Manager: William Essilfie (wke2102)
Tester: Chang Liu (cl3403)
Language Guru: Ashutosh Nanda (an2655)
System Architect: Craig Rhodes (cdr2139)

DCL - Dynamic Callback Language

Goal:

We want to create an event-driven language called DCL that compiles down to LLVM. DCL is similar in syntax to Java but is special because of its buteverytime callback. This callback allows users to define particular blocks of code and then have them be run automatically in later execution of the language based on values at that future state. It is helpful for first time programmers: e.g., trying to understand where their variables are being updated; furthermore, the authors believe that the language makes it easier to encode logic for special cases by making that logic clear in definition code rather than later execution code.

Applications:

This programming language will have many useful applications. We think this language will be especially helpful for people new to programming because our unique callback technique will allow users to quickly see how their code is running, which will be helpful for when they are trying to debug their code. The language will also be helpful in use cases that either revolve around callbacks for events or specific actions for different cases. In addition, the language has a special tilde operator, which is useful because it allows users to track the original value a variable was set to versus what the value is changed to since the callback is called between normal lines of code; an example of this use case is shown below in one of our code samples.

Syntax:

Comments: /* */
Operators:
This language will support the following arithmetic operators:

- + (Numeric: addition, String: concatenation)
- - (Numeric: subtraction, String: undefined)
- * (Numeric: multiply, String: repeat string)
- / (Numeric: divide, String: split into pieces)
- ^ (Numeric: exponentiation, String: undefined)

Comparison Operators:
Strings will be compared by lexical ordering, and numeric values will take the usual ordering.

- >= (greater than or equal to)
- <= (less than or equal to)
- > (greater than)
- < (less than)
- == (compares two values and checks if they are structurally equal)
- != (not equal)

- | (boolean logic for integer values of 0 and 1; behavior undefined for other values)
- & (boolean logic for integer values of 0 and 1; behavior undefined for other values)

Primitive Types:
This language will support the following primitive types:

- Integer (int)
- Double (double)
- String (string)

Other Data Types:

- Arrays (must be all of times)

Keywords:

- butwhile
- butfor
- buteverytime (only possible when assigning to variable and does introspection between lines of code; every handler executes exactly once between lines of code)
- butthistime (affects current value of the expression; same as ternary if)

Special Operators:

- ~ (Allows user to access a variable's value before line of code was executed; only works within buteverytime callback blocks)

Code Sample:

```
/*this is a sample of a program in our language*/

/*this function takes in two numbers and performs division */
/*but everytime when y is 0, it prints out an error message without performing the division */

float div(int x, int y) {
        return x / y;
}
buteverytime (y == 0) {
        print("denominator cannot be 0");
}



/*this function takes in a number and returns its factorial */
/*but everytime when the number is 0 or 1, it returns 1; but everytime when the number is
negative, it prints out an error message */

int factorial(int n) {
        return n * factorial(n - 1);
}
buteverytime (n == 1 or n == 0) {
        return 1;
}
buteverytime (n < 0) {
      print("input should be a nonnegative number");
      return -1;
}
```

Tilde operator examples:

/* 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(i);
};
```

/* 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("a is 0 which is an illegal value for a");
};
```

/* 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("the value of a[2] is changed");
}
buteverytime (len(a) != ~len(a)) {
         print("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(i);
}
```

/* This function prints out the message “Hi Drake” only once because we are using buteverytime to actually change the increment variable x. */

```
void printDrake() {
        int x = 0 buteverytime (x < 10) {
          print(“Hi Drake”);
          x = x + 10;
        };
        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 < len(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 buteverytime ((~cost - cost) / (~cost butthistime (~cost == 0) 1) < 0.001) {
        stop = 1;
}

int stop = 0;
while(!stop) {
        cost = cost(x, y, intercept, slope);
}

print(“Slope: ” + slope)
print(“Intercept: ” + intercept)
```