double x = 1.0;

void

In DCL, the void type is for when a function returns nothing or an empty value after being called. In this language, a void language will return null.
An example of working with the void type in the correct DCL syntax is as follows:

```
void <function_name>( <parameters> ) {
        <code placed here>
}
```

Arrays

In DCL, arrays are a data structure that allows a programmer to store groupings of one or more elements together in memory. For indexing an array, it begins at value 0 rather than 1. In addition, arrays must group items of the same type. As such, if a programmer tries to make a list of with different primitive data types, it will cause an error in the code. Arrays of types void are not allowed and unchecked in DCL.

To create an array, you need to specify the primitive data type for and the name of the array.

Sample line for creating an empty array:
double[] purple;

Sample line for setting variables for an array:
purple = [3.0,4.0,5.0];

Casting

There is no syntax for casting in DCL at this time. This means that mixing types will cause errors because it has not been supported in the current version of DCL. However, to print a value, the user simply needs to call print if they wish to print without a newline character and print_line to print a statement with a newline character at the end. These print statements will print values of the same type and will return an error if there are multiple types included in the print line.

To print, the user can call print or print_line and only supply one primitive data type into the function.

Expressions and Operators

Expressions

Below is a list of expressions in DCL.

- A literal value
- Two operands separated by an operator
- Accessing an array
- Expression between ()

For arithmetic expressions, they require at least one operand and zero or more operators. Below are some examples of arithmetic expressions.
10; /* becomes an integer of value 10 (int 10) */
1700 + 89; /* becomes an integer of value 1789 (int 1789) */
1800.0 - 20.0 /* returns a double of 1780.0 (double 1780.0) */
100.0/20.0 /* returns a double 5.0 (double 5.0) */

In DCL, you can also use parentheses when trying to group multiple operands:

```
(10 * (2+2)  - (4*4) )  /* expression evaluates to integer 24 */
```

Function Calls to Expressions

An expression is any call to a function that returns a value.

Example:

```
print_line("hello"); /* print_line is a void function and will return nothing*/
```

Array Access as Expressions

When a programmer creates an array in DCL, that array returns the type based on the primitive data type passed to it. When indexed, it returns the appropriate value.

Example:

```
double[] myArray = [1.0, 2.0];
myArray{|0|};  /* evaluates to 1.0 */
```

An operator is an operation that is applied to operands. Depending on the operation, the operator may need one or two operands.

Assignment Operators

Assignment operators store values into variables. In DCL, there are various ways to use the assignment operator. For DCL's syntax assignment is done through the "=" operator. The rule for it is that the value on the right side of the operand is saved to the operand on the left. DCL does not support the left operand to be a literal or constant value.

Example:

```
int celie = 20;
double nettie = 16.5;
string plt = "We should have written this in Swift instead of OCaml";
int PierreAndNatasha = 1800 + 12;
1 = 6 /* invalid */
"groundhogDay" = "greatCometof1812" /* invalid */
```

Arithmetic Operators

DCL offers the standard arithmetic operations featured in most modern programming languages: addition, subtraction , multiplication, division, and negation.

```
/* Addition */
int x = 1700 + 76;
```

For variables of type string, they also can be concatenated via the addition operator. An example of this is the following:

```
string hello = “hello ”;
string world = “world”;
print_line(hello + world);

/* Subtraction */
double y = 33.0 - 19.0;

/* Multiplication */
int z = 6 * 6;

/* Division */
double w = 100/24

/* Negation */
int f = -10;
```

Type designation for mixed types occurs from left to right

Conditional Operators

In DCL, you can use the conditional operators to determine the relationship between two operands. This includes checking if they’re equal and if one is greater or less than the other. For booleans, users should just use 0 for false and 1 if true.

```
int x = 10;
int y = 3;
int test = (x==y) /* evaluates to 0 (false) */
int w = 4;
int q = 4;
int equal = (w==q) /*evaluates to 1 (true) */
```

It is important to note that in DCL, comparing float values will yield unexpected results because of the underlying LLVM implementation. In other cases, users can use the greater than, greater than or equal to, less than, and less than or equal to.

```
string ghana = “1957”;
string random = “1957”;
int compared = (ghana == random) /* evaluates to true */
int x = 4;
int y = 3;
int z = x > 3 /* evaluates to 1 true */
```

The ==, !=, >=, >, <, <= operators are all defined to operate between any two values both either being of int or double. The The ==, != are also designated to compare any two given values in DCL, and if they are not both of type double, it will only return true if the memory address is identical.
In DCL, you can use and (&&), or (||) , and not (!)).

Length Operator

DCL supports getting the length of both strings and arrays. This can be done by using the hashtag (#) operator. Calling this operator returns an integer corresponding to the size of the string or array. For strings, this is the number of characters in the string. For arrays, this is the number of elements in the array. An example of using the length operator can be found below:

```
string bodega = “store”;
int bodega_length = #bodega;
print_line(bodega_length); /* should print 5 */
```

Tilde Operator

DCL has a special tilde operator denoted with “~”. Tildes only work within buteverytime blocks, which define a global callback as defined in the Callbacks section. (In essence, the callback [the specified block of code] will be executed every time an iteration occurs.) In short, the tilde operator allows the callback to access the old value of the variable. Here’s an instructive example:

```
int i = 0 buteverytime (1) {
        print(~i);
        print(“ (has been updated to) ”);
        print(i);
};
```

After every line, one can see the old value and new value of i. Tildes are supported on variables of type string, array, int, and double. They are not supported on the void type. For arrays, they can be used when checking a specific value in an array. Their functionality is not defined on any other type not specified above.

Operator Precedence

When a given expression contains multiple operators, the operators are grouped based on the rules of precedence. Below is the list from highest precedence to lowest.

- Function calls, array subscripting, and membership access operator expressions.
- Unary operators, including logical negation and unary negative.
- When there are several unary operators that are in consecutive order, the later ones are nested within the earlier operators. For instance, not-w translates not(-w)
- Multiplication and division
- Addition and subtraction
- Greater than, less than, greater than or equal to, less than or equal to
- Expressions
- Equal and not equal
- AND expressions
- OR expressions
- All assignment expressions

Statements

A statement is the basic level of code hierarchy; this will become important later when defining the concept of callbacks.

Expression Statements

An expression statement is just an expression with a semicolon, and the expression is evaluated when the line of code is run. Examples are shown below:

```
/* Assignment Expression */
i  = x + y;

/* Function Evaluation */
sayHello(“Don”);
```

Declaration Statements

A declaration statement creates a new variable; you can choose to either provide the initial value or take the initial default value by providing no value.

```
/* Provided Value */
int i = 13;

/* Default Value */
int i;
```

Control Flow Statements

DCL traditionally executes statements one after the other; control flow statements extend this functionality by adding the possibility for certain blocks of code to either be conditionally executed once or executed many times.

If/else statements

If/else statements allow a block of code to be executed or not depending on whether the value of the condition evaluates to true. The structure is the condition, the block of code to be executed when the condition is 1 (true), and optionally the block of code to be executed when the condition is 0 (false).

```
/* Succeeding if condition */

if (1) {
        print_line("Succeeds!"); /* "Succeeds" will be written to standard output */
}

/* Failing if condition */

if (0) {
        print_line("Fails!"); /* "Fails" will not be written to standard output */
}

/* Conditional if/else execution */

if (condition) {
        print_line("Condition was truthful");
} else {
```

```
        print_line("Condition was false");
}
```

In the last example, one of (but not both!) of the blocks of code will be run; furthermore, the convention is that an otherwise clause gets assigned to the last unmatched if/else statement.

for and while Statements

Both the for and while keywords exist to facilitate blocks of code to be executed multiple times. The key difference is that for provides access to an index of the iteration whereas while does not.

```
/* for example */
for(int i = 0; i < 10; i += 1) {
        print_line(i);
}
```

The initialization (here it is: int i = 0) takes place once before the loop block has executed; the termination condition is evaluated every single time before the loop block executes. Should the termination condition (here it is: i < 10) be true, the loop will be terminated immediately. The loop block (here it is: print(i);) executes after the termination condition is checked, and finally, the update (here it is: i += 1) is then executed after the block is executed.

While statements are close to for statements; essentially, one only keeps the termination condition and loop block. Consequently, all setup should occur before the while statement, and updates should occur within the loop block.

```
/* while example */
int make_me_bigger = 3;
while (make_me_bigger <= 10) {
        make_me_bigger = make_me_bigger * 2;
}
```

Callbacks

The dynamic callbacks are the special and relatively unique part of DCL. Essentially, it allows variables to have global sets of instructions that need to be executed depending

on the state of themselves and other variables; these are introduced by the buteverytime keyword. An example should prove useful.

```
int i = 0 buteverytime (i == 0) {
        print_line("i can't be zero, changing!");
        i = 1;
}
```

This declaration statement defines a new variable i and also attaches a dynamic callback to i. In essence, whenever i is used later in the program, the block of code with the print and assignment statements will be executed if the value of i is 0. In this example, the value of i will change immediately after the declaration statement finishes because the condition for that clause is met.

More specifically, after each statement is executed, all callbacks will be checked and those satisfying their conditions will be executed. (Callbacks cannot be nested within each other.) This results in a powerful construct as demonstrated by the next example. To do this, DCL automatically creates functions for the functions defined in a callback prepending them with "__." Normal functions are barred from being named with "__" at the front to make this possible.

```
int i = 0 buteverytime (i < 10) {
        print_line(i);
        i = i + 1;
}
```

This set of code will print the integers from 0 to 9 separated by newline.

The buteverytime construct also introduces a new operator: the tilde (~). The ~ helps with making more powerful buteverytime code chunks: it lets the source code track changes to variables; essentially, the ~ operator gives the value of the variable before the statement was executed. Here's a slight twist on the second example.

```
int i = 0 buteverytime (i != ~i) {
        print_line(i);
        if (i < 10) {
                i = i + 1;
        }
}
```

It is important to note that the initial value of a variable before it is defined is equal to the default value for that type but is not equal to any other value. This reformulated version achieves the exact same functionality as the second example.

Code Examples

/* Hello World in DCL */

```
int main() {
print_line(“Hello world!”);
return 0;
}
```

/* For loop in DCL */

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

/* Function definition in DCL */

```
double div(double x, double y) {
        if (y == 0) {
                print_line(“denominator cannot be 0”);
                return x;
        } else {
                return x / y;
        }
}
```

/* Variable a is initialized to be 5, but every time the value of a changes to 0, a warning message is printed out because we don’t want variable a to be 0. */

```
int a = 5 buteverytime (a == 0) {
            print_line(“a is 0 which is an illegal value for a”);
}
```

/* Variable i is initialized to be 0, but every time when the value of i changes and is not equal to the original value assigned, the new value of i is printed out. */

```
int  i = 0 buteverytime (i != ~i) {
```

```
        print_line(i);
}
```

/* a is initialized to a list with 6 items, but every time when the value of a[2] changes, a message is printed out to deliver that information; also, when the length of a is changed and is not equal to the original length, a message is printed out as well */

```
int[] a = [1, 7, 10, 5, 9, 8]
buteverytime (a[2] != ~a[2]) {
        print_line("the value of a[2] is changed");
}
buteverytime (len(a) != ~len(a)) {
        print_line("the length of list a is changed");
}
```

/* This for loop prints values 0 to 29 and 81 to 99 because we are using butfor to skip a range of numbers that we don't want our for loop to iterate through. */

```
for (int i = 0 buteverytime(i == 30) {i = i + 50;}; i < 100; i++) {
        print_line(i);
}
```

/* This function prints out the message "Hi Stephen Edwards" only once because we are using buteverytime to actually change the increment variable x. */

```
int x = 0 buteverytime (x < 10) {
        print_line("Hi Stephen Edwards");
        x = x + 10;
 }
void printDrake() {

       while (x < 20) {
            x = x+1;
       }
}
```

Linear Regression written in DCL:

/* This function utilizes gradient descent to find optimal parameter values for the linear regression problem with the given data. The usage of buteverytime is noteworthy here

because it allows for variable changes to be coupled together; explicitly, both slope and intercept change at the same time. The authors feel that this structuring of the code makes more sense because all logic relating to particular variables is mostly near the variable definition, which adds clarity. */

```
double dB1(double[] x, double[] y, double currentB0, double currentB1) {
double dB1 = 0;
for(int i = 0; i < len(x); i++) {
dB1 = dB1- 2 * (y{| i |} - currentB1 * x{| i |} - currentB0) * x{| i |};
        }
return dB1;
}
double dB0(double[] x, double[] y, double currentB0, double currentB1) {
double dB0 = 0;
for(int i = 0; i < len(x); i++) {
dB0 = dB0 - 2 * (y{| i |} - currentB1 * x{| i |}- currentB0);
        }
return dB0;
}
double cost(double[] x, double[] y, double currentB0, double currentB1) {
double cost = 0;
for(int i = 0; i < #x; i++) {
cost = cost + (y{| i |} - currentB1 * x{| i |}- currentB0) ^ 2;
        }
return cost;
}

double[] x = [1, 6, 3, 8];
double[] y = [2, 8, 5, 10];

double intercept = 0 buteverytime (~cost != cost) {
intercept = intercept - dB0(x, y, intercept, slope) * alpha
}
double slope = 1 buteverytime (~cost != cost) {
slope = slope - dB1(x, y, intercept, slope) * alpha
}
double alpha = 0.5;
double cost = 0.0 buteverytime (cost == 0 || (~cost - cost) / cost < 0.001)
```

```
) < 0.001) {
stop = 1;
}
void main() {
int stop = 0;
while(!stop) {
cost = cost(x, y, intercept, slope);
}

print_line(“Slope: ” + slope);
print_line(“Intercept: ” + intercept);
}
```

Bee Movie Example:
This program takes in the bee movie script and replaces every instance of the word “bee” (or Bee) and replaces it with " b + + “:

```
/* Bee Movie but every time they say "bee"... */

string[] split(string whole, string sep) {
        int number_of_parts = 0;
    int looking_for_parts = 0;
    int index;
    for(index = 0; index < #whole; index = index + 1) {
        if(whole[ index ] == sep) {
            if(looking_for_parts) {
                looking_for_parts = 0;
            }
        } else {
            if(!looking_for_parts) {
                number_of_parts = number_of_parts + 1;
                looking_for_parts = 1;
            }
        }
    }
    string[] parts = [number_of_parts of ""];
    int current_index = 0;
```

```
    looking_for_parts = 0;
    string current = "";
    for(index = 0; index < #whole; index = index + 1) {
        if(whole{| index |} == sep) {
            if(looking_for_parts) {
                looking_for_parts = 0;
                parts[ current_index ] = current;
                current_index = current_index + 1;
                current = "";
            }
        } else {
            if(!looking_for_parts) {
                looking_for_parts = 1;
            }
            current = current + whole{| index |};
        }
    }
    parts[ current_index ] = current;
    return parts;
}

string join(string[] parts, string sep) {
          string total = "";
    for(int part = 0; part < #parts; part = part + 1) {
        total = total + parts{| part |} + sep;
    }
    return total;
}

int starts_with(string haystack) {
    return (haystack{| 0 |} == 'b' || haystack{| 0 |} == 'B') &&
           haystack{| 1 |} == 'e' && haystack{| 2 |} == 'e';
}

string current_word = "" buteverytime (starts_with(current_word)) {
    current_word = "     b + +     ";
}
```

```
void main() {
    string bee_movie_script = read("bee_movie_script.txt");

    string[] bee_movie_words = split(bee_movie_script, " ");
    string[] modified_bee_movie_words = [#bee_movie_words of ""];

    for(int i = 0; i < #bee_movie_words; i = i + 1) {
        current_word = bee_movie_words{|i|};
        modified_bee_movie_words[i] = current_word;
    }

    write("b++_movie_script.txt", join(modified_bee_movie_words, " "));
}
```

Language Grammar

We have included the full grammar of a DCL program below:

```
%{
open Ast
%}

/* Ocamlyacc parser for DCL */
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE EXPONT ASSIGN NOT TILDE
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR DOUBLE STRING
BUTEVERYTIME
%token RETURN IF ELSE FOR WHILE INT BOOL VOID LINDEX RINDEX
%token LSQUARE RSQUARE OF LENGTH
%token <int> INTLITERAL
%token <float> DBLLITERAL
%token <string> STRLITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
```

```
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right EXPONT
%right NOT NEG LENGTH
%left LINDEX

%start program
%type <Ast.program> program

%%


program:
  decls EOF { $1 }

decls:
   /* nothing */ { [], [] }
 | decls globalstmt { ($2 :: fst $1), snd $1 }
 | decls bdecl { fst $2 :: fst $1, (snd $2 :: snd $1) }
 | decls fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    { { typ = $1;
          fname = $2;
          formals = $4;
          body = List.rev $7 } }

bdecl:
   typ ID ASSIGN expr BUTEVERYTIME LPAREN expr RPAREN LBRACE stmt_list
RBRACE
    { (GlobalAssign($1, $2, $4), { typ = $1;
   fname = "__" ^ $2;
   formals = [($1, $2) ; ($1, "~" ^ $2)];
   body = let full_stmt_list = (List.rev $10) @ [ Return (Id($2)) ] in
```

```
        [ If($7, Block(full_stmt_list), Return (Id($2))) ]  (*(Id($2))*)
      })
    }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    typ ID                   { [($1,$2)] }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }

dtyp:
    INT { Int }
  | DOUBLE { Double }
  | STRING { String }

dim_list:
    LSQUARE RSQUARE  { 1 }
  | LSQUARE RSQUARE dim_list { 1 + $3 }

atyp:
    dtyp dim_list { Array($1, $2) }

typ:
    dtyp { Simple($1) }
  | VOID { Void }
  | atyp { $1 }

globalstmt_list:
    /* nothing */    { [] }
  | globalstmt_list globalstmt { $2 :: $1 }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }
```

```
stmt:
    expr SEMI { Expr $1 }
  | RETURN SEMI { Return Noexpr }
  | RETURN expr SEMI { Return $2 }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | typ ID SEMI {Local($1, $2)}

globalstmt:
    typ ID SEMI { Global($1, $2) }
  | typ ID ASSIGN expr SEMI { GlobalAssign($1, $2, $4) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

index:
    LINDEX expr RINDEX { $2 }

val_list:
    expr               { [ $1 ] }
  | expr COMMA val_list { [ $1 ] @ $3 }

simple_arr_literal:
    LSQUARE val_list RSQUARE { $2 }

expr:
    INTLITERAL       { IntLiteral($1) }
  | DBLLITERAL       { DblLiteral($1) }
  | STRLITERAL       { StrLiteral($1) }
  | simple_arr_literal { ArrLiteral($1) }
  | TILDE ID       { TildeOp($2) }
  | ID               { Id($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
```

```
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EXPONT expr { Binop($1, Exp,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr         { Unop(Not, $2) }
  | LENGTH expr      { Unop(Length, $2) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LSQUARE expr OF expr RSQUARE { DefaultArrLiteral($2, $4) }
  | ID LSQUARE expr RSQUARE ASSIGN expr { ArrayAssign($1, [$3], $6) }
  | expr index { Index($1, [$2]) }
 /* | ID index ASSIGN expr  { Assign(Index(Id($1), $2), $4) } */
  | LPAREN expr RPAREN { $2 }
  | typ ID ASSIGN expr {LocalAssign($1, $2, $4)}

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

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

Standard Utilities

To help users make useful tools through DCL, we have built out a standard utilities to support users in doing more in our language.

read

read allows a user to read in a file into a string. The function will return a string of the entire file or will quit if there is an issue opening the file.

/* open a file and have result returned as a string */
string beemovie = read("beemovie.txt");

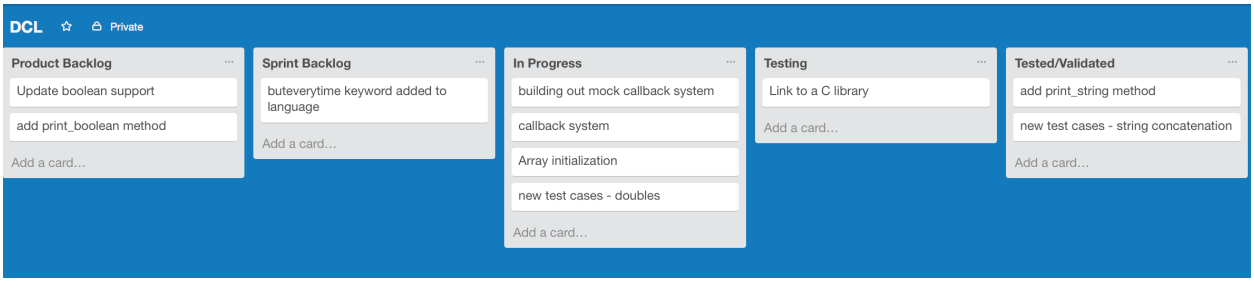
write

write allows the user to write to a file. To use this function, the user must provide a string containing the text they want to output as well as the name of the file to output to.
/* writes the string beemovie to beemovie.txt */
string beemovie = "According to all known laws of aviation….";
write(beemovie, "beemovie.txt");

Chapter 4: Project Plan

Project Planning

In order to complete this project on time, our team followed the Agile methodology through the Scrum process. This meant that we worked iteratively and made sure to keep a master branch that could be run at anytime. This meant each of our iterations was shippable, deployable code--albeit not the full end product we wanted. We also had two members of our team meet with Professor Edwards to discuss our callback system and the best ways to implement it. Below, we have included an example of what our Trello Board for dealing with product to-dos looked like.



Specification Process

When we first started this project, we knew we wanted the language to be cross-platform. Since members of our group use different computers and different operating systems, this was a main goal because then we could all ultimately use this language. This meant that we needed to write the language such that it compiled down to LLVM IR since this was the easiest way to achieve this overarching goal.

After deciding on this standard, we began brainstorming what we wanted to make this language centered around. Since many of us did not rely on object-oriented programming for most of our general uses, we settled on a scripting language that offered a new system offered callbacks. With callbacks, we figured we would be able to make a base language that was quite useful as well as a new format that required using less if statements and was even easier to read. Once we decided on this, we were quickly able to come up with our name: Dynamic Callback Language (DCL).

Development Process

To ensure on-time delivery of this project, we stuck to the main course deadlines and advice from our TA. We began by working on the scanner and parser to help with planning our LRM. Afterwards, we began to work on our semant file followed by the AST file and finally codegen. As feature plans changed as well as specifications, we went back through the old files and updated them as necessarily to be in line with our language. In addition, for some features we leveraged pair programming. This was immensely helpful to avoid people getting stuck and to speed up the pace at which we implemented features.

Testing Process

Creating DCL required a lot of testing and documenting. When we began adding new features, we focused on writing out sample programs that we could later use to test if the new feature was working correctly.

When it came to debugging code, we would try to log how we tried to debug a problem either in a shared note or by writing comments in the code we shared. This was useful for speeding up debugging later on because we could look back at past problems other team members faced and how they resolved them.

Team Responsibilities

Every member of the team played an active role in creating this language. To effectively accomplish this, we each took on more than just our assigned titles. The table below shows what each member focused on, but in the end the entire team mostly worked on all parts of this project.

Team Member	Responsibilities
Ashutosh Nanda	Doubles, Arrays, Codegen, Parser
Chang Liu	Test Suite, Strings, Parser, Codegen, Semant
Craig Rhodes	Callbacks, Circle CI, Docker, AST, LRM
William Essilfie	Scanner, Parser, Test Suite, Semant, Final Report

GitHub

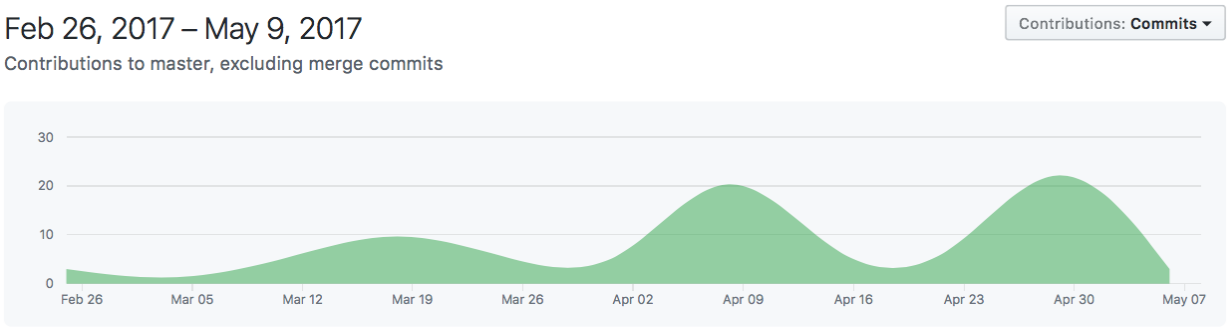
Due to the scope of this project and the need to keep track of updates to the language, we used Git version control system on GitHub. We worked on the language from the following Github accounts:

- Ashutosh Nanda - ashutoshnanda
- Chang Liu - cl3403
- Craig Rhodes - CraigRhodes
- William Essilfie - wessilfie

Project Log

Thanks to the nifty analytics tools provided by GitHub, we were able to track our commit history across the entire project.

Looking at this, we can see our commits rise quickly as we near the end of a sprint and as we started adding our bigger language features near the end of the semester. In addition, every member of the team played an active role in development throughout all the stages.



Git Commit History

commit 953acb7f4a7cce0c80045bf714c6afaac16ffa65
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 19:29:06 2017 -0400

    Updated Makefile to not clean everytime

commit 6ef0cf8cd530c892a5a7f6cdbd799534b8314c80
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Wed May 10 18:37:22 2017 -0400

    Change name of exponentiation functions

commit e4510b2164b010f89071770ab3ce7dbb2a76291c
Merge: 49d585c d54b47f
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 18:25:19 2017 -0400

    Merge branch 'master' of https://github.com/PLT-DCL/dcl

commit 49d585cb1390e2fb9383da01bf499ccaadd2c90b
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 18:25:16 2017 -0400

    Fixed tests

commit d54b47f82fdc08e08ca02786dd9f4e052ae31643

Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Wed May 10 18:14:08 2017 -0400

    Cleaning up master

commit 42b229eab5690f00c725407d1e5297b9f0ac1d72
Merge: cedcc80 5432298
Author: Chang Liu <cl3403@columbia.edu>
Date: Wed May 10 17:20:17 2017 -0400

    Merge branch 'master' of https://github.com/PLT-DCL/dcl

commit cedcc80a1fc2910835325421984b7545901cef4c
Merge: 6827587 7e49764
Author: Chang Liu <cl3403@columbia.edu>
Date: Wed May 10 17:19:22 2017 -0400

    combining all test cases

commit 543229828ff882c43010e12c81c203e75c559b40
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 17:17:18 2017 -0400

    Modified CI files

commit c1c3c377b95e95be7f0890de2a63b9e38f938f3b
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 17:13:16 2017 -0400

    Modified CI files

commit b815e35703e0f41967d60ad885d20b8c843c36df
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 17:05:00 2017 -0400

    Modified CI files

commit 527da1916fef7f0a09facdd91aa5a5dc4890dd3e

Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 16:56:59 2017 -0400

    Modified CI files

commit 7e49764e028b0ed91c77f0682a5c3c539f0896ef
Merge: 5cb1c0d 666c24e
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 16:52:44 2017 -0400

    Resolved merge conflicts from arrays-and-callbacks

commit 666c24ee7d3a923bcb160e3aa14fb39a8166dce8
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Wed May 10 16:48:16 2017 -0400

    Figured out why multi-statement function can't be called in buteverytime condition

    Basically, you would end up triggering the buteverytime condition on every line of the function, so you'd eventually segfault.. It's a feature!

commit 682758737267ae3979bde8cc5a86e722bd69c93e
Author: Chang Liu <cl3403@columbia.edu>
Date: Wed May 10 16:40:55 2017 -0400

    adding callbacks test cases

commit 50c4bcb80d9612710a37aeca35e18d4c4b83e141
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Wed May 10 16:32:37 2017 -0400

    Fixed tilde bug

commit 5cb1c0d555454672e5b2206222e0b5aa31b08b15
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 14:50:46 2017 -0400

    Updated circle.yml

commit 3308960c418898ebf79a66933a2d34c5f926e362
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 14:47:26 2017 -0400

    Updated circle.yml

commit a85bfc4e1089b6e45014bf95d09e049570977bd0
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 14:46:18 2017 -0400

    Updated circle.yml

commit f89a9d6a1b5faa1767ae62741e79b1c052de7b58
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 14:43:11 2017 -0400

    Updated circle.yml

commit 72554d7026a896567925f60eccda1b3cd22dd073
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Wed May 10 14:39:09 2017 -0400

    DEMO

commit e8c8a8fdb1048d37d597665c7afdfe04f4889048
Merge: 9f2c656 39185ee
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 14:30:24 2017 -0400

    Merge branch 'arrays-and-callbacks' of https://github.com/PLT-DCL/dcl into arrays-and-callbacks

commit 9f2c6560700640019ca085edb7f5edf6f7d1467d
Merge: a2a0b94 5225d56
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 14:28:59 2017 -0400

    Merge branch 'arrays-and-callbacks' of https://github.com/PLT-DCL/dcl into arrays-and-callbacks

commit 5225d5632f9d9fd90685e25cab3311a45307c395
Author: Chang Liu <cl3403@columbia.edu>
Date: Wed May 10 14:17:23 2017 -0400

    fixing all test cases

commit 39185ee196c1c21b48b34296d1237effee543d37
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Wed May 10 13:26:38 2017 -0400

    Fixed the way value is passed into callback

commit a2a0b94003d02070496a84797c266bf180bcb0ca
Merge: cb79e9e 4ddf282
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 13:02:31 2017 -0400

    Merge branch 'arrays-and-callbacks' of https://github.com/PLT-DCL/dcl into arrays-and-callbacks

commit 4ddf2825c53f432ef75d4f8bd84145ed08547b22
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Wed May 10 13:01:44 2017 -0400

    Global assignment works

commit cb79e9e907139627bf512ea408310036e9545dd6
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Wed May 10 11:02:35 2017 -0400

    Updated codegen.ml with tilde operator currently set to track the value of the variable it's mirroring

commit 9161402ece0e4700dd28f8d0b0a02073db1b38b4
Author: Craig Rhodes <cdr2139@columbia.edu>

```
Date:   Wed May 10 02:55:05 2017 -0400

    Fixed an edge case of the read function

commit 56977da2a9a87057203bfd74a7e4fe0e8d2943d3
Merge: 1b68ba8 ab886cb
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Wed May 10 02:27:25 2017 -0400

    Merged in new code relevant to tilde

commit 1b68ba8b2b8a1020a0804a22f8bc9308a18e5e2f
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Wed May 10 02:24:14 2017 -0400

    Added tilde operator

commit ab886cbe8b4f43555c87d34a80d2ffc05113fa31
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date:   Wed May 10 01:53:19 2017 -0400

    File I/O changes

commit cc348e8157911131764e46befde87fdfbd4a5de6
Author: Chang Liu <cl3403@columbia.edu>
Date:   Tue May 9 22:25:32 2017 -0400

    adjusted the error msg

commit 44566c83889a3ec2c52d4bc8798129135c125853
Author: Chang Liu <cl3403@columbia.edu>
Date:   Tue May 9 22:25:05 2017 -0400

    fixed a parsing error

commit 9488860161c9e6fd2794d0fa35208770aa9ad111
Author: Chang Liu <cl3403@columbia.edu>
Date:   Tue May 9 22:19:49 2017 -0400
```

```
    adjusted the test cases

commit d0156a590aafecf41b35835151cc538cf58e7933
Author: Chang Liu <cl3403@columbia.edu>
Date:   Tue May 9 22:19:02 2017 -0400

    fixed the error msg

commit ade8fae68a16d7d103d039751d5b9337d48a0f96
Author: Chang Liu <cl3403@columbia.edu>
Date:   Tue May 9 22:18:47 2017 -0400

    fixed the error msg

commit f4984c2b07148b8968993631fb442083dff13d9c
Author: Chang Liu <cl3403@columbia.edu>
Date:   Tue May 9 22:17:19 2017 -0400

    fixed duplicate formals bug

commit f7714cf0e6be07939f2719bf7c75be1431e71beb
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Tue May 9 20:37:35 2017 -0400

    Updated Makefile to correctly link to object file again

commit f224026b3d264caecec84f75c5c94a1e4d22066a
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Tue May 9 19:23:39 2017 -0400

    Updated Makefile to be less verbose

commit a510247776657cd7a8cc168ae1ffc5767fe33d76
Merge: 3b05fda 25c3938
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Tue May 9 18:56:16 2017 -0400
```

    Merge branch 'arrays-and-callbacks' of https://github.com/PLT-DCL/dcl into arrays-and-callbacks

commit 3b05fda4b70ca40f699048d165819792f4a99b02
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Tue May 9 18:55:38 2017 -0400

    Updated tests with new print_line function

commit 25c3938dff0f2191c6c4e0cd3543e666a13677d3
Author: Chang Liu <cl3403@columbia.edu>
Date: Tue May 9 18:43:53 2017 -0400

    make global strings work

commit 1d792f99dc7fb0ff8e9faade2efc7cd4e60226e7
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Tue May 9 17:41:20 2017 -0400

    Updated strcmp function to not try to use addstr

commit 9ca06546e54e343ee028b7ed4451ff695a1a0446
Merge: 25ce4c1 c902652
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sun May 7 21:07:10 2017 -0400

    Merged arrays into master

commit 25ce4c19f80461451ce12c26d40dcf9f7ada65a9
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sun May 7 18:54:44 2017 -0400

    Made callbacks work for all callbacks and not just a single one

commit f1004d879dbe5c6cf72f7782f7954339ebe1db93
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sun May 7 16:37:55 2017 -0400

Callbacks now work again

commit c90265293ae94a4df2cacf701cb0555076a36e84
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Sun May 7 15:33:03 2017 -0400

Full Array + String Functionality

commit 8d9331f6fc63272f8a5a85f38673b9f390edc143
Author: Chang Liu <cl3403@columbia.edu>
Date: Sun May 7 14:48:46 2017 -0400

using llvm build and store again

commit 2681ba4e0c65bfc1705c3d17350228e024a61e5e
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sat May 6 22:08:28 2017 -0400

Passing in an actual as an argument to callbacks, retrieved from hastable in codegen

commit 211ce5b14c022425723994e2fb1489195fbec59c
Merge: e410a37 9c384e5
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sat May 6 19:59:52 2017 -0400

resolved merge conflicts with upstream

commit e410a37748472fcd8e58f9f0e69d62aa889d12ea
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sat May 6 19:58:40 2017 -0400

Updated buteverytime statements to take in a formal for the locally defined value defining the declaration

commit 9c384e5c3d644552b04e4db66ea1fba0d8e0c8d5
Author: Chang Liu <cl3403@columbia.edu>
Date: Sat May 6 19:57:29 2017 -0400

    now global assignment happens globally

commit 45c8b7779f3ea8be7480509d8934facb782d028f
Author: Chang Liu <cl3403@columbia.edu>
Date: Sat May 6 01:56:03 2017 -0400

    hopefully please fix the callback issue

commit 50f0be9f644e7cddf7493ec792019d08b93fac3b
Merge: 65e426b dccdcf9
Author: Chang Liu <cl3403@columbia.edu>
Date: Sat May 6 01:55:05 2017 -0400

    Merge branch 'standard-library' of https://github.com/PLT-DCL/dcl into standard-library

commit 65e426bc81c7976780fee0ca77b038a275b9c863
Author: Chang Liu <cl3403@columbia.edu>
Date: Sat May 6 01:53:34 2017 -0400

    hopefully fixing the callback bug

commit dccdcf9bd2f2975d593eccb883829f120abfc46f
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sat May 6 01:37:52 2017 -0400

    Added test dcl file for callbacks

commit 028ca905c05fc0f5dd0965e06fa4fc424747e193
Merge: 4719943 d8c0d31
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sat May 6 01:36:23 2017 -0400

    Merge branch 'standard-library' of https://github.com/PLT-DCL/dcl into standard-library

commit d8c0d319aa2f76d2c57f731a09a289b848778f31
Author: Chang Liu <cl3403@columbia.edu>

```
Date:   Sat May 6 01:36:08 2017 -0400

    potentially solving callback bug

commit 4719943035f581f911c5b80268fa4c254d18be28
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Sat May 6 01:22:32 2017 -0400

    Modified parser to use GlobalAssign for bdecl variable

commit eac152a83c8915d124342a5543312830cf02770e
Merge: bb9bce3 e3c1acb
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Sat May 6 00:10:26 2017 -0400

    Merged in recent changes for standard-library

commit e3c1acbc21cf21dabbe46f7964379ab89618b56d
Author: Chang Liu <cl3403@columbia.edu>
Date:   Fri May 5 23:45:54 2017 -0400

    fixed a bug with checking global duplicates

commit 7eed8102f5928afbb3b1910f1a37e4b44abc7316
Author: Chang Liu <cl3403@columbia.edu>
Date:   Fri May 5 23:00:58 2017 -0400

    now semant checks for local and global duplicates

commit bb9bce395093cb28794e91a280bc36c82f5a3a4d
Author: Chang Liu <cl3403@columbia.edu>
Date:   Fri May 5 03:21:20 2017 -0400

    now global variable assignment works

commit b2d9e1a528d1345dc1a4f4d612a914159fda9ce8
Author: Chang Liu <cl3403@columbia.edu>
Date:   Thu May 4 23:10:28 2017 -0400
```

fixed bugs for this test file

commit f49de5685de7698bf228ff3a776f473a7bc77502
Author: Chang Liu <cl3403@columbia.edu>
Date: Thu May 4 22:28:25 2017 -0400

debuging Makefile and test files

commit d8a5a30ae8dbb9a2742518e3a2fb1a7121896a5b
Author: Will Essilfie <wke2102@columbia.edu>
Date: Thu May 4 21:49:42 2017 -0400

working on fixing testing

commit eb629ac11dd7f7f1af0c4ee74d79a33d5dd49802
Merge: f69b630 fb01de9
Author: Craig D. Rhodes III <cdr2139@columbia.edu>
Date: Fri May 5 11:56:45 2017 -0400

Merge pull request #14 from PLT-DCL/callbacks

Callbacks

commit fb01de9ae78b522ae1bcae2c1b958aa9ad527104
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Fri May 5 03:50:58 2017 -0400

Callbacks are now generated between every function call and are not generated inside callbacks

commit 0563f7a0aac7b3eb21aa81fa40041aae42b9cbaa
Author: Chang Liu <cl3403@columbia.edu>
Date: Fri May 5 03:21:20 2017 -0400

now global variable assignment works

commit 4b01b15ec180ca9c3ed369abb1e275c35b5513c9

Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Fri May 5 00:16:36 2017 -0400

    Updated bdecl to declare buteverytime variable globally

commit 2dabc2f87d16b5e49858bb774821e3b60864d6b0
Author: Chang Liu <cl3403@columbia.edu>
Date: Thu May 4 23:10:28 2017 -0400

    fixed bugs for this test file

commit 77d80046206fcc2b7914fbd9c5b8f1d54a770154
Author: Chang Liu <cl3403@columbia.edu>
Date: Thu May 4 22:28:25 2017 -0400

    debuging Makefile and test files

commit bcf0719fdaa7ef7652c92ec871bdebb94be3bb79
Author: Will Essilfie <wke2102@columbia.edu>
Date: Thu May 4 21:49:42 2017 -0400

    working on fixing testing

commit 2e59796ff00f20f1c0259af5355783942f002f9f
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Thu May 4 21:08:38 2017 -0400

    Updated parser to store bdecls properly

commit 0eb3379ac0686b3d20ff53f96984d112e56998e8
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Thu May 4 20:57:14 2017 -0400

    Updated parser to recognize buteverytime again

commit f69b63097e497bce8daa8ea4a80063f03136cfb7
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Thu May 4 20:00:43 2017 -0400

```
    Removed locals from bdecl in parser

commit 16b6a3a90201434469af4dbe6fd95e72b5162a5e
Merge: ae9eb16 490bd15
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Thu May 4 19:57:37 2017 -0400

    Merging standard library functions

commit ae9eb166a91525fa2b50fec338b5140501f8afaf
Merge: f68f186 77cfe17
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Thu May 4 19:50:17 2017 -0400

    Fixed merge conflict with buteverytime

commit 490bd158041b8d0912bcc71029ec82097dc0a9c7
Author: Chang Liu <cl3403@columbia.edu>
Date:   Wed May 3 22:45:43 2017 -0400

    test file for FILE I/O

commit 60cf221ec5da06c94e75bcef1b3cbc48848d5eaf
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Wed May 3 21:30:53 2017 -0400

    debugging file i/o

commit 99a113ee7dffbbffa474d790c7309b06b514ea49
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Wed May 3 19:32:47 2017 -0400

    Adding basic standard library and merging c calls into one file

commit f68f18644a96089d6253758898dbedaad704d641
Merge: 532ec3f 979f576
Author: William Essilfie <wke2102@columbia.edu>
```

```
Date:   Wed May 3 18:22:35 2017 -0400

    Merge pull request #13 from PLT-DCL/variable

    Making variable branch the master branch

commit 77cfe17946c73d07781d7c94d20d595a15cfeca2
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Tue May 2 00:02:55 2017 -0400

    Added buteverytime functionality

commit 979f576d40cc4238fd089c36b8d8935a4ee31a94
Author: Chang Liu <cl3403@columbia.edu>
Date:   Sun Apr 30 22:04:05 2017 -0400

    made variable declarations & assignment work

commit 532ec3fd61bf2ede0984d9c8e1766fd41ade347f
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Sun Apr 30 22:00:44 2017 -0400

    adding more array focused test files

commit 9c4fba0579bfeaee356ece647170f05c76c6c565
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Sun Apr 30 20:02:59 2017 -0400

    fixing failing test cases (func4)

commit cfe898d993a842d770446df82ae9b1503360df3f
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Sun Apr 30 19:58:38 2017 -0400

    adding new test cases

commit e7026065fc47d5999150753453b5e41274972c73
Author: Will Essilfie <wke2102@columbia.edu>
```

```
Date:   Sun Apr 30 17:36:59 2017 -0400

    removing printbig test

commit d64fdffde7b353b457a04ad72ae0d4e1c4f0e58f
Author: William Essilfie <wke2102@columbia.edu>
Date:   Sun Apr 23 01:32:45 2017 -0400

    updating README.md

commit 824a8fe0e7638f16a4e748cb773bf6139048ada7
Author: William Essilfie <wke2102@columbia.edu>
Date:   Sun Apr 23 01:31:55 2017 -0400

    make README  a README.md

commit e0a0b697cbd5ffbbf1b89553ceed47dbdc1175fa
Merge: 0002067 719d500
Author: Chang Liu <cl3403@columbia.edu>
Date:   Sat Apr 22 12:19:20 2017 -0400

    Merge branch 'adding-arrays' of https://github.com/PLT-DCL/dcl into adding-arrays

commit 9781f6e81a8844c50744a0e038c096ee0a1baf8e
Author: Craig D. Rhodes III <cdr2139@columbia.edu>
Date:   Sat Apr 22 12:12:28 2017 -0400

    Update README.md

commit 573f780e5a81e8a675476a26bc512eaf7168d083
Author: Craig D. Rhodes III <cdr2139@columbia.edu>
Date:   Sat Apr 22 12:12:16 2017 -0400

    Update README.md

commit 719d500c1289a67e2fbb43e30dee77aa284ee830
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Sat Apr 15 15:55:58 2017 -0400
```

```
    Fast-forwarded adding-arrays to new master

commit 6c6eb1dbd3c874940fd083984ad3bdfb92ec8b7e
Merge: 737ca08 012e3d6
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Sat Apr 15 15:53:24 2017 -0400

    Merge adding-arrays

commit 012e3d6e1389734c4242c5271c3e650a3dd44dfc
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Sat Apr 15 15:52:09 2017 -0400

    Removed _build directory

commit 8bd0e2cb570ff21f68cbe76c104bf85c6335eca0
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Sat Apr 15 15:42:32 2017 -0400

    adding string as global variable

commit 7191f6e1fc91d4c9a5946200d6b2c5d4b7cf0a04
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Sat Apr 15 15:06:30 2017 -0400

    working on print_double

commit 00020671cd77517b9189ba463560e170787709e2
Merge: 4f2666a a66fbb6
Author: Chang Liu <cl3403@columbia.edu>
Date:   Sat Apr 15 14:56:46 2017 -0400

    Merge branch 'adding-arrays' of https://github.com/PLT-DCL/dcl into adding-arrays

commit 4f2666a833681c0c4af90cec4c7a16290b99059c
Merge: 1572fe0 1c71dd7
Author: Chang Liu <cl3403@columbia.edu>
```

Date: Sat Apr 15 14:55:55 2017 -0400

    Merge branch 'adding-arrays' of https://github.com/PLT-DCL/dcl into adding-arrays

commit a66fbb608ff92d8ddca7a931ef11f2a842c2f1c8
Author: Will Essilfie <wke2102@columbia.edu>
Date: Sat Apr 15 14:55:52 2017 -0400

    adding bool

commit 1c71dd795c74cb2972aa9a2bd6cfbef5e5afd21d
Author: Chang <cl3403@columbia.edu>
Date: Sat Apr 15 13:48:57 2017 -0400

    changing .mc to .dcl

commit 1572fe0324cadf5c9b8b31c12062393ae4e3145e
Author: Chang Liu <cl3403@columbia.edu>
Date: Sat Apr 15 13:45:49 2017 -0400

    changing .mc to .dcl

commit 1d7cd52cc02d4c1f4b56397ef83559da8fcadb29
Author: Will Essilfie <wke2102@columbia.edu>
Date: Sat Apr 15 13:38:53 2017 -0400

    fixing if5 test

commit b39469541f4fafc755a09deae10052b089e81360
Author: Will Essilfie <wke2102@columbia.edu>
Date: Sat Apr 15 13:38:08 2017 -0400

    fixing if1 test

commit acd5745345b9aa787d98b8557a11206c7c25ef89
Author: Will Essilfie <wke2102@columbia.edu>
Date: Sat Apr 15 13:31:04 2017 -0400

```
    remove square

commit 916fbc948371900fcd20aab9ce596c5d2fa4a8e0
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Sat Apr 15 13:30:14 2017 -0400

    some removals

commit 5c5f6856cc4c496d02714f32aa084d6c7bd557aa
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Sat Apr 15 13:28:54 2017 -0400

    removing array

commit e14d846a9d7209f40ee449367f717a7c619b6f9e
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Sat Apr 15 13:26:50 2017 -0400

    removing some array stuff

commit d6e06f20c5739e45a39e386fe9ce42b40f8f2549
Merge: 5beb782 9c726a1
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Sat Apr 15 13:22:28 2017 -0400

    Merge branch 'bool' of https://github.com/PLT-DCL/dcl into adding-arrays

commit 5beb782e6767bc3a306e01a1beee7acd876833c4
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Sat Apr 15 13:21:14 2017 -0400

    Updating test suite

commit 9c726a155ed85eee4c4a29d2b5436909b3f9d8e3
Author: Chang <cl3403@columbia.edu>
Date:   Sat Apr 15 13:16:09 2017 -0400

    Change "Boolean" to "Bool"
```

```
commit 259e0abbb06ee03098a533da359d50cd011bdf08
Author: Chang <cl3403@columbia.edu>
Date:   Sat Apr 15 13:14:00 2017 -0400

    Change "Boolean" to "Bool"

commit ab254d4855fb8a35057dfddca9fd8cbc90c13c36
Author: Chang <cl3403@columbia.edu>
Date:   Sat Apr 15 13:12:35 2017 -0400

    Change "Boolean" to "Bool"

commit b56bd94adef581b5335b279eb728451c1a9391f2
Merge: 61488d6 d12f83f
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Sat Apr 15 13:02:12 2017 -0400

    Merge branch 'bool' of https://github.com/PLT-DCL/dcl into adding-arrays

commit d12f83f387227e971ffda4483e523a9a2c1a207d
Author: William Essilfie <wke2102@columbia.edu>
Date:   Sat Apr 15 13:00:42 2017 -0400

    update test-add

commit 1f6bd1c376882d54a7857c04a70503bd4886dfc2
Author: William Essilfie <wke2102@columbia.edu>
Date:   Sat Apr 15 12:59:21 2017 -0400

    updating to print_int

commit 293101cfad56a933966790da83924bbd3f3fb1e2
Author: William Essilfie <wke2102@columbia.edu>
Date:   Sat Apr 15 12:25:31 2017 -0400

    fixing bool issues
```

commit 61488d6fadb98a3c998d152fafb8cfc320d3dac2
Author: Will Essilfie <wke2102@columbia.edu>
Date: Thu Apr 13 20:13:48 2017 -0400

  starting work on adding arrays

commit ab460ca89964ee89f2111d5d4facae5d801c0911
Author: Will Essilfie <wke2102@columbia.edu>
Date: Thu Apr 13 16:24:49 2017 -0400

  adding bool back to DCL

commit 17d17527c2789839f91bc601f97e40b881ac3408
Author: Chang Liu <cl3403@columbia.edu>
Date: Wed Apr 12 22:12:40 2017 -0400

  added bool back to codegen.ml in this new branch

commit 4c8474739da774da90f534989189f9f60b2597e8
Author: Chang Liu <cl3403@columbia.edu>
Date: Wed Apr 12 22:04:31 2017 -0400

  added bool back to ast.ml

commit 737ca0838a9324a8616e9918ee9a2a3fa781cb8b
Author: Will Essilfie <wke2102@columbia.edu>
Date: Wed Apr 12 21:30:54 2017 -0400

  adding string into scanner

commit a17f04f4720096d0e3fb129becaf7246e158226a
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Sun Apr 9 19:45:25 2017 -0400

  Implement string comparison

commit d4dc2bda25f104d2a824806db3a679eff24a964d
Merge: 8d86030 8a4d964

Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Sun Apr 9 19:10:38 2017 -0400

    Merge branch 'making-doubles-work' of https://github.com/PLT-DCL/dcl into making-doubles-work

commit 8d860305cfb74692d03958ec691672a2d8cce10d
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Sun Apr 9 19:04:07 2017 -0400

    Make string concatenation work

commit 8a4d964502f9313a7ecb3ee38d58696db9908662
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sun Apr 9 18:38:17 2017 -0400

    Modified Makefile to clean every time

commit 412f7e75ba785616f416524f4e0bce2b22d70b18
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sun Apr 9 18:12:28 2017 -0400

    Modified docker Makefile to stop assigning a name to container except if you want to modify the iamge

commit 887ae6a69d7262c9e1827f8df60830e6f4dcf1f4
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sun Apr 9 17:52:28 2017 -0400

    Updated file extensions to use .dcl

commit f17e689cc2a1d1d990f0fc2beba5ae8bd56026cc
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Sat Apr 1 13:25:39 2017 -0400

    Make double quoted strings usable again

commit c7fb0e4f3921ca36f727c7278ea058d95825f79d

Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Sat Apr 1 00:55:17 2017 -0400

    Updated the example script

commit 2be370abdda5ff3d507875bfd6ba2ebbe30b39a2
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Fri Mar 31 06:57:32 2017 -0400

    Exponentiation and Hello World

    They said it couldn't be done; they were wrong.
    (Not complete coverage on strings yet..)

commit 4a63b15124ad5064c38a0184c094699d6893edcf
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Mon Mar 27 12:41:36 2017 -0400

    Provide full functionality for double

    We finally have the extra type we need to make tests work.

commit 518e83fe9d8a2231f7c35d1611f3d2f2b9223100
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Mon Mar 27 00:09:31 2017 -0400

    Doubles still aren't working... but maybe

commit f337db26ae07398cfa21180e903455f98df32b9c
Author: ashutoshnanda <ashutosh.nanda@gmail.com>
Date: Sat Mar 25 12:42:35 2017 -0400

    Remove bools and change Literal -> IntLiteral

    We need to remove these holdovers from MicroC since DCL doesn't have them.

commit 97f0b4eed0b534d315237f215d761bfa9a99d06b
Author: Craig Rhodes <cdr2139@columbia.edu>

Date: Sat Mar 25 12:11:15 2017 -0400

    Updated microc to one with bindings

commit 5c6b32877a93d283c5e0cbcd61966b06b3c975d2
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Sat Mar 25 11:45:42 2017 -0400

    Updated Makefile to work with /bin/bash entrypoint for docker image

commit 2dff55b37c053bbaefab15a61021cbaa652a742f
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Tue Mar 21 02:01:00 2017 -0400

    Added clean as prerequisite (before compile) to test target in docker Makefile

commit c757c5140cee36e0f1262f189d38fe73b52a338c
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Mon Mar 20 18:49:08 2017 -0400

    Made README look better

commit 9738808fa80bfd3949073c6e5c63241086f2262e
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Mon Mar 20 18:47:36 2017 -0400

    Made README look better

commit 15465811d011c0abbf0c55fa6cded42e203b7425
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Mon Mar 20 18:42:44 2017 -0400

    Improved README format

commit d4d71f8808941f5e68148c478e5c734b3875fe5d
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Mon Mar 20 18:35:48 2017 -0400

    Added CircleCI status badge to README

commit ce61cf6214e2c33e9623f678c7984c982ecb9623
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Mon Mar 20 18:20:32 2017 -0400

    Updated circle.yml with spaces instead of tabs

commit 4fd214fa5b18d8b15c5148f5d732fff38d05a8ce
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Mon Mar 20 18:17:29 2017 -0400

    Added a circle.yml file for CI

commit 260df19825f73b5171d4d1d2165708cfe97defd6
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Mon Mar 20 17:32:21 2017 -0400

    Added a Dockerfile and modified docker Makefile to automatically run the testall.sh script

commit b7e3876881d8c9c4bb490c270c7b5d05ff5ecf5b
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Mon Mar 20 15:13:34 2017 -0400

    Updated docker Makefile for more easily modifying docker image

commit c2cac07d7bac308db943a2e8464152f707f628c7
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Tue Mar 14 23:15:43 2017 -0400

    Made docker Makefile remove containers on exit

commit 39a95326115aafaa63554ae639debedec0e94130
Author: Craig Rhodes <cdr2139@columbia.edu>
Date: Mon Mar 13 20:38:38 2017 -0400

    Updated docker Makefile to not commit on pushing

```
commit 8269dbff1702fac66cd2c38c7a5a228465dcd90a
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Mon Mar 13 15:32:00 2017 -0400

    Updated README.md with new commands inside docker environment

commit 19573c72b12b40ed3ee8ee5306f61c3e4e975c60
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Mon Mar 13 15:28:55 2017 -0400

    Modified README.md to avoid a step

commit 55119f3426ef3e7f95f0cadfafd8278e85ed361b
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Mon Mar 13 15:23:32 2017 -0400

    Updated README.md with compile and run instructions

commit cb820c25ae935316b085bf98f04e24d8a76c8735
Author: Will Essilfie <wke2102@columbia.edu>
Date:   Thu Mar 2 23:15:34 2017 -0500

    updated README

commit 7ba701379fe1700fb51f8d066f0fc033c5d68e9e
Author: Craig D. Rhodes III <cdr2139@columbia.edu>
Date:   Thu Mar 2 15:43:16 2017 -0500

    Set theme jekyll-theme-cayman

commit 16b6f099b73a32f2e40894f7ec22a6f316204806
Author: Craig Rhodes <cdr2139@columbia.edu>
Date:   Thu Mar 2 14:27:53 2017 -0500

    Initial commit.
```

Software Development Environment

For this project, we were able to save a lot of time by using a Docker image to make working and running files easy on any machine. Our Docker image was loaded with the following files.

- Ubuntu 15.10 - We used Ubuntu as an easy way to access OCaml and LLVM software. This also helped for setting the standard for our Docker image.
- LLVM-3.7 - We used the most recent LLVM version to ensure easy code generation in OCaml.
- Facebook Messenger - We used Messenger to communicate with one another because it was easy to use and a platform most of us are on often. This made it easy to quickly ping the group to get help and search the message history for past discussions on bugs we had in the language.
- GitHub - We used GitHub to save our commits and maintain our different branches. This made it a lot easier to get work done and revert to old versions in case something broke.
- Sublime Text - We used Sublime Text for writing code because it was fast and easy to use.

Programming Style Guide

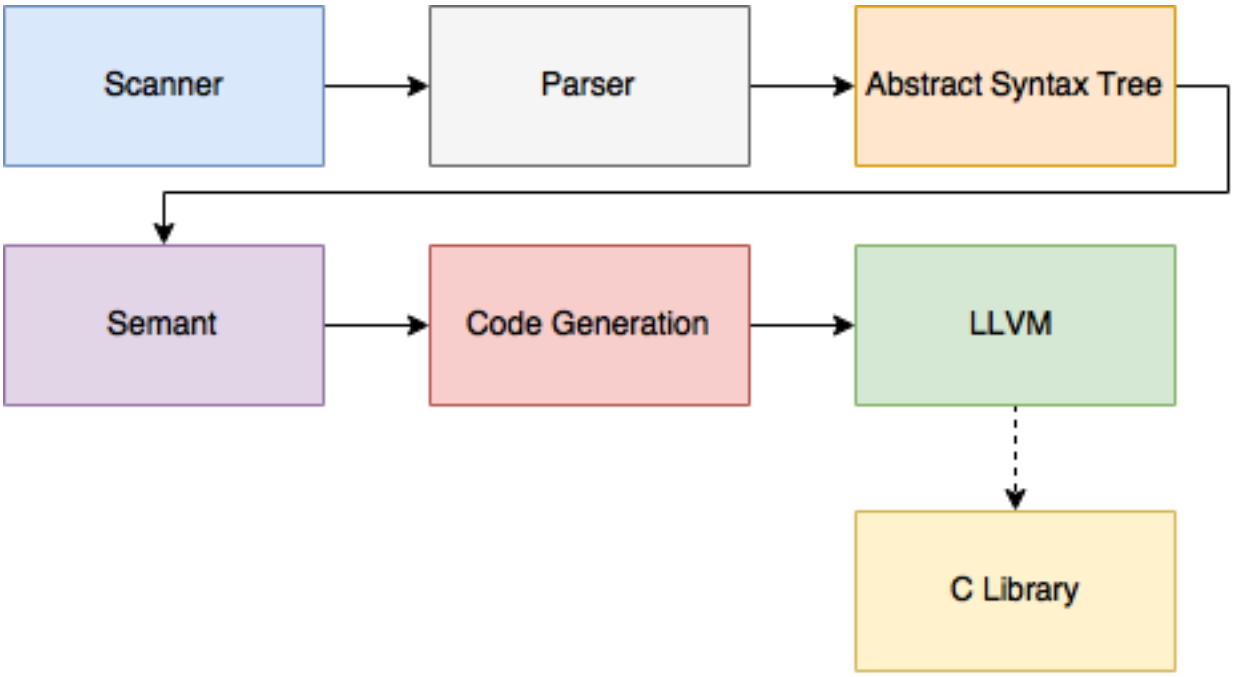
To make it easy for everyone to understand our code base we tried to adhere to the following programming style guide:

- Keep lines shorter than 80 characters
- When making major code changes, comment the code so others can understand the rationale
- If a certain algorithm is needed multiple time, split it out into a new function

Chapter 5: Architectural Design

Diagram Overview

A diagram laying out the setup of DCL can be found below.



Scanner

Scanner takes in a DCL program and tokenizes based on the defined tokens laid out by the language. This is a quick and easy process with the main work being the regular expressions meant to identify special types such as strings, integers, and escaped characters.

Parser

Parser requires a tokenized program (created by Scanner) and runs the program through the Context Free Grammar (CFG) that comprises the DCL language. If the program passes, an Abstract Syntax Tree (AST) is successfully built that now represents the original program. For dealing with associativity, this is set for most operators allowed in the language to avoid issues related to ambiguity. In addition, we focused on avoiding shift/reduce errors in the language.

Semant

Semant does a postorder traversal of the Abstract Syntax Tree. This is helpful for verifying type matching, ensuring that a user does not use specific keywords for function names. If an error is found, the program will fail to compile and the user will get a message with some details of the error in their console.

Codegen

Once the semantic check has occured successfully, code generator uses the abstract syntax tree to create the LLVM IR file. This file specifies the final instructions for the program that is now ready to be executed.

Who Did What

As stated also in Chapter 4, each member played an important role in getting this project completed:

Team Member	**Responsibilities**
Ashutosh Nanda	Doubles, Arrays, Codegen, Parser
Chang Liu	Test Suite, Strings, Parser, Codegen, Semant
Craig Rhodes	Callbacks, Circle CI, Docker, AST, LRM
William Essilfie	Scanner, Parser, Test Suite, Semant, Final Report

Chapter 6: Test Plan

Unit and Integration Testing

Testing was a major part of our project because of its size and scope. When run, the scanner, parser, semantic analyzer, and code generation files need to run and execute as specified by us. In fact, we relied on test driven development when adding new features. For a new feature to be added, we went through the following steps:

1. Create a sample program that follows intended specifications for the new feature.
2. Open scanner file and add new tokens if needed.

3. Update the parser file with new rules if the new features changes the language grammar.
4. Test that sample program accepted by parser and scanner. Verify that wrong implementations fail. If there are errors here, return to previous two steps.
5. Update semant file to check for possible issues with the feature's semantics.
6. Repeat step four, but include testing on semant file.
7. In the code generation file, work on generating the correct LLVM IR code for the added feature.
8. Update sample program to actually use your feature to ensure it works.
9. Continue testing the new feature with sample programs and modifying code generation as needed.
10. Save sample programs and add to test files directory.

While this process is tedious, it was helpful for ensuring that new features were implemented correctly and helped stop issues from occurring in later features because they relied on other new features. This was also useful for verifying new features did not break old features. For our testing purposes, we mostly used integration testing since new features needed to go through the entire system flow to be effectively tested. Unit testing was not as useful because a specific unit test could pass but the entire system could now have a bug because of that the original unit test failed to catch. Our test cases were based on the current test cases from MicroC, a language written by Stephen Edwards at Columbia University, as well as new test cases added to reflect DCL-specific language features.

Test Suite and Automated Regression Testing

Part of DCL's test suite is based on MicroC's automated regression testing system. Within the tests directory, we have included scores of tests that cover various aspects of our language along with their expected output. In addition, we have test cases for programs that should fail and the expected errors that these programs should produce.

This automated system was useful for testing language features because within seconds, the system can run all the test files and report which files passed and which failed. This meant when a new featured was tested, it did not take long to discover if it was functioning correctly. To use the test script, a user just needs to type ./testall.sh from their command line console.

In addition to using a script, we set up Continuous Integration for DCL. This meant that whenever the master branch of DCL's GitHub is updated, a system automatically runs all of the current test cases and reports whether the test suite passed. It also updates the button on the DCL Github page showing the language's status and notifies the team via email. This made it easy to see if a merge caused issues in the language and allows the team to quickly rectify any issues. Lastly, because our group used test driven development, all of us worked on making test cases for the language.

Test Cases and Outputs

fail-assign1.dcl:

```
int main()
{
  int i;
  double b;

  i = 42;
  b = 25.7;
  i = 25.7; /* Fail: assigning a double to an integer */
}
```

fail-assign1.err:

```
Fatal error: exception Failure("illegal assignment int = double in i = 25.7")
```

fail-assign2.dcl:

```
int main()
{
  int i;
  string b;

  b = 48; /* Fail: assigning an integer to a bool */
}
```

fail-assign2.err:
Fatal error: exception Failure("illegal assignment string = int in b = 48")

fail-assign3.dcl:

```
void myvoid()
{
  return;
}

int main()
{
  int i;

  i = myvoid(); /* Fail: assigning a void to an integer */
}
```

fail-assign3.err:
Fatal error: exception Failure("illegal assignment int = void in i = myvoid()")

fail-concatenation1.dcl:

```
void main() {
        string g;
        int[] h;
        string i;
        g = "hello";
        h = [3,4,5,6];
        i = g + h;
}

/* should fail because string + array concatenation not supported */
```

fail-concatenation1.err:
Fatal error: exception Failure("illegal binary operator string + int[] in g + h")

fail-dead1.dcl:

```
int main()
{
  int i;

  i = 15;
  return i;
  i = 32; /* Error: code after a return */
}
```

fail-dead1.err:
Fatal error: exception Failure("nothing may follow a return")

fail-dead2.dcl:

```
int main()
{
  int i;

  {
    i = 15;
    return i;
  }
  i = 32; /* Error: code after a return */
}
```

fail-dead2.err:

```
Fatal error: exception Failure("nothing may follow a return")
```

fail-exponents1.dcl:

```
double main() {
        double a;
        string b;
        b = "hello";
        a = b^2;
        print_line(a);
        return a;
}

/* verify exponents only work on doubles */
```

fail-exponents1.err:

```
Fatal error: exception Failure("illegal binary operator string ^ int in b ^ 2")
```

fail-expr1.dcl:

```
int a;
double b;

void foo(int c, double d)
{
  int dd;
  double e;
  a + c;
  c - a;
  a * 3;
  c / 2;
  d + a; /* Error: double + int */
}

int main()
```

```
{
  return 0;
}
```

fail-expr1.err:

```
Fatal error: exception Failure("illegal binary operator double + int in d + a")
```

fail-expr2.dcl:

```
int a;
double b;

void foo(int c, double d)
{
  int d;
  double e;
  b + a; /* Error: double + int */
}

int main()
{
  return 0;
}
```

fail-expr2.err:

```
Fatal error: exception Failure("illegal binary operator double + int in b + a")
```

fail-for1.dcl:

```
int main()
{
  int i;
```

```
  for ( ; 1 ; ) {} /* OK: Forever */

  for (i = 0 ; i < 10 ; i = i + 1) {
    if (i == 3) return 42;
  }

  for (j = 0; i < 10 ; i = i + 1) {} /* j undefined */

  return 0;
}
```

fail-for1.err:
Fatal error: exception Failure("undeclared identifier j")

fail-for2.dcl:

```
int main()
{
  int i;

  for (i = 0; j < 10 ; i = i + 1) {} /* j undefined */

  return 0;
}
```

fail-for2.err:
Fatal error: exception Failure("undeclared identifier j")

fail-for3.dcl:

```
int main()
{
```

```
  double j;

  for (int i = 0; j ; i = i + 1) {} /* j is a double, not an integer */

  return 0;
}
```

fail-for3.err:
Fatal error: exception Failure("expected int expression in j")

fail-for4.dcl:

```
int main()
{
  int i;

  for (i = 0; i < 10 ; i = j + 1) {} /* j undefined */

  return 0;
}
```

fail-for4.err:
Fatal error: exception Failure("undeclared identifier j")

fail-for5.dcl:

```
int main()
{
  int i;

  for (i = 0; i < 10 ; i = i + 1) {
    foo(); /* Error: no function foo */
```

```
  }

  return 0;
}
```

fail-for5.err:
Fatal error: exception Failure("unrecognized function foo")

fail-func1.dcl:

```
int foo() {}

int bar() {}

int baz() {}

void bar() {} /* Error: duplicate function bar */

int main()
{
  return 0;
}
```

fail-func1.err:
Fatal error: exception Failure("duplicate function bar")

fail-func2.dcl:

```
int foo(int a, double b, int c) {}

void bar(int a, double b, int a) {} /* Error: duplicate formal a in bar */
```

```
int main()
{
  return 0;
}
```

fail-func2.err:

```
Fatal error: exception Failure("duplicate variable a in bar")
```

fail-func3.dcl:

```
int foo(int a, double b, int c) { }

void bar(int a, void b, int c) {} /* Error: illegal void formal b */

int main()
{
  return 0;
}
```

fail-func3.err:

```
Fatal error: exception Failure("illegal void formal b in bar")
```

fail-func4.dcl:

```
int foo() {}

void bar() {}

int print() {} /* Should not be able to define print */

void baz() {}
```

```
int main()
{
  return 0;
}
```

fail-func4.err:

```
Fatal error: exception Failure("function print may not be defined")
```

fail-func5.dcl:

```
int foo() {}

int bar() {
  int a;
  void b; /* Error: illegal void local b */
  double c;

  return 0;
}

int main()
{
  return 0;
}
```

fail-func5.err:

```
Fatal error: exception Failure("illegal void local b in bar")
```

fail-func6.dcl:

```
void foo(int a, double b)
{
```

```
}

int main()
{
  foo(42, 32.3);
  foo(42); /* Wrong number of arguments */
}
```

fail-func6.err:

```
Fatal error: exception Failure("expecting 2 arguments in foo(42)")
```

fail-func7.dcl:

```
void foo(int a, double b)
{
}

int main()
{
  foo(42, 23.4);
  foo(42, 23.5, 636.1); /* Wrong number of arguments */
}
```

fail-func7.err:

```
Fatal error: exception Failure("expecting 2 arguments in foo(42, 23.5, 636.1)")
```

fail-func8.dcl:

```
void foo(int a, double b)
{
}
```

```
void bar()
{
}

int main()
{
  foo(42, 32.5);
  foo(42, bar()); /* int and void, not int and double */
}
```

fail-func8.err:

```
Fatal error: exception Failure("illegal actual argument found void expected double in
bar()")
```

fail-func9.dcl:

```
void foo(int a, double b)
{
}

int main()
{
  foo(42, 23.3);
  foo(42, 42); /* Fail: int, not double */
}
```

fail-func9.err:

```
Fatal error: exception Failure("illegal actual argument found int expected double in
42")
```

fail-global1.dcl:

```
int c;
double b;
void a; /* global variables should not be void */


int main()
{
  return 0;
}
```

fail-global1.err:

```
Fatal error: exception Failure("illegal void global a")
```

fail-global2.dcl:

```
int b;
double c;
int a;
int b; /* Duplicate global variable */

int main()
{
  return 0;
}
```

fail-global2.err:

```
Fatal error: exception Failure("duplicate global b")
```

fail-if1.dcl:

```
int main()
{
```

```
  if (1) {}
  if (0) {} else {}
  if (0.5) {} /* Error: non-int predicate */
}
```

fail-if1.err:

```
Fatal error: exception Failure("expected int expression in 0.5")
```

fail-if2.dcl:

```
int main()
{
  if (1) {
    foo; /* Error: undeclared variable */
  }
}
```

fail-if2.err:

```
Fatal error: exception Failure("undeclared identifier foo")
```

fail-if3.dcl:

```
int main()
{
  if (1) {
    42;
  } else {
    bar; /* Error: undeclared variable */
  }
}
```

fail-if3.err:
Fatal error: exception Failure("undeclared identifier bar")

fail-nomain.dcl:

fail-nomain.err:
Fatal error: exception Failure("unrecognized function main")

fail-return1.dcl:

```
int main()
{
  return 0.5; /* Should return int */
}
```

fail-return1.err:
Fatal error: exception Failure("return gives double expected int in 0.5")

fail-return2.dcl:

```
void foo()
{
  if (1) return 42; /* Should return void */
  else return;
}

int main()
{
  return 42;
```

```
}
```

fail-return2.err:

```
Fatal error: exception Failure("return gives int expected void in 42")
```

fail-string1.dcl:

```
void main() {

        string[] g;
        g = ['hello', 'from', 'the', 'other', 'side'];
        print_line(g{|0|});
        int value = g{|2|};

}

/* verifies that setting string to int fails */
```

fail-string1.err:

```
Fatal error: exception Failure("illegal assignment int = string in int value = g{|2|}")
```

fail-string2.dcl:

```
void main() {
        string g;
        int h;
        string i;
        g = "hello";
        h = 3;
        i = g + h;
}
/* should fail because string + int concatenation not allowed */
```

fail-string2.err:
Fatal error: exception Failure("illegal binary operator string + int in g + h")

fail-string3.dcl:

```
void main() {
        string g;
        double h;
        string i;
        g = "hello";
        h = 3.0;
        i = g + h;
}

/* should fail because string + double concatenation not allowed */
```

fail-string3.err:
Fatal error: exception Failure("illegal binary operator string + double in g + h")

fail-while1.dcl:

```
int main()
{
  int i;

  while (1) {
    i = i + 1;
  }

  while (32.5) { /* Should be boolean */
    i = i + 1;
  }

}
```

fail-while1.err:
Fatal error: exception Failure("expected int expression in 32.5")

fail-while2.dcl:

```
int main()
{
  int i;

  while (1) {
    i = i + 1;
  }

  while (1) {
    foo(); /* foo undefined */
  }

}
```

fail-while2.err:
Fatal error: exception Failure("unrecognized function foo")

test-add1.dcl:

```
int main()
{
        int a;
        int b;
        a = 17;
        b = 25;
```

```
        print_line(a+b);
  return 0;
}
```

test-add1.out:

```
42
```

test-arith1.dcl:

```
int main()
{
  print_line(39 + 3);
  return 0;
}
```

test-arith1.out:

```
42
```

test-arith2.dcl:

```
int main()
{
  print_line(1 + 2 * 3 + 4);
  return 0;
}
```

test-arith2.out:

```
11
```

test-arith3.dcl:

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

int main()
{
  int a;
  a = 42;
  a = a + 5;
  print_line(a);
  return 0;
}
```

test-arith3.out:

```
47
```

test-array1.dcl:

```
void main() {
        int a;
        int[] b;
        b = [4,4,5];
        print_line(b{|0|});

}

/* verifies printing a value at array index works */
```

test-array1.out:

```
4
```

test-array2.dcl:

```
void main() {
	int a;
	double[] b;
	b = [4.1,4.2,5.3];
	print_line(b{|1|});

}

/* verifies printing a value at array index works -- for doubles */
```

test-array2.out:

```
4.2
```

test-array3.dcl:

```
void main() {
	int a;
	string[] b;
	b = ['craig','will','chang'];
	print_line(b{|2|});

}

/* verifies printing a value at array index works -- for doubles */
```

test-array3.out:

```
chang
```

test-callbacks1.dcl:

```
int x = 0 buteverytime (x==2) { print_line(x); }

int main() {
	x = 2;
```

```
        print_line("Hello");
        x = 1;
        print_line("world");
        x = 2;
        x = 0;

}

/* test setup up of a callback set globally */
```

test-callbacks1.out:

```
2
Hello
World
2
```

test-callbacks2.dcl:

```
int y = 0 buteverytime (y==3) {
    print_line("y is 3 now!!!");
    y = 0;
}

int main() {
        y = 3;
        print_line("Testing 1");
        y = 1;
        print_line("Testing 2");
        y = 3;
        y = 2;
   return 0;
}

/* test setup up of a callback set globally */
```

test-callbacks3.dcl:

```
double z = 3.5 buteverytime (z == 1.414) {
      print_line("z is the square root of 2 now! And we are changing it back to 3.5");
      z = 3.5;

}

int main() {
    z = 2.3;
    z = 1.414;
    z = 6.7;
    z = 5.2;
    return 0;

}

 /* testing callbacks with double */


test-callbacks4.dcl:
int x = 5 buteverytime (x == 0) {
            print_line("x is zero now! x should not be 0! Setting it back to 5");
            x = 5;

}

int y = 10 buteverytime (y == 0) {
            print_line("y is also zero now, and y should not be 0. Setting it back to 10");
            y = 10;
}


double z;
int c;

int main() {
    z = 3.2;
    x = 0;
    y = 0;
```

```
    c = 4;

    return 0;

}

 /* testing several callbacks together */


test-double1.dcl:
double a;
double b = 3.5;

int main() {
    a = 3.5;
          double c = a + b;
          print_line(c);
          return 0;
}

/* testing doubles */


test-fib.dcl:
int fib(int x)
{
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}

int main()
{
  print_line(fib(0));
  print_line(fib(1));
  print_line(fib(2));
  print_line(fib(3));
  print_line(fib(4));
  print_line(fib(5));
```

```
  return 0;
}
```

test-fib.out:

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

test-for1.dcl:

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

test-for1.out:

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

test-for2.dcl:

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

test-for2.out:

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

test-func1.dcl:

```
int add(int a, int b)
{
  return a + b;
}

int main()
{
  int a;
  a = add(39, 3);
  print_line(a);
  return 0;
}
```

test-func1.out:
42

test-func2.dcl:

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

int main()
{
  int i;
  i = 1;

  fun(i = 2, i = i+1);

  print_line(i);
  return 0;
}
```

test-func2.out:
2

test-func3.dcl:

```
void printem(int a, int b, int c, int d)
{
  print_line(a);
```

```
  print_line(b);
  print_line(c);
  print_line(d);
}

int main()
{
  printem(42,17,192,8);
  return 0;
}
```

test-func3.out:

```
42
17
192
8
```

test-func4.dcl:

```
int add(int a, int b)
{
  int c;
  c = a + b;
  return c;
}

int main()
{
  int d;
  d = add(52, 10);
  print_line(d);
  return 0;
}
```

test-func4.out:
62

test-func5.dcl:
```
int foo(int a)
{
  return a;
}

int main()
{
  return 0;
}
```

test-func5.out:

test-func6.dcl:
```
int bar(int a, int c) { return a + c; }

int main()
{
  print_line(bar(17, 25));
  return 0;
}
```

test-func6.out:
42

test-func7.dcl:

```
int a;

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

int main()
{
  foo(73);
  print_line(a);
  return 0;
}
```

test-func7.out:

```
115
```

test-func8.dcl:

```
void foo(int a)
{
  print_line(a + 3);
}

int main()
{
  foo(40);
  return 0;
}
```

test-func8.out:

43

test-gcd.dcl:

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

int main()
{
  print_line(gcd(2,14));
  print_line(gcd(3,15));
  print_line(gcd(99,121));
  return 0;
}
```

test-gcd.out:

```
2
3
11
```

test-gcd2.dcl:

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

```
int main()
{
  print_line(gcd(14,21));
  print_line(gcd(8,36));
  print_line(gcd(99,121));
  return 0;
}
```

test-gcd2.out:

```
7
4
11
```

test-global1.dcl:

```
int a;
int b;

void printa()
{
  print_line(a);
}

void printb()
{
  print_line(b);
}

void incab()
{
  a = a + 1;
  b = b + 1;
}

int main()
```

```
{
  a = 42;
  b = 21;
  printa();
  printb();
  incab();
  printa();
  printb();
  return 0;
}
```

test-global1.out:

```
42
21
43
22
```

test-global2.dcl:

```
double i;

int main()
{
  int i; /* Should hide the global i */

  i = 42;
  print_line(i + i);
  return 0;
}
```

test-global2.out:

```
84
```

test-global3.dcl:

```
int i;
double b;
int j;

int main()
{
  i = 42;
  j = 10;
  print_line(i + j);
  return 0;
}
```

test-global3.out:

```
52
```

test-hello.dcl:

```
int main()
{
  print_line(42);
  print_line(71);
  print_line(1);
  return 0;
}
```

test-hello.out:

```
42
71
1
```

test-if1.dcl:

```
int main()
{
  if (1) print_line(42);
  print_line(17);
  return 0;
}
```

test-if1.out:

```
42
17
```

test-if2.dcl:

```
int main()
{
  if (1) print_line(42); else print_line(8);
  print_line(17);
  return 0;
}
```

test-if2.out:

```
42
17
```

test-if3.dcl:

```
int main()
{
  if (0) print_line(42);
```

```
  print_line(17);
  return 0;
}
```

test-if3.out:

```
17
```

test-if4.dcl:

```
int main()
{
  if (0) print_line(42); else print_line(8);
  print_line(17);
  return 0;
}
```

test-if4.out:

```
8
17
```

test-if5.dcl:

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

```
int main()
{
 print_line(cond(1));
 print_line(cond(0));
 return 0;
}
```

test-if5.out:

```
42
17
```

test-interleave1.dcl:

```
int main() {
        int a;
        int c;
        a = 5;
        int d;
        d = 5;

        print_line(a);
        print_line(7);
        print_line(d);
        return a;
}

/* verifies interleaving of code and variable initialization */
```

test-interleave1.out:

```
5
7
5
```

test-interleave2.dcl:

```
double test(double x) {
	return x;

}

int main() {
	test(4.3);
	int a;
	a = 5;
	print_line(a);
	return a;
}

/* interleaving calling other functions and initializing variables */
```

test-interleave2.out:

```
4.3
5
```

test-interleave3.dcl:

```
string hello_world() {
	print_line("hello world!");
	return "hello world!";
}

int main() {
	int a;
	int b;
	a = 5;
	b = 10;
	hello_world();
	string s;
	s = "hello world again!";
```

```
        print_line(s);

}

/* verifying that interleaving of initializing variables and calling functions */
```

test-interleave3.out:

```
hello world!
hello world again!
```

test-local1.dcl:

```
void foo(double i)
{
  int i; /* Should hide the formal i */

  i = 42;
  print_line(i + i);
}

int main()
{
  foo(3.5);
  return 0;
}
```

test-local1.out:

```
84
```

test-local2.dcl:

```
int foo(int a, double b)
{
```

```
  int c;
  double d;

  c = a;

  return c + 10;
}

int main() {
 print_line(foo(37, 3.5));
 return 0;
}
```

test-local2.out:

```
47
```

test-ops1.dcl:

```
int main()
{
  print_line(1 + 2);
  print_line(1 - 2);
  print_line(1 * 2);
  print_line(100 / 2);
  print_line(99);
  print_line(1 == 2);
  print_line(1 == 1);
  print_line(99);
  print_line(1 != 2);
  print_line(1 != 1);
  print_line(99);
  print_line(1 < 2);
  print_line(2 < 1);
  print_line(99);
  print_line(1 <= 2);
```

```
  print_line(1 <= 1);
  print_line(2 <= 1);
  print_line(99);
  print_line(1 > 2);
  print_line(2 > 1);
  print_line(99);
  print_line(1 >= 2);
  print_line(1 >= 1);
  print_line(2 >= 1);
  return 0;
}
```

test-ops1.out:

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

1

test-ops2.dcl:

```
int main()
{
  print_line(-10);
  print_line(--42);
}
```

test-ops2.out:

```
-10
42
```

test-string1.dcl:

```
int hamilton() {
        int thomasjefferson = 1789;
        int myshot = 1;
        return thomasjefferson;
}

string colorpurple() {
        string leadin = "Like a blade of corn, like a honey bee";
        string color = "Like the color purple. Where do it come from?";
        string response = leadin + color;
        print_line(response);
        return response;

}

double groundhog_day() {
        double dayone = 42.0;
        double daytwo = 42.0;
```

```
        double daythree = 42.0;
        print_line("fasfas");
        return daythree;


}

int main(){
        int ham = hamilton();
        colorpurple();
        groundhog_day();
        print_line(ham);
        return ham;


}
/* verifies string concatenation */
```

test-string1.out:

```
Like a blade of corn, like a honey bee Like the color purple. Where do it come from?
it's groundhog day!!!!!!!
1789
```

test-string2.dcl:

```
void main() {
        string g;
        string[] h;
        string i;
        g = "hello";
        h = ['world', 'ghana'];
      i = (h{|0|}) + g;
        print_line(i);
}
```

```
/*should work because string + string concatenation is supported */
```

test-string2.out:
hello world

test-tilde1.dcl:

```
int i = 7 buteverytime (i == 0 && ~i != 0) {
	print_line("You might do it, but don\'t divide by zero accidentally!");
}

int return_one_if_not_empty(string hi) {
	return #hi > 0;
}

string craig = "Hello" buteverytime(~craig != "Hello" && craig == "Hello") {
	print_line("Previously on DCL... " + ~craig);
	print_line("You said hello to Craig");
}

string chang = "all over the place" buteverytime(return_one_if_not_empty(chang)) {
	print("chang..");
}

void main() {
	i = 0;
	i = 0;
	i = 7;
	i = 0;
	i = 0;
	int bye_craig = 47;
	craig = "bye";
	craig = "Hello";
	if("hi" <= "hi") {
		print_line("hi");
	}
```

```
}

/* testing tilde operator */


test-tilde2.dcl:
double a = 7.2 buteverytime (a == 1.414 && ~a != 1.414) {
        print_line("Careful! a is now the square root of 2 ");
}

int b = 3 buteverytime(b == 10 && ~b != 10) {
        print_line("b is now 10!");
}


void main() {
        a = 0.3;
        b = 2;
        a = 1.414;
        print(a);
        b = 10;
        print(b);
}

/* should only print each message only once */


test-tilde3.dcl:
string x;
double y;

int a = 4 buteverytime(a == -1 && ~a != -1) {
        print_line("a is -1 now which is dangerous");
}

string c = "Hi" buteverytime (c == "Hola" && ~c != "Hola") {
        print_line("c is now in Spanish");
}
```

```
void main() {
   x = "DCL";
   y = 3.5;
         a = 0;
         c = "Guten Tag";
         a = 3;
         c = "Hola";
         a = -1;
         c = "Konnichiwa";
}

/* testing callbacks, tilde operator, and global variables altogether */
```

test-var1.dcl:

```
int main()
{
  int a;
  a = 42;
  print_line(a);
  return 0;
}
```

test-var1.out:

```
42
```

test-var2.dcl:

```
int a;

void foo(int c)
{
  a = c + 42;
```

```
}

int main()
{
  foo(73);
  print_line(a);
  return 0;
}
```

test-var2.out:

```
115
```

test-while1.dcl:

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

test-while1.out:

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

test-while2.dcl:

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

int main()
{
  print_line(foo(7));
  return 0;
}
```

test-while2.out:

```
14
```

Chapter 7: Lessons Learned

Ashutosh

There are two major takeaways I have from this project. On the technical side, it's really empowering to have an idea of how every feature in a language goes from idea to fully functioning: not identifying it during scanning, getting conflicts during parser, breaking during semantic analysis, and creating LLVM errors and broken modules. On a more serious note, it's really cool to understand how much underlies something as an array; you might think it's just a block of memory, but you can't appreciate it without understanding how to make it interact with other things within a language you design. I now have a much deeper appreciation for how well languages (even like Java) work

today and the fact that people much smarter than me have lost sleep over simple things that we take for granted in programming things today.

The other significant takeaway is about working on a team project. You can't build a good programming language within a semester alone, and it's really important to rely on teammates and their abilities. While not everybody is going to be good at the same things, it's important to keep an understanding of where certain people can shine. Furthermore, just walking through a problem with someone else makes not only your understanding of the problem deeper but also lets other team members get a more comprehensive idea of each component within the project, which is important when you and your team are working on the same issues. There's no "I" in team, and there's definitely no "I" in PLT, so working well within your team is key to getting the project working without going to hell and back.

Chang

The two most intimidating things about this project are understanding how the code could work in detail and figuring out how to implement new features that are different from all the previous work.

It was not an easy task to actually understand all the basic code provided. First of all, the OCaml syntax (hundreds of lines of the nested let statements) is not that easy to understand even after doing a homework assignment, and it is a demanding job to work with parser, AST, and the other files all at the same time to make the language compile. Once you understand the whole thing pretty well, the next big challenge is to figure out how to implement new features, especially those that are not similar to existing ones.

Due to lack of documentation of LLVM, there were many things that we thought could work very well logically but had no idea how to implement in LLVM, and there were also things that should have worked but failed because of some LLVM features.
In short, time is the ultimate limit on this project, because with the elusive OCaml syntax and lack of LLVM documentation, it simply takes lots of time to get used to the syntax and try to implement new things with several approaches that could logically work in LLVM.

In addition, there are also many other things to worry about for this project such as group work, writing reports, preparing for the demo, etc. It is a BIG project, so it is very important to keep up the hard work with the group throughout the entire semester.

Craig

The project introduced me to a side of Computer Science that I had never seen before; I got to interact with the entire process of compilation to build the compiler for our own language. I find it invaluable that I can now describe the typical process of compilation from lexical analysis to idealized machine code. Needless to say, with the successes came many failures and lessons to be learned as a result.

One such lesson that I learned was to plan ahead. Although we did plan meetings every week, we had a noticeable lack of trajectory that led us to aimlessly work on some feature that had not yet been completed. On the surface, there does not appear to be too much wrong with that, but retrospectively, it has certainly slowed us down.

Throughout the project, I learned the importance of team meetings and the utility of pair programming. At times, we opted to work remotely instead of physically meeting. This led to some miscommunications about what was currently done and what needed to be worked on and, later, led to some very unfortunate merge conflicts. Physically meeting gave us the opportunity to "pair program" on difficult tasks by assigning two people to work on one problem together. Pair programming afforded us success on some problem areas like implementing different types and using C bindings.

Overall, I feel that I learned a great deal throughout this project that can be applied to areas far beyond Computer Science.

William

Working on this project showed me a lot about the intricacies of building a programming language and about working on teams to build "products."At the start of the project, we had some fairly lofty goals for what this language would be able to do and by when we would have each featured finished. After a few iterations, it became clear that the velocity at which we would be able to build features would be slower than we initially planned due to our workloads and the time it took to fully understand how to solve the technical problems we faced. This meant that by the end, we had to look at our product backlog and be much more selective in choosing which features to work on. While this may have been disappointing at first, it showed us that there is always a balance between the goals you have for a product versus the core features needed within a limited development window.

Moreover, our group learned a lot about the usefulness of doing pair programming. By having two people working on a feature, it made sure both of them had a good understanding of how their feature worked in the context of the entire language, but also made it easy for others to find help when needed if one of the two people was unavailable to talk. It also helped speed up building features because two brains are always better than one.

Lastly, make sure to plan and do work early. This project will take much longer than you expect to complete and you will get frustrated often. If you think your team is behind, be more proactive in pushing the group to meet more and work on debugging sooner to avoid being overloaded with work in the later half of the semester.

Chapter 8: Code Listing

A full copy of the code written for DCL can be found below. Since each team member worked on all the files, we have not added individual sign offs.

DCL.ml

```
(* Top-level of the DCL compiler: scan & parse the input,
   check the resulting AST, generate LLVM IR, and dump the module *)

type action = Ast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);  (* Print the AST only *)
                              ("-l", LLVM_IR);  (* Generate LLVM, don't check
*)
                              ("-c", Compile) ] (* Generate, check LLVM IR *)
  else Compile in
```

```
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  Semant.check ast;
  match action with
    Ast -> print_string (Ast.string_of_program ast)
  | LLVM_IR -> print_string (Llvm.string_of_llmodule
(Codegen.translate ast))
  | Compile -> let m = Codegen.translate ast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

Scanner.mll

```
(* Ocamllex scanner for DCL *)


{ open Parser }


let exponent_rule = ('e' | 'E') ('+' | '-')? ['0'-'9']+
let float_rule =              '.'['0'-'9']+ exponent_rule? |
                 ['0'-'9']+ ('.'['0'-'9']* exponent_rule? |
                                            exponent_rule)
```

```
let string_rule = ('\'' | '\"')
                  ([' '-'!'] |
                   ['#'-'&'] |
                   ['('-']'] |
                   [']'-'~'] |
                      "\\\\" |
                       "\\r" |
                       "\\n" |
                       "\\t" |
                      "\\\'" |
                      "\\\""  )*
                  ('\'' | '\"')
```

```
rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)

(* Brackets and Punctuation *)
| '('      { LPAREN }
| ')'      { RPAREN }
```

```
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }


(* Mathematical Operators *)
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '^'      { EXPONT }
| '='      { ASSIGN }


(*Equality Operators *)
| "=="     { EQ }
| "!="     { NEQ }
| "<"      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }


(* Logical Operators *)
```

```
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "~"      { TILDE }

(* Conditional Operators *)
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }


(* Keywords for functions *)
| "return" { RETURN }
| "buteverytime" { BUTEVERYTIME }

(* Keywords for Data Types *)
| "int"    { INT }
| "double" { DOUBLE }
| "string" { STRING }
| "void"   { VOID }
```

```
| "["      { LSQUARE }
| "]"      { RSQUARE }
| ","      { COMMA }
| "{|"     { LINDEX }
| "|}"     { RINDEX }
| "of"     { OF }
| "#"      { LENGTH }
| ['0'-'9']+ as lxm { INTLITERAL(int_of_string lxm) }
| float_rule as lxm { DBLLITERAL(float_of_string lxm) }
| string_rule as lxm { STRLITERAL(let rec int_range = function
                                    0 -> [ ]
                                  |  1 -> [ 0 ]
                                  | n -> int_range (n - 1) @ [ n - 1
] in
                                 let rec glob = function
                                   | '\\' :: 'n' :: rest -> '\n' ::
(glob rest)
                                   | '\\' :: 'r' :: rest -> '\r' ::
(glob rest)
                                   | '\\' :: 't' :: rest -> '\t' ::
(glob rest)
                                   | '\\' :: '\\' :: rest -> '\\' ::
(glob rest)
                                   | '\\' :: '"' :: rest -> '\"' ::
(glob rest)
```

```
                                    | '\\' :: '\'' :: rest -> '\'' ::
(glob rest)

                                    | x :: rest ->  x :: (glob rest)

                                    | [] -> [] in

                                   let char_cleaned = glob (List.map
(fun x -> lxm.[x]) (int_range (String.length lxm))) in

                                   let cleaned = String.concat
"" (List.map (fun x -> String.make 1 x) char_cleaned) in

                                   let strlen = String.length cleaned
in

                                if strlen == 2 then "" else String.sub
cleaned 1 (strlen - 2)) }

| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }

| eof { EOF }

| _ as char { raise (Failure("illegal character " ^ Char.escaped
char)) }



and comment = parse

  "*/" { token lexbuf }

| _    { comment lexbuf }
```

Parser.mly

```
%{

open Ast

%}
```

```
/* Ocamlyacc parser for DCL */

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA

%token PLUS MINUS TIMES DIVIDE EXPONT ASSIGN NOT TILDE

%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR DOUBLE STRING
BUTEVERYTIME

%token RETURN IF ELSE FOR WHILE INT BOOL VOID LINDEX RINDEX

%token LSQUARE RSQUARE OF LENGTH

%token <int> INTLITERAL

%token <float> DBLLITERAL

%token <string> STRLITERAL

%token <string> ID

%token EOF



%nonassoc NOELSE

%nonassoc ELSE

%right ASSIGN

%left OR

%left AND

%left EQ NEQ

%left LT GT LEQ GEQ

%left PLUS MINUS

%left TIMES DIVIDE

%right EXPONT
```

```
%right NOT NEG LENGTH
%left LINDEX

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }

decls:
   /* nothing */ { [], [] }
 | decls globalstmt { ($2 :: fst $1), snd $1 }
 | decls bdecl { fst $2 :: fst $1, (snd $2 :: snd $1) }
 | decls fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
```

```
     { { typ = $1;
       fname = $2;
       formals = $4;
       body = List.rev $7 } }


bdecl:
   typ ID ASSIGN expr BUTEVERYTIME LPAREN expr RPAREN LBRACE stmt_list
RBRACE
     { (GlobalAssign($1, $2, $4), { typ = $1;
   fname = "__" ^ $2;
   formals = [($1, $2) ; ($1, "~" ^ $2)];
   body = let full_stmt_list = (List.rev $10) @ [ Return (Id($2)) ] in
          [ If($7, Block(full_stmt_list), Return (Id($2))) ]
(*(Id($2))*)
        })
     }


formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }


formal_list:
```

```
    typ ID                    { [($1,$2)] }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }


dtyp:
    INT { Int }
  | DOUBLE { Double }
  | STRING { String }


dim_list:
    LSQUARE RSQUARE  { 1 }
  | LSQUARE RSQUARE dim_list { 1 + $3 }


atyp:
    dtyp dim_list { Array($1, $2) }


typ:
    dtyp { Simple($1) }
  | VOID { Void }
  | atyp { $1 }


globalstmt_list:
```

```
    /* nothing */    { [] }
  | globalstmt_list globalstmt { $2 :: $1 }


stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI { Expr $1 }
  | RETURN SEMI { Return Noexpr }
  | RETURN expr SEMI { Return $2 }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | typ ID SEMI {Local($1, $2)}

globalstmt:
    typ ID SEMI { Global($1, $2) }
  | typ ID ASSIGN expr SEMI { GlobalAssign($1, $2, $4) }
```

```
expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

index:
    LINDEX expr RINDEX { $2 }

val_list:
    expr                { [ $1 ] }
  | expr COMMA val_list { [ $1 ] @ $3 }

simple_arr_literal:
    LSQUARE val_list RSQUARE { $2 }

expr:
    INTLITERAL       { IntLiteral($1) }
  | DBLLITERAL       { DblLiteral($1) }
  | STRLITERAL       { StrLiteral($1) }
  | simple_arr_literal { ArrLiteral($1) }
  | TILDE ID       { TildeOp($2) }
  | ID               { Id($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
```

```
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EXPONT expr { Binop($1, Exp,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr         { Unop(Not, $2) }
  | LENGTH expr      { Unop(Length, $2) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LSQUARE expr OF expr RSQUARE { DefaultArrLiteral($2, $4) }
  | ID LSQUARE expr RSQUARE ASSIGN expr { ArrayAssign($1, [$3], $6) }
  | expr index { Index($1, [$2]) }
/* | ID index ASSIGN expr  { Assign(Index(Id($1), $2), $4) } */
  | LPAREN expr RPAREN { $2 }
  | typ ID ASSIGN expr {LocalAssign($1, $2, $4)}
```

```
actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }


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

Semant.ml

```
(* Semantic checking for the DCL compiler *)


module A = Ast
open Ast
open Hashtbl
open Llvm


module StringMap = Map.Make(String)
let formals:(string, A.typ) Hashtbl.t = Hashtbl.create 100
let symbols:(string, A.typ) Hashtbl.t = Hashtbl.create 100
let globalsymbols:(string, A.typ) Hashtbl.t = Hashtbl.create 100
```

```
(* Semantic checking of a program. Returns void if successful,
   throws an exception if something is wrong.
   Check each global variable, then check each function *)

let check (globals, functions) =

  (* Raise an exception if the given list has a duplicate *)
  let report_duplicate exceptf list =
    let rec helper = function
  n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
      | _ :: t -> helper t
      | [] -> ()
    in helper (List.sort compare list)
  in

  (* Raise an exception if a given binding is to a void type *)
  let check_not_void exceptf = function
      (Void, n) -> raise (Failure (exceptf n))
    | _ -> ()
  in
```

```
  let report_local_duplicate exceptf s =
      if Hashtbl.mem symbols s then raise (Failure (exceptf s));
  in


  let report_global_duplicate exceptf s =
      if Hashtbl.mem globalsymbols s then raise (Failure (exceptf s));
  in


  let check_var_void exceptf t n =
      if t == Void then raise (Failure (exceptf n))
  in


  (* Raise an exception of the given rvalue type cannot be assigned to
     the given lvalue type *)
  let check_type lvaluet rvaluet err =
     (*let _ = print_endline (string_of_typ lvaluet) in
     let _ = print_endline (string_of_typ rvaluet) in
     let _ = print_endline (string_of_bool(string_of_typ lvaluet ==
string_of_typ rvaluet)) in
     let _ = print_int (String.length (string_of_typ lvaluet)) in
     let _ = print_int (String.length (string_of_typ rvaluet)) in
```

```
    let _ = print_int (String.compare (string_of_typ lvaluet)
(string_of_typ rvaluet)) in*)



    (* See if = could be used :O *)

    if (String.compare (string_of_typ lvaluet) (string_of_typ
rvaluet)) == 0 then lvaluet else raise err

  in


  (**** Checking Global Variables ****)



   let type_of_identifier s =

      try Hashtbl.find symbols s

      with Not_found -> try Hashtbl.find formals s

                        with Not_found -> try Hashtbl.find
globalsymbols s

                                            with Not_found -> raise
(Failure ("undeclared identifier " ^ s ))

    in



    (* Return the type of an expression or throw an exception *)



  (**** Checking Functions ****)
```

```
  if List.mem "print" (List.map (fun fd -> fd.fname) functions)
  then raise (Failure ("function print may not be defined")) else ();

  if List.mem "print_line" (List.map (fun fd -> fd.fname) functions)
  then raise (Failure ("function print_line may not be defined")) else
();

  if List.mem "read" (List.map (fun fd -> fd.fname) functions)
  then raise (Failure ("function read may not be defined")) else ();

  if List.mem "write" (List.map (fun fd -> fd.fname) functions)
  then raise (Failure ("function write may not be defined")) else ();

  report_duplicate (fun n -> "duplicate function " ^ n)
    (List.map (fun fd -> fd.fname) functions);

  (* Function declaration for a named function *)
  let built_in_decls =
    StringMap.add "read"
    { typ = A.Simple(A.String); fname = "read"; formals =
[(Simple(String), "file_name")]; body = [] }
    (StringMap.singleton "write"
```

```
    { typ = A.Simple(A.Int); fname = "write"; formals =
[(Simple(String), "file_name") ; (Simple(String), "string_to_write")];
body = [] }) in
```

```
  let function_decls = List.fold_left (fun m fd -> StringMap.add
fd.fname fd m)
                          built_in_decls functions
  in

  let function_decl s = try StringMap.find s function_decls
       with
       Not_found -> (let _ = print_string s in raise (Failure
("unrecognized function " ^ s)))
  in

  let _ = function_decl "main" in (* Ensure "main" is defined *)

  let check_function func =

    List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^
```

```
      " in " ^ func.fname)) func.formals;



    report_duplicate (fun n -> "duplicate variable " ^ n ^ " in " ^
func.fname)

      (List.map snd func.formals

      );



    let symbol = List.iter (fun (t, n) -> Hashtbl.add formals n t )

  func.formals

    in



    let rec expr = function

         IntLiteral _ -> Simple(Int)

      | DblLiteral _ -> Simple(Double)

      | StrLiteral _ -> Simple(String)

      | ArrLiteral(l) -> let first_type = expr (List.hd l) in

                         let _ = (match first_type with

                                    Simple _ -> ()

                                  | _ -> raise (Failure ("'" ^
string_of_expr (List.hd l) ^ "' is not simple and is in array"))

                                  ) in

                         let _ = List.iter (fun x -> if
string_of_typ(expr x) == string_of_typ first_type then ()
```

```
                                                    else raise
(Failure ("'" ^ string_of_expr x ^ "' doesn't match array's type"))) l
in

                        Array((match first_type with Simple(x) -> x),
1)

      | DefaultArrLiteral(e1, e2) -> if string_of_typ (expr e1) ==
string_of_typ(Simple(Int))

                                    then (match expr e2 with

                                            Simple(t) -> Array(t,
1)

                                          | _ -> raise (Failure
("'" ^ string_of_expr e2 ^ "' is not a simple type")))

                                    else raise (Failure ("'" ^
string_of_expr e1 ^ "' is not an integer"))

      | Index(a, i) -> if string_of_typ(expr (List.hd i)) !=
string_of_typ(Simple(Int))

                        then raise ( Failure("Array index ('" ^
string_of_expr (List.hd i) ^ "') is not an integer") )

                        else

                          let type_of_entity = expr a in

                          (match type_of_entity with

                             Array(d, _) -> Simple(d)

                           | Simple(String) -> Simple(String)

                           | _ -> raise (Failure ("Entity being indexed
('" ^ string_of_expr a ^"') cannot be array")))

      | TildeOp(e) as ex -> type_of_identifier e

      | Id s -> type_of_identifier s

      | Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
```

```
      (match op with
        Equal | Neq when t1 = t2 -> Simple(Int)
        |  Add | Sub | Mult | Div when t1 = Simple(Int) && t2 =
Simple(Int) -> Simple(Int)
            | Less | Leq | Greater | Geq when t1 = Simple(Int) && t2 =
Simple(Int) -> Simple(Int)
            | And  | Or when t1 = Simple(Int) && t2 = Simple(Int) ->
Simple(Int)
        | Exp when t1 = Simple(Int) && t2 = Simple(Int) ->
Simple(Double)
        | Add | Sub | Mult | Div | Exp when t1 = Simple(Double) && t2
= Simple(Double) -> Simple(Double)
        | Less | Leq | Greater | Geq
        when t1 = Simple(Double) && t2 = Simple(Double) -> Simple(Int)



        | Add when t1 = Simple(String) && t2 = Simple(String) ->
Simple(String)
        | Less | Leq | Greater | Geq when t1 = Simple(String) && t2 =
Simple(String) -> Simple(Int)


        | _ -> raise (Failure ("illegal binary operator " ^
              string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
              string_of_typ t2 ^ " in " ^ string_of_expr e))
        )
      | Unop(op, e) as ex -> let t = expr e in
   (match op with
```

```
          Neg when t = Simple(Int) -> Simple(Int)
        | Not when t = Simple(Int) -> Simple(Int)
    | Neg when t = Simple(Double) -> Simple(Double)
    | Length when t = Simple(String) -> Simple(Int)
    | Length when t = Array(Double, 1) -> Simple(Int)
    | Length when t = Array(String, 1) -> Simple(Int)
    | Length when t = Array(Int, 1) -> Simple(Int)
    | _ -> raise (Failure ("illegal unary operator " ^ string_of_uop op
^
                    string_of_typ t ^ " in " ^ string_of_expr ex)))
      | Noexpr -> Void
      | Assign(var, e) as ex -> let lt = type_of_identifier var
                                and rt = expr e in
        check_type lt rt (Failure ("illegal assignment " ^
string_of_typ lt ^
              " = " ^ string_of_typ rt ^ " in " ^
              string_of_expr ex))
       | ArrayAssign(v, i, e) as ex -> let type_of_left_side =
                                       if string_of_typ(expr (List.hd
i)) != string_of_typ(Simple(Int))
                                       then raise ( Failure("Array
index ('" ^ string_of_expr (List.hd i) ^ "') is not an integer") )
                                       else
                                         let type_of_entity =
type_of_identifier v in
```

```
                                        (match type_of_entity with
                                           Array(d, _) -> Simple(d)
                                         | _ -> raise (Failure
("Entity being indexed ('" ^ v ^"') cannot be array"))) in
                                      let type_of_right_side = expr e
in
                                      check_type type_of_left_side
type_of_right_side
                                      (Failure ("illegal assignment "
^ string_of_typ type_of_left_side ^
                                                " = " ^ string_of_typ
type_of_right_side ^ " in " ^
                                                string_of_expr ex))
      | LocalAssign (t, s, e) as ex -> check_var_void (fun n ->
"illegal void local " ^ n ^
      " in " ^ func.fname) t s; report_local_duplicate (fun n ->
"duplicate local " ^ n ^ " in " ^ func.fname) s;
      let lt = t and rt = expr e in
        check_type lt rt (Failure ("illegal assignment " ^
string_of_typ lt ^
             " = " ^ string_of_typ rt ^ " in " ^
             string_of_expr ex)); Hashtbl.add symbols s t; t
      | Call(fname, actuals) as call ->
         if fname = "print" || fname = "print_line"
         then (if List.length actuals == 1
               then let arg_type = string_of_typ (expr (List.hd
actuals)) in
                    if arg_type = string_of_typ (Simple(Int)) ||
```

```
                    arg_type = string_of_typ (Simple(Double)) ||
                    arg_type = string_of_typ (Simple(String))
                  then Void
                  else raise (Failure ("illegal actual argument
found " ^ string_of_typ (expr (List.hd actuals)) ^
                                                    " in " ^
string_of_expr (List.hd actuals)))
              else raise (Failure ("expecting 1 argument in " ^
string_of_expr call)))
        else (let fd = function_decl fname in
              if List.length actuals != List.length fd.formals then
                raise (Failure ("expecting " ^ string_of_int
                  (List.length fd.formals) ^ " arguments in " ^
string_of_expr call))
              else List.iter2 (fun (ft, _) e -> let et = expr e in
                    ignore (check_type ft et
                      (Failure ("illegal actual argument found " ^
string_of_typ et ^
                      " expected " ^ string_of_typ ft ^ " in " ^
string_of_expr e))))
                  fd.formals actuals;
              fd.typ)
    in


    let check_int_expr e = if string_of_typ (expr e) != string_of_typ
(Simple(Int))
```

```
     then raise (Failure ("expected int expression in " ^
string_of_expr e))

     else () in




    let globalstmt = function

       Global(t,s) as ex -> check_var_void (fun n -> "illegal void
global " ^ n) t s;

       report_global_duplicate (fun n -> "duplicate global " ^ n) s;

       Hashtbl.add globalsymbols s t;

     | GlobalAssign(t,s,e) as ex -> check_var_void (fun n -> "illegal
void global " ^ n) t s;

     report_global_duplicate (fun n -> "duplicate global " ^ n) s;

     let lt = t and rt = expr e in

        check_type lt rt (Failure ("illegal global assignment " ^
string_of_typ lt ^

             " = " ^ string_of_typ rt ^ " in " ^

             string_of_globalstmt ex)); Hashtbl.add globalsymbols s t
in

        Hashtbl.clear globalsymbols; let globalvars = List.map
globalstmt globals in




    (* Verify a statement or throw an exception *)

    let rec stmt = function

  Block sl -> let rec check_block = function
```

```
          [Return _ as s] -> stmt s
        | Return _ :: _ -> raise (Failure "nothing may follow a
return")
        | Block sl :: ss -> check_block (sl @ ss)
        | s :: ss -> stmt s ; check_block ss
        | [] -> ()
       in check_block sl
      | Expr e -> ignore (expr e)
      | Local (t, s) as ex -> check_var_void (fun n -> "illegal void
local " ^ n ^
      " in " ^ func.fname) t s; report_local_duplicate (fun n ->
"duplicate local " ^ n ^ " in " ^ func.fname) s;
      ignore(Hashtbl.add symbols s t);
      | Return e -> let t = expr e in if t = func.typ then () else
         raise (Failure ("return gives " ^ string_of_typ t ^ "
expected " ^
                         string_of_typ func.typ ^ " in " ^
string_of_expr e))

      | If(p, b1, b2) -> check_int_expr p; stmt b1; stmt b2
      | For(e1, e2, e3, st) -> ignore (expr e1); check_int_expr e2;
                               ignore (expr e3); stmt st
      | While(p, s) -> check_int_expr p; stmt s
    in
```

```
    stmt (Block func.body)


  in

  List.iter check_function functions
```

Ast.ml

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

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater
| Geq |

          And | Or  | Exp


type uop = Neg | Not | Tilde | Length


type dtyp = Int | Double | String

type typ = Simple of dtyp | Void | Array of dtyp * int

type bind = typ * string


(*type arr_literals =

    ArrLiteral of expr list

  | MultiArrLiteral of arr_literals list *)


type expr =

  (* arr_literals

  |*)

    IntLiteral of int

  | DblLiteral of float
```

```
  | StrLiteral of string
  | ArrLiteral of expr list
  | DefaultArrLiteral of expr * expr
  | Index of expr * expr list
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | TildeOp of string
  | Assign of string * expr
  | ArrayAssign of string * expr list * expr
  | Call of string * expr list
  | Noexpr
  | LocalAssign of typ * string * expr


type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Local of typ * string
```

```
type globalstmt =
    Global of typ * string
  | GlobalAssign of typ * string * expr

type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    body : stmt list;
  }

type program = globalstmt list * func_decl list

(* Pretty-printing functions *)
let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Exp -> "^"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
```

```
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"


let string_of_uop = function
    Neg -> "-"
  | Not -> "!"
  | Tilde -> "~"
  | Length -> "#"

let convert_array l conversion joiner =
    let glob_item original data = original ^ (conversion data) ^
joiner in
    let full = (List.fold_left glob_item "" l) in
    "[" ^ String.sub full 0 ((String.length full) - 2) ^ "]"

let string_of_d_typ = function
    Int -> "int"
  | Double -> "double"
  | String -> "string"

let rec repeat c = function
    0 -> ""
```

```
  | n -> c ^ (repeat c (n - 1))

let string_of_typ = function

   Void -> "void"

  |  Simple(d) -> string_of_d_typ d

  | Array(d, n) -> string_of_d_typ d ^ repeat "[]" n

let rec string_of_expr = function

    IntLiteral(l) -> string_of_int l

  | DblLiteral(l) -> string_of_float l

  | StrLiteral(l) -> "\"" ^ l ^ "\""

  | ArrLiteral(l) -> convert_array l string_of_expr ", "

(*  | MultiArrLiteral(l) -> convert_array l string_of_expr ",\n" *)

  | DefaultArrLiteral(e1, e2) -> "[" ^ string_of_expr e1 ^ " of " ^
string_of_expr e2 ^ "]"

  | Id(s) -> s

  | Index(e, l) -> string_of_expr e ^

                   (*convert_array l (fun e -> "[" ^ string_of_expr e
^ "]") ""*)

                   "{|" ^ string_of_expr (List.hd l) ^ "|}"

  | Binop(e1, o, e2) ->

      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr
e2

  | Unop(o, e) -> string_of_uop o ^ string_of_expr e

  | TildeOp(id) -> "~" ^ id
```

```
  | ArrayAssign(v, l, e) -> v ^ "[" ^ string_of_expr (List.hd l) ^ "]"
^ " = " ^ string_of_expr e

  | Assign(v, e) -> v ^ " = " ^ string_of_expr e

  | Call(f, el) ->

      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"

  | Noexpr -> ""

  | LocalAssign(t, s, e) -> string_of_typ t ^ " " ^ s ^ " = " ^
string_of_expr e



let rec string_of_stmt = function

    Block(stmts) ->

      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"

  | Expr(expr) -> string_of_expr expr ^ ";\n";

  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";

  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s

  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^

      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2

  | For(e1, e2, e3, s) ->

      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; "
^

      string_of_expr e3  ^ ") " ^ string_of_stmt s

  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
string_of_stmt s

  | Local(t, s) -> string_of_typ t ^ " " ^ s ^ ";\n"
```

```
let string_of_globalstmt = function

    Global(t,s) -> string_of_typ t ^ " " ^ s ^ ";\n"

  | GlobalAssign(t,s,e) -> string_of_typ t ^ " " ^ s ^ " = " ^
string_of_expr e ^ ";\n"



let string_of_fdecl fdecl =

  string_of_typ fdecl.typ ^ " " ^

  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals)
^

  ")\n{\n" ^

  String.concat "" (List.map string_of_stmt fdecl.body) ^

  "}\n"

let string_of_program (vars, funcs) =

  String.concat "" (List.map string_of_globalstmt vars) ^ "\n" ^

  String.concat "\n" (List.map string_of_fdecl funcs)
```

Codegen.ml

```
(* Code generation: translate takes a semantically checked AST and

produces LLVM IR



LLVM tutorial: Make sure to read the OCaml version of the tutorial



http://llvm.org/docs/tutorial/index.html
```

```
Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/

This is for DCL.
*)

module L = Llvm
module A = Ast

open Printf
open List
open Hashtbl
open Llvm

module StringMap = Map.Make(String)

let local_vars:(string, llvalue) Hashtbl.t = Hashtbl.create 100
let global_vars:(string, llvalue) Hashtbl.t = Hashtbl.create 100
```

```
let expr_store_local:(string, llvalue) Hashtbl.t = Hashtbl.create 100

let expr_store_global:(string, llvalue) Hashtbl.t = Hashtbl.create 100



let translate (globals, functions) =

  let context = L.global_context () in

  let the_module = L.create_module context "DCL" in

  let globalbuilder = builder context

  and i32_t  = L.i32_type    context

  and i8_t   = L.i8_type     context

  and f64_t  = L.double_type context

  and ptr_t = L.pointer_type (L.i8_type (context))

  and void_t = L.void_type   context in



  let rec int_range = function

      0 -> [ ]

    | 1 -> [ 0 ]

    | n -> int_range (n - 1) @ [ n - 1 ] in



  let rec ltype_of_typ = function

      A.Simple(A.Int) -> i32_t

    | A.Simple(A.String) -> L.struct_type context [| i32_t ;
L.pointer_type i8_t |]
```

```
    | A.Simple(A.Double) -> f64_t

    | A.Array(d, _) ->  L.struct_type context [| i32_t ;
L.pointer_type (ltype_of_typ (A.Simple(d))) |]

    | A.Void -> void_t in



  let default = function

    A.Simple(A.Int)    -> L.const_int          i32_t 0

  | A.Simple(A.Double) -> L.const_float        f64_t 0.

  | A.Simple(A.String) -> L.const_null (L.struct_type context [| i32_t
; L.pointer_type i8_t |])

  | A.Array(d, _)      -> L.const_null (L.struct_type context [| i32_t
; L.pointer_type (ltype_of_typ (A.Simple(d))) |]) in



  (* Declare each global variable; remember its value in a map *)

  let lookupglobal n = try Hashtbl.find global_vars n

                       with Not_found -> raise (Failure ("undeclared
id: " ^ n ))



    in



  (* Declare printf(), which the print built-in function will call *)
```

```
  let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t
|] in

  let printf_func = L.declare_function "printf" printf_t the_module in



  (* String concatenation *)

  let strcmp_t = L.function_type i32_t [| L.pointer_type i8_t ;
L.pointer_type i8_t |] in

  let strcmp_func = L.declare_function "strcmp" strcmp_t the_module in



  (* Exponent *)

  let expint_t = L.function_type f64_t [| i32_t ; i32_t |] in

  let expint_func = L.declare_function "__exp_int" expint_t the_module
in



  let expdbl_t = L.function_type f64_t [| f64_t ; f64_t |] in

  let expdbl_func = L.declare_function "__exp_dbl" expdbl_t the_module
in



  (* File I/O *)



  let read_t = L.function_type (ltype_of_typ (A.Simple(A.String))) [|
ltype_of_typ (A.Simple(A.String)) |] in

  let read_func = L.declare_function "read" read_t the_module in
```

```
  let write_t = L.function_type i32_t [| ltype_of_typ
(A.Simple(A.String)) ; ltype_of_typ (A.Simple(A.String)) |] in

  let write_func = L.declare_function "write" write_t the_module in



  (* Define each function (arguments and return type) so we can call
it *)

  let function_decls =

    let function_decl m fdecl =

      let name = fdecl.A.fname

      and formal_types =

  Array.of_list (List.map (fun (t,_) -> ltype_of_typ t)
fdecl.A.formals)

      in let ftype = L.function_type (ltype_of_typ fdecl.A.typ)
formal_types in

      StringMap.add name (L.define_function name ftype the_module,
fdecl) m in

    List.fold_left function_decl StringMap.empty functions in



  (* Fill in the body of the given function *)

  let build_function_body fdecl =

    Hashtbl.clear local_vars;

    let (the_function, _) = StringMap.find fdecl.A.fname
function_decls in

    let builder = L.builder_at_end context (L.entry_block
the_function) in
```

```
    let int_format_str = L.build_global_stringptr "%d" "fmt" builder
in

    let dbl_format_str = L.build_global_stringptr "%f" "fmt" builder
in

    let str_format_str = L.build_global_stringptr "%s" "fmt" builder
in

    let int_format_str_nl = L.build_global_stringptr "%d\n" "fmt"
builder in

    let dbl_format_str_nl = L.build_global_stringptr "%f\n" "fmt"
builder in

    let str_format_str_nl = L.build_global_stringptr "%s\n" "fmt"
builder in



    (* Construct the function's "locals": formal arguments and locally

       declared variables.  Allocate each on the stack, initialize
their

       value, if appropriate, and remember their values in the
"locals" map *)




    (* Return the value for a variable or formal argument *)

    let lookup n = try Hashtbl.find local_vars n

                   with Not_found -> try Hashtbl.find global_vars n

                                     with Not_found -> raise (Failure
("undeclared id: " ^ n ))
```

```
    in


    let findValue s = if Hashtbl.mem expr_store_local s then
Hashtbl.find expr_store_local s
                  else Hashtbl.find expr_store_global s



    in


    let build_string_from_code e' = let size = L.operand (L.size_of
(L.type_of e')) 1 in
                                    let dest = L.build_array_malloc
i8_t size "tmp" builder in
                                    List.iter (fun x ->
                                      let more = (L.build_gep dest  [|
L.const_int i32_t x |] "tmp2" builder) in
                                      let x = L.build_extractvalue e'
x "tmp2" builder in
                                      ignore (L.build_store x more
builder)
                                    ) (int_range ((match
(L.int64_of_const size) with Some i -> Int64.to_int i) - 1)) ;
                                    L.build_in_bounds_gep dest [|
L.const_int i32_t 0 |] "whatever" builder in
```

```
    let clean_up_string_stuff dest = L.build_free dest builder in

    (* Construct code for an expression; return its value *)

    let rec expr builder = function

        A.IntLiteral i -> L.const_int i32_t i

      | A.DblLiteral d -> L.const_float f64_t d

      | A.StrLiteral s -> let llvm_string = L.const_string context s
in

                          let size = String.length s in

                          let new_array = L.build_array_malloc i8_t
(L.const_int i32_t (size + 1)) "tmp" builder in

                          List.iter (fun x ->

                            let more = (L.build_gep new_array  [|
L.const_int i32_t x |] "tmp2" builder) in

                            let y = if x != size

                                    then L.build_extractvalue
llvm_string x "tmp2" builder

                                    else L.const_int i8_t 0 in

                            ignore (L.build_store y more builder)

                          ) (int_range (size + 1)) ;

                          let new_literal = L.build_malloc
(ltype_of_typ (A.Simple(A.String))) "arr_literal" builder in

                          let first_store = L.build_struct_gep
new_literal 0 "first" builder in

                          let second_store = L.build_struct_gep
new_literal 1 "second" builder in
```

```
                            let store_it = L.build_store (L.const_int
i32_t size) first_store builder in

                            let store_it_again = L.build_store new_array
second_store builder in

                            let actual_literal = L.build_load
new_literal "actual_arr_literal" builder in

                            actual_literal

      | A.Noexpr -> L.const_int i32_t 0

      | A.Id s -> L.build_load (lookup s) s builder

      | A.ArrLiteral(l) -> let size = L.const_int i32_t (List.length
l) in

                            let all = List.map (fun e -> expr builder
e) l in

                            let new_array = L.build_array_malloc
(L.type_of (List.hd all)) size "tmp" builder in

                            List.iter (fun x ->

                               let more = (L.build_gep new_array [|
L.const_int i32_t x |] "tmp2" builder) in

                               let intermediate = List.nth all x in

                               ignore (L.build_store intermediate more
builder)

                            ) (int_range (List.length l)) ;

                            let type_of_new_literal = L.struct_type
context [| i32_t ; L.pointer_type (L.type_of (List.hd all)) |] in

                            let new_literal = L.build_malloc
type_of_new_literal "arr_literal" builder in

                            let first_store = L.build_struct_gep
new_literal 0 "first" builder in
```

```
                            let second_store = L.build_struct_gep
new_literal 1 "second" builder in

                            let store_it = L.build_store size
first_store builder in

                            let store_it_again = L.build_store
new_array second_store builder in

                            let actual_literal = L.build_load
new_literal "actual_arr_literal" builder in

                            actual_literal

      | A.DefaultArrLiteral(e1, e2) -> let size = expr builder e1 in

                                       let first = expr builder e2 in

                                       let d = if L.type_of first ==
ltype_of_typ (A.Simple(A.Int))

                                               then A.Int

                                               else if L.type_of first
== ltype_of_typ (A.Simple(A.Double))

                                                    then A.Double

                                                    else A.String in

                                       let new_array =
L.build_array_malloc (ltype_of_typ (A.Simple(d))) size "tmp" builder
in

                                       let start = L.build_alloca
i32_t "start" builder in

                                       let store = L.build_store
(L.const_int i32_t 1) start builder in

                                       let first_pointer_in_array =
L.build_gep new_array [| (L.const_int i32_t 0) |] "gep_ptr_in_array"
builder in

                                       let first_store_in_array =
L.build_store first first_pointer_in_array builder in
```

```
                                        let position = L.block_parent
(L.insertion_block builder) in

                                        let continue_basic_block =
L.append_block context "continue" position in

                                        ignore (L.build_br
continue_basic_block builder) ;

                                        let iteration_basic_block =
L.append_block context "iterate" position in

                                        let end_builder =
L.builder_at_end context iteration_basic_block in

                                        let cur_value = L.build_load
start "cur_value" end_builder in

                                        let pointer_in_array =
L.build_gep new_array [| cur_value |] "gep_ptr_in_array" end_builder
in

                                        let store_in_array =
L.build_store (expr end_builder e2) pointer_in_array end_builder in

                                        let update = L.build_add
cur_value (L.const_int i32_t 1) "tmp" end_builder in

                                        let new_store = L.build_store
update start end_builder in

                                        let loop = L.build_br
continue_basic_block end_builder in

                                        L.block_terminator
(L.insertion_block end_builder) ;

                                        let continue_builder =
L.builder_at_end context continue_basic_block in

                                        let cur_value = L.build_load
start "cur_value" continue_builder in

                                        let continue_value =
L.build_icmp L.Icmp.Slt cur_value size "tmp" continue_builder in
```

```
                                        let merge_basic_block =
L.append_block context "merge" position in

                                        ignore (L.build_cond_br
continue_value iteration_basic_block merge_basic_block
continue_builder) ;

                                        L.builder_at_end context
merge_basic_block ;

                                        L.position_at_end
merge_basic_block builder ;

                                        let new_literal =
L.build_malloc (ltype_of_typ (A.Array(d, 1))) "arr_literal" builder in

                                        let first_store =
L.build_struct_gep new_literal 0 "first" builder in

                                        let second_store =
L.build_struct_gep new_literal 1 "second" builder in

                                        let store_it = L.build_store
size first_store builder in

                                        let store_it_again =
L.build_store new_array second_store builder in

                                        let actual_literal =
L.build_load new_literal "actual_arr_literal" builder in

                                        actual_literal

      | A.Index(a, i) -> let a' = expr builder a in

                         let i' = expr builder (List.hd i) in

                         let extract_array = L.build_extractvalue a' 1
"extract_ptr" builder in

                         let extract_value = L.build_gep extract_array
[| i' |] "extract_value" builder in

                         if L.type_of extract_array == L.pointer_type
i8_t
```

```
                        then let first_value = L.build_load
extract_value "value" builder in

                             let new_string = L.build_array_malloc
i8_t (L.const_int i32_t 2) "tmp" builder in

                             let more = L.build_gep new_string [|
L.const_int i32_t 0 |] "tmp2" builder in

                             let store_it = L.build_store first_value
more builder in

                             let more = L.build_gep new_string [|
L.const_int i32_t 1 |] "tmp2" builder in

                             let store_it_again = L.build_store
(L.const_int i8_t 0) more builder in

                             let new_literal = L.build_malloc
(ltype_of_typ (A.Simple(A.String))) "arr_literal" builder in

                             let first_store = L.build_struct_gep
new_literal 0 "first" builder in

                             let second_store = L.build_struct_gep
new_literal 1 "second" builder in

                             let store_it = L.build_store
(L.const_int i32_t 1) first_store builder in

                             let store_it_again = L.build_store
new_string second_store builder in

                             let actual_literal = L.build_load
new_literal "actual_arr_literal" builder in

                             actual_literal

                        else L.build_load extract_value "value"
builder

      | A.Binop (e1, op, e2) ->

    let e1' = expr builder e1
```

```
    and e2' = expr builder e2 in

    (match op with

      A.Add       -> if      L.type_of e1' == ltype_of_typ
(A.Simple(A.Int))    then L.build_add  e1' e2' "tmp" builder

                     else if L.type_of e1' == ltype_of_typ
(A.Simple(A.Double)) then L.build_fadd e1' e2' "tmp" builder

                     else

                          let first_size = L.build_extractvalue e1' 0
"extract_size" builder in

                          let second_size = L.build_extractvalue e2' 0
"extract_size" builder in

                          let first_array = L.build_extractvalue e1' 1
"extract_size" builder in

                          let second_array = L.build_extractvalue e2'
1 "extract_size" builder in

                          let total_size = L.build_add first_size
second_size "tmp" builder in

                          let new_array = L.build_array_malloc i8_t
(L.build_add total_size (L.const_int i32_t 1) "tmp" builder) "tmp"
builder in

                          let start = L.build_alloca i32_t "start"
builder in

                          let store = L.build_store (L.const_int i32_t
0) start builder in

                          let position = L.block_parent
(L.insertion_block builder) in

                          let continue_basic_block = L.append_block
context "continue" position in

                          ignore (L.build_br continue_basic_block
builder) ;
```

```
                        let iteration_basic_block = L.append_block
context "iterate" position in

                        let end_builder = L.builder_at_end context
iteration_basic_block in

                        let cur_value = L.build_load start
"cur_value" end_builder in

                        let pointer_in_array = L.build_gep new_array
[| cur_value |] "gep_ptr_in_array" end_builder in

                        let position_for_value = L.build_gep
first_array [| cur_value |] "gep_ptr_in_first_array" end_builder in

                        let value_to_store = L.build_load
position_for_value "tmp" end_builder in

                        let store_in_array = L.build_store
value_to_store pointer_in_array end_builder in

                        let update = L.build_add cur_value
(L.const_int i32_t 1) "tmp" end_builder in

                        let new_store = L.build_store update start
end_builder in

                        let loop = L.build_br continue_basic_block
end_builder in

                        L.block_terminator (L.insertion_block
end_builder) ;

                        let continue_builder = L.builder_at_end
context continue_basic_block in

                        let cur_value = L.build_load start
"cur_value" continue_builder in

                        let continue_value = L.build_icmp L.Icmp.Slt
cur_value first_size "tmp" continue_builder in

                        let merge_basic_block = L.append_block
context "merge" position in
```

```
                            ignore (L.build_cond_br continue_value
iteration_basic_block merge_basic_block continue_builder) ;

                            L.builder_at_end context merge_basic_block ;

                            L.position_at_end merge_basic_block
builder ;

                            let store = L.build_store (L.const_int i32_t
0) start builder in

                            let position = L.block_parent
(L.insertion_block builder) in

                            let continue_basic_block = L.append_block
context "continuetwo" position in

                            ignore (L.build_br continue_basic_block
builder) ;

                            let iteration_basic_block = L.append_block
context "iteratetwo" position in

                            let end_builder = L.builder_at_end context
iteration_basic_block in

                            let cur_value = L.build_load start
"cur_value" end_builder in

                            let pointer_in_array = L.build_gep new_array
[| (L.build_add cur_value first_size "tmp" end_builder) |]
"gep_ptr_in_array" end_builder in

                            let position_for_value = L.build_gep
second_array [| cur_value |] "gep_ptr_in_first_array" end_builder in

                            let value_to_store = L.build_load
position_for_value "tmp" end_builder in

                            let store_in_array = L.build_store
value_to_store pointer_in_array end_builder in

                            let update = L.build_add cur_value
(L.const_int i32_t 1) "tmp" end_builder in
```

```
                            let new_store = L.build_store update start
end_builder in

                            let loop = L.build_br continue_basic_block
end_builder in

                            L.block_terminator (L.insertion_block
end_builder) ;

                            let continue_builder = L.builder_at_end
context continue_basic_block in

                            let cur_value = L.build_load start
"cur_value" continue_builder in

                            let continue_value = L.build_icmp L.Icmp.Slt
cur_value second_size "tmp" continue_builder in

                            let merge_basic_block = L.append_block
context "mergetwo" position in

                            ignore (L.build_cond_br continue_value
iteration_basic_block merge_basic_block continue_builder) ;

                            L.builder_at_end context merge_basic_block ;

                            L.position_at_end merge_basic_block
builder ;

                            let pointer_in_array = L.build_gep new_array
[| (L.build_add first_size second_size "tmp" builder) |]
"gep_ptr_in_array" builder in

                            let store_in_array = L.build_store
(L.const_int i8_t 0) pointer_in_array builder in

                            let new_literal = L.build_malloc
(ltype_of_typ (A.Simple(A.String))) "str_literal" builder in

                            let first_store = L.build_struct_gep
new_literal 0 "first" builder in

                            let second_store = L.build_struct_gep
new_literal 1 "second" builder in
```

```
                        let store_it = L.build_store total_size
first_store builder in

                        let store_it_again = L.build_store new_array
second_store builder in

                        let actual_literal = L.build_load
new_literal "actual_str_literal" builder in

                        actual_literal

    | A.Sub       -> (if L.type_of e1' == ltype_of_typ
(A.Simple(A.Int)) then L.build_sub else L.build_fsub) e1' e2' "tmp"
builder

    | A.Mult      -> (if L.type_of e1' == ltype_of_typ
(A.Simple(A.Int)) then L.build_mul else L.build_fmul) e1' e2' "tmp"
builder

    | A.Div       -> (if L.type_of e1' == ltype_of_typ
(A.Simple(A.Int)) then L.build_sdiv else L.build_fdiv) e1' e2' "tmp"
builder

    | A.Exp       -> if   L.type_of e1' == ltype_of_typ
(A.Simple(A.Int))

                     then L.build_call expint_func [| e1' ; e2' |]
"tmp" builder

                     else L.build_call expdbl_func [| e1' ; e2' |]
"tmp" builder

    | A.And       -> L.build_and e1' e2' "tmp" builder

    | A.Or        -> L.build_or e1' e2' "tmp" builder

    | A.Equal     -> let result = (

                     if      L.type_of e1' == ltype_of_typ
(A.Simple(A.Int))    then L.build_icmp L.Icmp.Eq e1' e2' "tmp" builder

                     else if L.type_of e1' == ltype_of_typ
(A.Simple(A.Double)) then L.build_fcmp L.Fcmp.Oeq e1' e2' "tmp"
builder

                     else
```

```
                        let str_ptr_e1' = L.build_extractvalue e1' 1
"extract_char_array" builder in

                        let str_ptr_e2' = L.build_extractvalue e2' 1
"extract_char_array" builder in

                        let result = L.build_call strcmp_func [|
str_ptr_e1' ; str_ptr_e2' |] "tmp" builder in

                        L.build_icmp L.Icmp.Eq result (L.const_int
i32_t 0) "tmp" builder) in

                    L.build_mul (L.build_intcast result i32_t
"convert" builder) (L.const_int i32_t (-1)) "tmp" builder

    | A.Neq       -> let result = (

                    if      L.type_of e1' == ltype_of_typ
(A.Simple(A.Int))    then L.build_icmp L.Icmp.Ne e1' e2' "tmp" builder

                    else if L.type_of e1' == ltype_of_typ
(A.Simple(A.Double)) then L.build_fcmp L.Fcmp.One e1' e2' "tmp"
builder

                    else

                        let str_ptr_e1' = L.build_extractvalue e1' 1
"extract_char_array" builder in

                        let str_ptr_e2' = L.build_extractvalue e2' 1
"extract_char_array" builder in

                        let result = L.build_call strcmp_func [|
str_ptr_e1' ; str_ptr_e2' |] "tmp" builder in

                        let value = L.build_icmp L.Icmp.Ne result
(L.const_int i32_t 0) "tmp" builder in

                        value) in

                    L.build_mul (L.build_intcast result i32_t
"convert" builder) (L.const_int i32_t (-1)) "tmp" builder

    | A.Less      -> let result = (
```

```
                    if      L.type_of e1' == ltype_of_typ
(A.Simple(A.Int))   then    L.build_icmp L.Icmp.Slt e1' e2' "tmp"
builder

                    else if L.type_of e1' == ltype_of_typ
(A.Simple(A.Double)) then L.build_fcmp L.Fcmp.Olt e1' e2' "tmp"
builder

                    else

                      let str_ptr_e1' = L.build_extractvalue e1' 1
"extract_char_array" builder in

                      let str_ptr_e2' = L.build_extractvalue e2' 1
"extract_char_array" builder in

                      let result = L.build_call strcmp_func [|
str_ptr_e1' ; str_ptr_e2' |] "tmp" builder in

                      L.build_icmp L.Icmp.Slt result (L.const_int
i32_t 0) "tmp" builder) in

                    L.build_mul (L.build_intcast result i32_t
"convert" builder) (L.const_int i32_t (-1)) "tmp" builder

    | A.Leq       -> let result = (

                    if      L.type_of e1' == ltype_of_typ
(A.Simple(A.Int))   then L.build_icmp L.Icmp.Sle e1' e2' "tmp"
builder

                    else if L.type_of e1' == ltype_of_typ
(A.Simple(A.Double)) then L.build_fcmp L.Fcmp.Ole e1' e2' "tmp"
builder

                    else

                      let str_ptr_e1' = L.build_extractvalue e1' 1
"extract_char_array" builder in

                      let str_ptr_e2' = L.build_extractvalue e2' 1
"extract_char_array" builder in

                      let result = L.build_call strcmp_func [|
str_ptr_e1' ; str_ptr_e2' |] "tmp" builder in
```

```
                        L.build_icmp L.Icmp.Sle result (L.const_int
i32_t 0) "tmp" builder) in

                    L.build_mul (L.build_intcast result i32_t
"convert" builder) (L.const_int i32_t (-1)) "tmp" builder

    | A.Greater   -> let result = (

                    if      L.type_of e1' == ltype_of_typ
(A.Simple(A.Int))    then L.build_icmp L.Icmp.Sgt e1' e2' "tmp"
builder

                    else if L.type_of e1' == ltype_of_typ
(A.Simple(A.Double)) then L.build_fcmp L.Fcmp.Ogt e1' e2' "tmp"
builder

                    else

                      let str_ptr_e1' = L.build_extractvalue e1' 1
"extract_char_array" builder in

                      let str_ptr_e2' = L.build_extractvalue e2' 1
"extract_char_array" builder in

                      let result = L.build_call strcmp_func [|
str_ptr_e1' ; str_ptr_e2' |] "tmp" builder in

                      L.build_icmp L.Icmp.Sgt result (L.const_int
i32_t 0) "tmp" builder) in

                    L.build_mul (L.build_intcast result i32_t
"convert" builder) (L.const_int i32_t (-1)) "tmp" builder

    | A.Geq       -> let result = (

                    if      L.type_of e1' == ltype_of_typ
(A.Simple(A.Int))    then L.build_icmp L.Icmp.Sge e1' e2' "tmp"
builder

                    else if L.type_of e1' == ltype_of_typ
(A.Simple(A.Double)) then L.build_fcmp L.Fcmp.Oge e1' e2' "tmp"
builder

                    else
```

```
                        let str_ptr_e1' = L.build_extractvalue e1' 1
"extract_char_array" builder in

                        let str_ptr_e2' = L.build_extractvalue e2' 1
"extract_char_array" builder in

                        let result = L.build_call strcmp_func [|
str_ptr_e1' ; str_ptr_e2' |] "tmp" builder in

                        L.build_icmp L.Icmp.Sge result (L.const_int
i32_t 0) "tmp" builder) in

                    L.build_mul (L.build_intcast result i32_t
"convert" builder) (L.const_int i32_t (-1)) "tmp" builder

    )

      | A.TildeOp(id) -> let x = "~" ^ id in L.build_load (lookup (x))
x builder

      | A.Unop(op, e) ->

    let e' = expr builder e in

    (match op with

            A.Neg     -> if L.type_of e' == ltype_of_typ
(A.Simple(A.Int)) then L.build_neg e' "tmp" builder else L.build_fneg
e' "tmp" builder

          | A.Not     -> L.build_not e' "tmp" builder

          | A.Length  -> L.build_extractvalue e' 0 "extract_size"
builder)

      | A.Assign (s, e) -> let e' = expr builder e in

                    ignore (L.build_store e' (lookup s) builder);
ignore (Hashtbl.add expr_store_local s e'); e'

      | A.LocalAssign (t, s, e) -> let local_var = L.build_alloca
(ltype_of_typ t) s builder in

       Hashtbl.add local_vars s local_var;
```

```
      let e' = expr builder e in ignore (L.build_store e' local_var
builder); ignore (Hashtbl.add expr_store_local s e'); e'

      (* File I/O calls *)

      | A.Call ("read", [e]) -> let temp = expr builder e in

                                L.build_call read_func [| temp |]
"read" builder

      | A.Call ("write", [e1 ; e2]) -> let temp1 = expr builder e1 in

                                       let temp2 = expr builder e2 in

                                       L.build_call write_func [|
temp1 ; temp2 |] "write" builder

      (* https://www.ibm.com/developerworks/library/os-
createcompilerllvm1/ *)

      | A.ArrayAssign(v, i, e) -> let e' = expr builder e in

                                  let i' = expr builder (List.hd i) in

                                  let v' = L.build_load (lookup v) v
builder in

                                  let extract_array =
L.build_extractvalue v' 1 "extract_ptr" builder in

                                  let extract_value = L.build_gep
extract_array [| i' |] "extract_value" builder in

                                  ignore (L.build_store e'
extract_value builder); e'

      | A.Call("print", [e]) -> let e' = expr builder e in

                                if L.type_of e' == ltype_of_typ
(A.Simple(A.Int))

                                then L.build_call printf_func [|
int_format_str ; e' |] "print" builder

                                else if L.type_of e' == ltype_of_typ
(A.Simple(A.Double))
```

```
                                    then L.build_call printf_func [|
dbl_format_str ; e' |] "print" builder

                                    else L.build_call printf_func [|
str_format_str ; L.build_extractvalue e' 1 "extract_char_array"
builder |] "print" builder

      | A.Call("print_line", [e]) -> let e' = expr builder e in

                                    if L.type_of e' == ltype_of_typ
(A.Simple(A.Int))

                                    then L.build_call printf_func [|
int_format_str_nl ; e' |] "print" builder

                                    else if L.type_of e' ==
ltype_of_typ (A.Simple(A.Double))

                                         then L.build_call
printf_func [| dbl_format_str_nl ; e' |] "print" builder

                                         else L.build_call
printf_func [| str_format_str_nl ; L.build_extractvalue e' 1
"extract_char_array" builder |] "print" builder

      | A.Call (f, act) ->

         let (fdef, fdecl) = StringMap.find f function_decls in

   let actuals = List.rev (List.map (expr builder) (List.rev act)) in

   let result = (match fdecl.A.typ with A.Void -> ""

                                        | _ -> f ^ "_result") in

         L.build_call fdef (Array.of_list actuals) result builder

    in



    (* Invoke "f builder" if the current block doesn't already

       have a terminal (e.g., a branch). *)
```

```
    let add_terminal builder f =

      match L.block_terminator (L.insertion_block builder) with

  Some _ -> ()

      | None -> ignore (f builder) in



      let globalstmt = function

    A.Global(t,s) -> let global_var = L.build_alloca (ltype_of_typ t)
s builder in

       Hashtbl.add global_vars s global_var;

  | A.GlobalAssign(t,s,e) -> let global_var = L.build_alloca
(ltype_of_typ t) s builder in

       Hashtbl.add global_vars s global_var;

       let e' = expr builder e in

    ignore (L.build_store e' (lookup s) builder); ignore (Hashtbl.add
expr_store_global s e') in



      let globalvars = List.map globalstmt globals in



    let add_formal (t, n) p = L.set_value_name n p;

  let local = L.build_alloca (ltype_of_typ t) n builder in

  ignore(L.build_store p local builder);

  Hashtbl.add local_vars n local in
```

```
  let formals = ignore(List.iter2 add_formal fdecl.A.formals

  (Array.to_list (L.params the_function))) in


    (* Build the code for the given statement; return the builder for

       the statement's successor *)

    let rec stmt builder = function

  A.Block sl -> List.fold_left stmt builder sl

      | A.Expr e -> let tildes:(string, llvalue) Hashtbl.t =
Hashtbl.create 100 in

                    ignore (if String.sub fdecl.A.fname 0 2 = "__"
then () (* Don't generate calls to callback within a callback *)

                    else (

                      let callbackStrMap = StringMap.filter (let x k v
= (String.sub k 0 2) = "__" in x ) function_decls in

                      let callbackList = StringMap.bindings
callbackStrMap in

                      for j = 0 to ((List.length callbackList) - 1) do

                        let (key, (fdef, fdec)) = List.nth
callbackList j in

                        let newStr = String.create ((String.length
fdec.A.fname) - 2) in

                        let varName = ignore(for i = 0 to
((String.length newStr) - 1) do String.set newStr i (String.get
fdec.A.fname (i + 2)) done); newStr in

                        Hashtbl.add tildes ("~" ^ varName)
(L.build_load (lookup varName) varName builder)

                      done
```

```
                    ));

                    ignore (expr builder e);

                    ignore (if String.sub fdecl.A.fname 0 2 = "__"
then () (* Don't generate calls to callback within a callback *)

                    else (

                      let callbackStrMap = StringMap.filter (let x k v
= (String.sub k 0 2) = "__" in x ) function_decls in

                      let callbackList = StringMap.bindings
callbackStrMap in

                      for j = 0 to ((List.length callbackList) - 1) do

                        let (key, (fdef, fdec)) = List.nth
callbackList j in

                        let newStr = String.create ((String.length
fdec.A.fname) - 2) in

                        let varName = ignore(for i = 0 to
((String.length newStr) - 1) do String.set newStr i (String.get
fdec.A.fname (i + 2)) done); newStr in

                        let pass_in = L.build_load (lookup varName)
varName builder in

                        let result = "" in

                        let new_value = (L.build_call fdef [|
pass_in ; Hashtbl.find tildes ("~" ^ varName) |] result builder) in

                        L.build_store new_value (lookup varName)
builder

                      done

                    ));

                    builder

      | A.Return e -> ignore (match fdecl.A.typ with
```

```
    A.Void -> L.build_ret_void builder
  | _ -> L.build_ret (expr builder e) builder); builder
      | A.If (predicate, then_stmt, else_stmt) ->
        let bool_val = expr builder predicate in
        let bool_val = L.build_trunc bool_val (L.i1_type context)
"convert" builder in
   let merge_bb = L.append_block context "merge" the_function in

   let then_bb = L.append_block context "then" the_function in
   add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
     (L.build_br merge_bb);

   let else_bb = L.append_block context "else" the_function in
   add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
     (L.build_br merge_bb);

   ignore (L.build_cond_br bool_val then_bb else_bb builder);
   L.builder_at_end context merge_bb

      | A.While (predicate, body) ->
    let pred_bb = L.append_block context "while" the_function in
```

```
    ignore (L.build_br pred_bb builder);



    let body_bb = L.append_block context "while_body" the_function in

    add_terminal (stmt (L.builder_at_end context body_bb) body)

      (L.build_br pred_bb);



    let pred_builder = L.builder_at_end context pred_bb in

    let bool_val = expr pred_builder predicate in

    let bool_val = L.build_trunc bool_val (L.i1_type context)
"convert" pred_builder in



    let merge_bb = L.append_block context "merge" the_function in

    ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);

    L.builder_at_end context merge_bb

      | A.Local (t, s) ->

      ignore (let local_var = L.build_alloca (ltype_of_typ t) s
builder in

       Hashtbl.add local_vars s local_var); builder

      | A.For (e1, e2, e3, body) -> stmt builder

      ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ; A.Expr
e3]) ] )

    in
```

```
    (* Build the code for each statement in the function *)
    let builder = stmt builder (A.Block fdecl.A.body) in


    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.A.typ with
        A.Void -> L.build_ret_void
      | t -> L.build_ret (default fdecl.A.typ)) ;
  in


  List.iter build_function_body functions;
  the_module
```

Externalcalls.c

```
#include <math.h>

#include <stdio.h>

#include <string.h>
#include <stdlib.h>


struct string {

  int size;

  char *array;

};
```

```
int write(struct string out_file_name, struct string string_to_write)
{

  FILE *fp;

  fp = fopen(out_file_name.array, "wb");

  if(fp == NULL) {

    perror("failed to open file");

    return 0;

  }

  int chars_written = fwrite(string_to_write.array, 1,
string_to_write.size, fp);

  fclose(fp);

  return chars_written;

}

struct string read(struct string in_file_name) {

  FILE *fp;

  int len;

  char *buffer;

  fp = fopen(in_file_name.array, "rb");

  if(fp == NULL) {

    perror("failed to open file");

    return (struct string) {0, ""};

  }

  fseek(fp, 0, SEEK_END);

  len = ftell(fp);
```

```
  buffer = malloc(sizeof(char) * (len + 1));
  if(buffer == NULL) {
    perror("malloc failed");
    return (struct string) {0, ""};
  }
  fseek(fp, 0, SEEK_SET);
  fread(buffer, len + 1, sizeof(char), fp);
  fclose(fp);
  return (struct string) {len, buffer};
}


double __exp_int(int x, int y)
{
  return pow(x, y);
}

double __exp_dbl(double x, double y)
{
  return pow(x, y);
}

#ifdef BUILD_TEST
int main()
```

```
{
  printf("5 ^ -2 == %f\n", __exp_int(5, -2));
  printf("5 ^ 0.5 == %f\n", __exp_dbl(5.0, 0.5));
  return 0;
}
#endif
```