DCL

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

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.

This language reference manual is organized in the following order:

Chapter 2 describes lexical conventions of DCL, including keywords, constants and comments.
Chapter 3 introduces the data types used in DCL, which are further divided into primitive types and reference types.
Chapter 4 presents the expressions and operators defined in DCL and how to use them.
Chapter 5 lists the several statements in DCL and how they can be used.
Chapter 6 gives several code examples written in DCL and shows how DCL can be very strong in various contexts.
Chapter 7 lists the references that have been used for this language reference manual.

Chapter 2: 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
   A
   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  butthistime
int    double    char    void    string
return
```

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, neither integer part nor fraction part can be missing from a double constant; either decimal point or the exponent part can be missing, but they cannot be missing together. Double constants following a negation operator - are negative floating-point numbers.

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        /*  illegal because both decimal point and exponent part are missing */
.5          /*  illegal because integer part is missing */
5.          /*  illegal because fraction part is missing */
```

Character Constant

Character constants are 1 or 2 characters inside single quotes. The two-character constants are only legal when the first one is a backslash. If the first character is a backslash and the second character is a non-alphabetic character, the character constant is simply the second character.

Examples:

```
‘a’     /* character a */
‘ab’    /* illegal */
‘\n’    /* newline character */
‘\t’    /* tab character */
‘\?’    /* character ? */
‘\’’    /* single quote ‘ */
‘\”’    /* double quote “ */
```

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 */
```

Chapter 3: Types

Primitive Data 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:
<scope> int

```
double <function_name>(double <parameters> ) {
        <code placed here>
        return <double value>;
}
```
One-line example
double x = 1.0;

Char (char)

A character is a single character that must be written within quotation marks. Each character variable can hold 8 bits (1 byte). For storing values larger than this, see the section on string literals.
An example of working with characters in the correct DCL syntax is as follows:

```
char <function_name>(char <parameters> ) {
        <code placed here>
        return <char value>;
}
```
One-line example:
char x = ‘h’;

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>
}
```

Non-Primitive Data Types

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.

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

Sample line for creating an empty array:
double purple[3];

Sample line for creating an empty array with variables:
double purple[3] = [3.0,4.0,5.0];

If you choose to initialize an array with values, you must fill the array. If the programmer fails to do so, DCL will not compile the program.
double purple[3] = [3.0,4.0]; /* this code will cause the compiler to print an error and the program will not compile */

Casting:
There is no current syntax for casting in DCL.

Chapter 4: 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 (double 1780) */
100.0/20.0 /* returns a double 5 (double 5) */
```

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(“hello”) /* evaluates/returns null */
```

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 */
```

Operators

Operators are used in conjunctions with operators. 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 ogie = “I love you like a table”;
int PierreAndNatasha = 1800 + 12;
1 = 6 /* invalid */
“dearEvanHansen” = “phantomOfTheOpera” /* 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;

/* 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. Since DCL does not have a boolean type, it will return 0 if the result evaluates to false and 1 if it evaluates to 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 1 (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 boolean operators (and, or, and not).

Tilde Operator

DCL has a special tilde operator denoted with “~”. Tildes work within buteverytime blocks where it lets you define a global callback as defined in Section 7. In essence this line of code will be executed every time an iteration occurs. This set of statements will be evaluated after every single line of source code. Examples of the tilde operator can be found in Section 7.

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

Chapter 5: 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.

butthistime Statement

butthistime allows a block of code to be executed or not depending on whether the boolean value of the condition evaluates to true. The structure is the condition, the block of code to be executed when the condition is true, and optionally the block of code to be executed when the condition is false.

```
/* Succeeding butthistime condition */

butthistime (1) {
	print(“Succeeds!”); /* “Succeeds” will be written to standard output */
}

/* Failing butthistime condition */

butthistime (0) {
	print(“Fails!”); /* “Fails” will not be written to standard output */
}

/* Conditional butthistime execution */

butthistime (condition) {
	print(“Condition was truthful”);
} otherwise {
	print(“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 butthistime.

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(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 in a boolean sense, 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;
}
```

Chapter 6: 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 itself and other variables; these are introduced by the buteverytime keyword. An example should prove useful.

```
int i = 0 buteverytime (i == 0) {
        print("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. This results in a pretty powerful construct as demonstrated by the next example.

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

This set of code will print the integers from 0 to 9 separated by newline; it's amazing how buteverytime has replicated the functionality of a for statement!

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(i);
        if (i < 10) {
                i = i + 1;
        }
};
```

(N.B.: 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.

Chapter 7: Code Examples

/*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)
```

Chapter 8: Language Grammar

S -> F | D | D F | F D
D (declaration) -> L; | L = R;
L (L-value) -> T v
R (R-value) -> i | x | c | s
B (buteverytime statement) -> L = R buteverytime (condition) { D* };
F (function) -> T v (A) { D* }
I (if statement) -> if (condition) { D* }
W (while loop) -> while (condition { D* }
Bw (butwhile) -> L = R while (condition) { D* }
Bf (butfor) -> L = R for (condition) { D* }
i (integer values) -> d
x (double values) -> d.d
c (char values) -> 'y'
s (string values) -> "yy" | no character
F (function) -> T v (A) { S }
T (type) -> int | double | char | string | void
A (args) -> L | A, A | no character
w (word) -> u | ww
d (digits) -> z | dd
v (variable name) -> w | wd
y (a character) -> z | u
z (a digit) -> a single digit from 0-9
u (a letter) -> a lowercase/uppercase letter from the English alphabet