**Executive Summary**

This project is a scripting language paired with an interpretation engine. This involves the use of a parsing module and an interpretation module written in C# using the .NET Framework 4.0. The project’s development proceeds in a test-driven manner.

**Project Description**

The project is a scripting language (FluentC) that aims to resemble natural language and a paired application from which to run scripts written in FluentC.

**Project Requirements**

The project must be able to do the following:

* Run scripts written in accordance with the FluentC specification as a command-line utility (e.g. “FluentC someFile”)
* Display outputs in the form of ASCII text
* Read inputs in a command-line format

The FluentC language must include the following constructs:

* Variable Declaration
* Variable Assignment
* Variable Deletion
* Statement Execution
* Conditionally Executed Statements
* Condition Evaluation
* Numerical Comparisons
* Arithmetical Operations
* Character Comparisons (alphabetic ordering)
* While Loops
* Do While Loops
* Function Declaration for functions that return values
* Function Invocation for functions that return values
* Function Parameters for functions that return values
* Function Return Values
* Function Declaration for functions that lack return values.
* Function Invocation for functions that lack return values.
* Function Parameters for functions that lack return values.

**Project Specification**

**Engine Specification**

The engine must support all features of FluentC’s specification. The engine runs as a command line utility such that the console command “FluentC someFile” runs the script at someFile. The file extension does not matter. However, the engine behaves in a defined manner only for files containing only ASCII text written in accordance to the language specification.

**FluentC Specification**

FluentC supports three value types: Numbers, Strings, and Conditions. Numbers allow the four basic arithmetic operations for both integral and floating-point values. Strings hold all other data in textual representations. If a user attempts operations that mix Strings and Numbers, the attempt results in a String. Conditions indicate whether the conditional expression they hold is satisfied as being a true condition. As they are an indication rather than a concrete type, the value of a Condition may not be set explicitly.

**Number-valued expressions**

Number-valued expressions may exist in any of the following forms where *x* and *y* are any decimal values written in the form ‘0.0’ or the form ‘-0.0’ where 0 is any number of digits 0-9 or variables containing such values. FluentC evaluates number-valued expressions using order of operations.

|  |  |  |
| --- | --- | --- |
| **#** | **Form** | **Value** |
| *1* | *x* | *x* |
| *2* | *x*+*x* | *x* added to *x* |
| *3* | *x*+*y* | *x* added to *y* |
| *4* | *x-x* | *x* subtracted from *x* |
| *5* | *x-y* | *y* subtracted from *x* |
| *6* | *x\*x* | *x* multiplied by *x* |
| *7* | *x*\**y* | *x* multiplied by *y* |
| *8* | *x/x* | *x* divided by *x* |
| *9* | *x/y* | *x* divided by *y* |
| 10 | (*x*) | *x* |

**String-valued** **expressions**

String-valued expressions may exist in any of the following forms where *x* and *y* are any number of ASCII characters surrounded by double quote marks (“) or variables containing such values.

|  |  |  |
| --- | --- | --- |
| **#** | **Form** | **Value** |
| *1* | *x+x* | *x* appended to the end of *x* |
| *2* | *x+y* | *y* appended to the end of *x* |
| *3* | *x* | x |

**Condition-Valued Expressions**

Condition-valued expressions may exist in any of the following forms where *x* and *y* are valued expressions or variables of any type.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Form** | **Case in which satisfied given** *x* **and** *y* | | |
| **Numbers** | **Strings** | **Conditions** |
| 1 | *x* **is larger than** *y* | *x is greater than y* | *x is after y alphabetically* | *Invalid* |
| 2 | *x* **is smaller than** *y* | *x is less than y* | *x is before y alphabetically* | *Invalid* |
| 3 | *x* **is the same as** *x* | *Always* | *Always* | *Always* |
| 4 | *x* **is the same as** *y* | *x is equal to y* | *x is the same as y* | *x is the same as y* |
| 5 | **It is not the case that** *x* | *Invalid* | *Invalid* | *x is not true* |
| 6 | *x* **>** *y* | *x is greater than y* | *x is after y alphabetically* | *Invalid* |
| 7 | *x* **<** *y* | *x is less than y* | *x is before y alphabetically* | *Invalid* |
| 8 | *x* **=** *x* | *Always* | *Always* | *Always* |
| 9 | *x* **=** *y* | *x is equal to y* | *x is the same as y* | *x is the same as y* |
| 10 | !*x* | *Invalid* | *Invalid* | *x is not true* |
| 11 | (*x*) | *Invalid* | *Invalid* | *x is true* |

**Variable Declaration, Assignment, and Deletion**

**Let** *x* **exist.** – Declares a new typeless variable where *x* is the variable’s name.

**Let** *x***be *value*.** – Assigns *value* to *x* where *value* is any valued expression and *x* is a variable’s name. Assigning a value to an undeclared variable declares the variable and assigns *value* to it.

**Forget** *x***.** – Removes the variable *x* from the available cache of variables.

**Non-Valued Function Declaration and Invocation**

**How to** *function-name* **with** *parameter-names***:** *statement-block***. –** Declares a new function where *function-name* is the function’s name, *parameter-names* is a comma-separated list of parameter names, and *statement-block* is any valid block of statements.

*function-name* **with** *parameter-values***.** – Invokes the function identified by *function-name* where *parameter-values* is a comma-separated list of expressions that are supplied as parameters. The **with** keyword is optional if there are fewer than two parameters.

**Native Non-Valued Functions**

**Tell me** *x***.** - Reveals the value of the expression *x* to the user.

**Valued Function Declaration and Invocation**

**How to know** *function-name* **with** *parameter-names***:** *statement-block***.** *expression***! –** Declares a new function where *function-name* is the function’s name, *parameter-names* is a comma-separated list of parameter names, *statement-block* is any valid block of statements, and *expression* is any valid expression.

*function-name* **with** *parameter-values***.** – Invokes the function identified by *function-name* where *parameter-values* is a comma-separated list of expressions (optionally paired with names) that is supplied as parameters for the function. The **with** keyword is optional if there are fewer than two parameters.

**Native Valued Functions**

**Ask me for a number with the prompt** *x***.** – Prompts the user for a number, displaying the value of the expression *x* as a prompt.

**Ask me for a string with the prompt** *x*. – Prompts the user for a string, displaying the value of the expression *x* as a prompt.

**Give me the part of the string with the source string** *s***, the starting index** *x***, the ending index** *y***.** – Returns the substring of the source string expression *s* starting at the index indicated by the number expression *x* (inclusive) and ending at the index indicated by the number expression *y* (exclusive).

**Give me the length of the string** *s***.** – Returns the length of the string *s* as a number.

**Parameters within Functions**

FluentC treats parameters as variables within functions. There is an implied variable deletion at the end of each function declaration for each parameter.

**Statements, Statement Blocks and Comments**

A statement is any variable declaration, assignment, or deletion; a statement may also be any function invocation. All statements must end with a ‘.’ except within a statement block. In such a case, the last statement must end with a ‘.’ and all preceding statements end with a ‘;’.

Any set of characters ending with a ‘?’ is a comment and is ignored by the engine.

FluentC allows comments only before or after statements.

Some example statements:

**Let** *x* **be** 7**.**

**Tell me** “Hello World”**.**

An example statement block:

**Let** *x* **exist;** **Let** *x* **be** 7+92**; Let** *x* **be** *x*\*7**;Tell me** *x***; Forget** *x***.**

An example comment surrounded by statements:

**Let** *x* **exist.** *The previous statement declared a variable and the following assigns a value to it?* **Let** *x* **be** 27**.**

**Conditionals and Looping**

FluentC expresses conditional statements and loops in any of the following forms where *condition* is a Condition-Valued Expression and *statement-block* is any valid block of statements.

1. **If** *condition***,** *statement-block***.**
2. **While** *condition***...** *statement-block***.**
3. **Do...** *statement-block***.****...Until** *condition***.**

**Duplicate variable and function names**

FluentC does not define behavior where available variables, parameters, and/or functions share one name.

**Whitespace**

FluentC ignores whitespace outside of statements; FluentC also ignores whitespace between conditionals, loops, function declarations, and their respective statement blocks.

**Language Context Free Grammars**

In the following list of context-free grammars for FluentC, all keywords are italicized and all regular expressions are followed by “(regular expression)”.

whitespace 🡪 \s+ (regular expression)

number\_expression 🡪 number

number\_expression 🡪 number\_expression + number\_expression

number\_expression 🡪 number\_expression - number\_expression

number\_expression 🡪 number\_expression \* number\_expression

number\_expression 🡪 number\_expression / number\_expression

number\_expression 🡪 (number\_expression)

number\_expression 🡪 variable\_name

number 🡪 whole\_number.whole\_number

number 🡪 .whole\_number

number 🡪 -whole\_number.whole\_number

number 🡪 whole\_number

number 🡪 -whole\_number

number 🡪 -.whole\_number

whole\_number 🡪 digit

whole\_number 🡪 digit,whole\_number

digit 🡪 \d (regular expression)

string\_expression 🡪string\_expression *+* string\_expression

string\_expression 🡪 “.\*” (regular expression)

string\_expression 🡪 variable\_name

nonconditional\_expression 🡪 string\_expression

nonconditional\_expression 🡪 number\_expression

conditional\_expression 🡪 nonconditional\_expression *=* nonconditional\_expression

conditional\_expression 🡪 nonconditional \_expression *>* nonconditional \_expression

conditional\_expression 🡪 nonconditional \_expression *<* nonconditional \_expression

conditional\_expression 🡪 nonconditional \_expression *is larger than* nonconditional \_expression

conditional\_expression 🡪 nonconditional \_expression *is smaller than* nonconditional \_expression

conditional\_expression 🡪 nonconditional \_expression *is the same as* nonconditional \_expression

conditional\_expression 🡪 conditional\_expression *=* conditional\_expression

conditional\_expression 🡪 (conditional\_expression)

conditional\_expression 🡪 *It is not the case that* conditional\_expression

expression 🡪 nonconditional\_expression

expression 🡪 conditional\_expression

expression 🡪 valued\_function\_invocation

variable\_declaration 🡪 *Let* variable\_name *exist*

variable\_name 🡪 \b[^;.?,”+/\*-]+\b (regular expression)

variable\_assignment 🡪 *Let* variable\_name *be* expression

variable\_deletion 🡪 *Forget* variable\_name

statement\_part 🡪 variable\_declaration

statement\_part 🡪 variable\_assignment

statement\_part 🡪 variable\_deletion

comment 🡪 variable\_name?

statement 🡪 statement\_part.

statement 🡪 statement\_part;whitespace statement

statement 🡪 conditional\_statement

statement 🡪 looping\_statement

statement 🡪 function\_invocation

statement 🡪 comment statement

statement 🡪 statement comment

conditional\_statement 🡪 *If* conditional\_expression,whitespace statement

looping\_statement 🡪 *While* conditional\_expression...whitespace statement

looping\_statement 🡪 *Do*...whitespace multiple\_statements whitespace...*Until* conditional\_expression.

multiple\_statements 🡪 statement

multiple\_statements 🡪 multiple\_statements whitespace statement

function\_declaration 🡪 *How to* variable\_name:whitespace statement

valued\_function\_declaration 🡪 *How to know* variable\_name:whitespace statement whitespace expression!

function\_declaration 🡪 *How* *to* variable\_name *with* parameter\_list:whitespace statement

valued\_function\_declaration 🡪 *How* *to* *know* variable\_name *with* parameter\_list: whitespace multiple\_statements whitespace expression!

parameter\_list 🡪 parameter

parameter\_list 🡪 parameter\_list,whitespace parameter

parameter 🡪 variable\_name

function\_invocation 🡪 variable\_name.

function\_invocation 🡪 variable\_name expression.

function\_invocation 🡪 variable\_name *with* expression\_list.

expression\_list 🡪 expression

expression\_list 🡪 *the* parameter expression

expression\_list 🡪 expression\_list, whitespace expression\_list

valued\_function\_invocation 🡪 variable\_name

valued\_function\_invocation 🡪 variable\_name expression

valued\_function\_invocation 🡪 variable\_name *with* expression\_list

script 🡪 valued\_function\_declaration

script 🡪 multiple\_statements

**Project Architecture**

The software consists of two main parts: a parsing module and a script-running module.

Script Execution

Text File

Parsed Commands

I/O

**Technological Description**

Development of this project utilizes the C# language and the .NET Framework version 4.0.

**Novelty**

Many students have trouble starting to formalize their thoughts and transferring them into a language that computers can understand. This project aims to help ease the process of beginning to write software by shortening the gap between the English language and machine languages. Personally, this project will help me to both understand the way I think in terms of grammar and to learn how to parse grammar from text. Separately, I could use the finished language and application in order to teach siblings or progeny the art of programming.

**Development Process**

The project progresses with test-driven development. Successful test output marks the majority of checkpoints.

By the end of week 1 (1 week of time) *Before Saturday, April 14*:

* Language specification and grammar finalized.
* Variable declaration, assignment, and deletion tests written.
* Variable declaration, assignment, and deletion tests pass.

By the end of week 2 (1 week of time) *Before Saturday, April 21*:

* Non-Valued function tests written.
* Non-Valued function tests pass.
* The Engine can display output as ASCII text through the console.

By the end of week 3 (1 week of time) *Before Saturday, April 28*:

* Valued function tests written.
* Valued function tests pass.
* The Engine can take input as ASCII text through the console.

By Tuesday of week 4 (1 workday of time) *Before Tuesday, May 1*:

* Conditional and Looping tests written.

By Wednesday of week 5 (1 week + 1 workday of time) *Before Wednesday, May 9*:

* Conditional and Looping tests pass.
* All tests pass.

By end of week 5 (3 workdays of time) *Before Saturday, May 12*:

* Project defended in front of board.

**Resources**

* Language Used
  + C#
* Frameworks
  + .NET Framework 4.0
* Learning Resources:
  + MSDN
  + StackOverflow
  + Wikipedia
* IDE
  + Visual Studio 2010 Ultimate
* Development Machine
  + Lenovo Thinkpad T510
* Source Control
  + Mercurial
  + TortoiseHg
  + Hosted on googlecode