Cornucopia User Manual

James Champ

Version: 0.8

17 May 2016

Contents

Introduction

This document will familiarize you with the Cornucopia programming language. You will learn about the language's syntax, data types, conditional structures, and the types of errors that you may encounter while compiling a Cornucopia program. Instructions on running the compiler are also included in this document.

Data Types

Cornucopia has two data types: `int`, which represents an integer, and `char`, which represents a single character. The language is strongly typed, so the compiler will throw an error if any type of operation is performed on two variables of different types.

Variables

Cornucopia variables must be declared before they can be used in a program. Program declarations must appear at the top of the main function, before any statements. Declarations have the following form:

```
var <identifier list> : <type>;
```

The identifier list can contain any number of identifiers. If more than one identifier is in a list, they must be separated by commas. Identifiers must start with a letter, and can contain only uppercase letters, underscores, and digits. Only the first eight characters of a variable are significant. This means that `identifier` and `identifiers` will be treated as the same variable by the compiler.

The type can be one of two things: `int` or `char`. Types must be in lowercase. Each declaration identifier on a declaration line will share the same data type. The following are valid declarations:

```
var INT1, INT2, INT3 : int;

var CHAR1 : char;

var REALLY_LONG_NAME, A : int;
```

Note that REALLY_LONG_NAME will be treated as REALLY_L by the compiler. This is important because if a variable is declared twice, the compiler will throw an error and shutdown. It should also be noted that Cornucopia variables all have global scope.

Statements

All statements must appear after the program's declarations. The first set of statements in a program is a compound statement, which is a list of statements surrounded by curly braces. A list of statements can be comprised of a single statement or a semicolon separated series of statements. There are a total of six types of statements:

Assignment Statements

Assignment statements are used to set a variable's value. These statements take the following form:

```
<id> = <expression>
```

`id` represents a previously declared identifier. Expressions will be covered in a later section.

Compound Statements

As previously mentioned, a compound statement is a list of statements within a set of braces.

Get Statements

Get statements are used to assign a user-inputted value to a variable. The variable must be declared before it is used. These statements take the following form:

```
get( <id> )
```

Put Statements

Put statements print the value of a variable to the console. They are similar to get statements in their form:

```
put( <id> )
```

The variable must be declared and assigned a value before it can be used in a put statement.

If Statements

If statements are the only conditional statement in the Cornucopia language. They take the following form:

```
if( <expression> ) <statement>
```

If the expression evaluates to true, the statement will be executed. Otherwise, the code appearing after the given statement will be executed. While there are no else statements in Cornucopia, the programmer can simulate else statements by using a flag and two sequential if statements.

If an expression contains a single integer value, the expression will evaluate to true if and only if that given value is equal to one. Expressions containing single character values cannot be evaluated, and will cause the compiler to throw an error.

Operators

Cornucopia has three main types of operators: relational operators, addition operators, and multiplication operators. Operators are used to perform mathematical operations on two are more variables or literal values.

Relational Operators

Relational operators test relations between two integer values. If the relation is true, a relational expression will evaluate to 1. Otherwise, the expression is false and will evaluate to 0. There are six kinds of relational operators:

`<value1> > <value2>`	Determines if `value1` is greater than `value2`.
`<value1> >= <value2>`	Determines if `value1` is greater than or equal to `value2`.
`<value1> == <value2>`	Determines if both values are equal.
`<value1> <= <value2>`	Determines if `value1` is less than or equal to `value2`.
`<value1> < <value2>`	Determines if `value1` is less than `value2`.
`<value1> <> <value2>`	Determines if both values are not equal to each other.

Addition Operators

Addition operators perform either addition or subtraction on two integer values and evaluate to the result of the operation. They take the following form:

`<value1> + <value2>`	Evaluates to the sum of both values.
`<value1> - <value2>`	Evaluates to the difference of both values.

Multiplication Operators

These operators perform either the multiplication, division, or modulo operation on two integer values. They appear as follows:

`<value1> * <value2>`	Multiplies the two values and evaluates to the result.
`<value1> / <value2>`	Divides `value1` by `value2`.
`<value1> % <value2>`	Evaluates to the remainder of `value1` divided by `value2`.

Values cannot be divided or modded by zero. The compiler will throw an error if this ever occurs.

Order of Operations

In cases where several values are to be evaluated on a single line of code, the values will be evaluated in a specific order. Multiplication operations are evaluated first, followed by addition operations, and

then, lastly, relational operations. If multiple operations of the same type exist on a line of code, they are evaluated from left to right.

Expressions

Expressions are simply any number of factors that are separated by operators. There are five types of factors: integer literals, character literals, variables, `true`, and `false`. `true` has the integer value of one, and `false` has the integer value of zero. Character literals take the following form:

```
' <character> '
```

Any variables that are used in an expression must first be assigned a value. Factors of different types will not be evaluated together. In such a case, the compiler will throw an error.

Comments

If the compiler, while parsing, encounters `//` in a line of code, those two slashes and any other characters to the right of them on the line are ignored.

End of Program Symbol

Every Cornucopia program must end with the `$` symbol. If any characters exist after this symbol, the program will not compile.

Main Function Signature

The main function has the following signature:

```
<access> static void <id> ( <identifier list> )
```

`access` can be either `public` or `private`, either one has no bearing on how the compiler will operate. `id` is an identifier that will be used to name the file containing the compiled code. The identifiers found in the identifier list cannot be used by a Cornucopia program. It is recommended that the programmer leave this list blank. Future versions will provide support for command line arguments.

Running the Compiler

The Cornucopia compiler is packaged as a runnable jar file. Your computer will need to have Java installed to be able to run the Cornucopia compiler. To compile a Cornucopia program, navigate to the folder containing cornucopia.jar from the command line and run the following command:

```
java –jar cornucopia.jar <source file location>
```

The source file location must be the full file path of the source code. If the program is successfully compiled, a .corn file will be created in the same folder as the source code file. The .corn file will be have the same name as the main function, only it will be in lowercase.

Appendix A: Compiler Errors

Illegal Expression Error:	This is thrown when a program tries to evaluate a single character in an if statement
Missing File Error:	This error occurs when the user forgets to pass an input file location as a parameter when they start the compiler.
Type Mismatch Error:	Occurs when a mathematical operation is performed on an integer and a character.
Uninitialized Variable Error:	An error that occurs when a variable is used in an expression or put statement without first being assigned a value.
Undeclared Variable Error:	This error occurs when an undeclared variable is used in a statement.
Variable Declared Twice Error:	Occurs when the same variable is declared more than once.
Relational Assignment Error:	Occurs when a relational operator is used to assign a value to a variable.
Character Multiplication Error:	Occurs when any value is multiplied by a character.
Character Addition Error:	Occurs when any value is added to or subtracted by/from a character.
Divide By Zero Error:	Thrown when a value is divided or modded by zero.
Syntax Error:	Thrown when a program is not syntactically correct. The compiler will attempt to tell the programmer the line and column numbers where the error occurred, but it is not precise (as of version 0.8).
Invalid Token Error:	A specific type of syntax error that occurs when the scanner encounters a token that is formed in an unusual way. Attempts to point the user to the location of the error, but is not precise.

Note:

Every error will cause the compiler to shut down without generating any code.

Appendix B: Sample Programs

```
//*********************************************************
//    ADD_SIX
//
//  Adds 6 ones together and stores them in a variable.
//  Prints the result.
//*********************************************************
private static void ADD_SIX()
{
      var INT1 : int;

      {
            INT1 = 1 + 1 + 1 + 1 + 1 + 1;
            put(INT1)
      }
} $
```

```
//*******************************************************
//    NEST_IF
//
//  Demonstrates a nested if statement.
//*******************************************************
private static void NEST_IF()
{
      var INT1, INT2, INT3, INT4 : int;

      {
            INT1 = 7;
            INT2 = 3;
            INT3 = 0;
            INT4 = 0;

            if(INT1 + INT2 == 10)       // true
            {
                  INT4 = 10;

                  if(INT4 == INT1 + INT2)     //true
                        INT3 = 3
            };

            put(INT3)                         // prints '3'
      }
} $
```