Clogic:
An Overview
by Tabari Alexander

Introduction

Clogic is a language designed to build and infer relationships between objects and allow a developer to easily handle situations where these relationships change. Based on concepts from logic programming languages, developers can establish facts about abstract objects, and use knowledge based on those truths in order to control their program. In addition, *Clogic* also allows a small set of imperative programming features for a bit of extra flexibility.

Background

Most of the logic programming concepts in *Clogic* are derived from *Prolog*, which is a declarative logic programming language. With *Prolog's* base datatype, an *atom*, a developer can create a set of facts and rules. Facts are a statement of truth about an *atom*, which itself is just a meaningless named abstraction. For example, the statement (not in *Prolog* syntax) "Jim is tall" declares a truth about the *atom*, Jim; Jim is tall. A rule is a statement that infers a truth from facts. An example of this would be "Football linebackers are people that are tall and heavy". If another truth were to be added, "Jim is heavy", then it could be inferred from the rule that Jim is a football linebacker.

Ultimately, the computation that is involved in *Prolog* is derived from a query, and the resulting answer is either a yes or no response, or a set of *atoms* that would make the query true. Given the rest of the already stated examples, a query in this context would be "Who is a football linebacker?", in which case the returned answer would be Jim. Additionally, if the query was "Is Michael a football linebacker?", the answer would be no. *Clogic* attempts to retain these concepts while at the same time allow queries and information that is obtained from the queries to be used in an imperative fashion.

Goals

Simplicity

Clogic is designed with simplicity in mind. In order to make this possible, most declarations

are made implicit and any variables that are used are dynamically typed in order to make writing the code fairly easy. In addition, the language syntax is also designed to keep the existence of declarative logic programming and imperative programming in mind, in an attempt to make really simple to blend the two styles.

Portable

To allow *Clogic* to be used on multiple platforms, it is designed to be compiled into bytecode to either be run via an interpreter similar to Java. The option also exists to build a standalone application, based on the bytecode.

Rule (In)validation Handling

One of the distinguishing features of *Clogic* is the capability for a developer to easily add logic for cases in which a rule becomes validated or invalidated. Traditional methods require the use of conditional statements to ensure that conditions are being met. With *Clogic's when*-clauses, the developer has the ability to be informed of the moment when the conditions are not being met and the opportunity to take an action at that time.

Language Features

A Program

At its core, a *Clogic* program is comprised of a set of function declarations, rules, and facts that are to be used in an execution block. A simple execution block could contain a single query, or a more complicated block might contain a long sequence of function calls, queries, and loops.

Datatypes

The primitive datatypes allowed in *Clogic* are integers, strings, lists, and atoms. Atoms in *Clogic* are similar to their *Prolog* counterparts; they are named meaningless abstractions. Other datatypes, like arrays and maps, can be created in the language by the creation of rules and facts.

Functions and Rules

Functions in *Clogic* are a means of grouping statements together for reuse. Recursion is supported, but overriding is not. Rules are the same in concept to their *Prolog* counterparts, but also act as a special type of function declaration for returning true or false values.

Program Examples

Any text between [[and]] are comments for the language.

```
[[This program is a code representation of the example given in the background.]]
tall(Jim); [[ Jim is tall. ]]
heavy(Jim); [[ Jim is heavy. ]]
footballLinebacker($person) => tall($person), heavy($person); [[ A football
linebacker is someone that is tall and heavy]]

[[ The curly braces denote a block of execution.]]
{
    ? footballLinebacker(Jim); [[ ? is a simplified construct for evaluating its
arguments and printing the result]]
    ? footballLinebacker(Michael);
    ? $p in footballLinebacker($p);
}
```

The result printed from this program is:

```
true
false
Jim
```

Another example that expands on the previous one:

```
[[ This program shows an example of
   1) how to create atoms that have the same facts
   2) how to negate a fact, and
   3) how to declare and use functions. ]]

tall(Jim);
heavy(Jim);

[[ Creates a new atom, Michael, that has all the same facts as Jim. Equivalent to
saying:
  tall(Michael);
  heavy(Michael);
It is very useful when multiple items have the same truths.]]
Michael isa Jim;
footballLinebacker($person) => tall($person), heavy($person);

[[ Creates a function countLinebackers which counts the number of linebackers.
Rather than an explicit return statement, the return value is placed after the
closing brace.  If no value is given, then an empty list is implicitly returned.]]
function countLinebackers()
{
    local $x = 0;
    for $p in footballLinebacker($p) do
        $x = $x + 1;
    done;
} $x;

[[ Creates the function run(), which negates the fact "heavy". No return statement
is given here, so the empty list [] is returned. ]]
function run($p) { ~heavy($p); };

[[ A when-clause establishes some code that is to be executed when a rule becomes
```

```
satisfied or invalidated.  The example below will execute any time an atom stops
being a footballLinebacker. ]]
when $p in ~footballLinebacker($p);
{
    ? $p + " is no longer a player.";
}


{
    ? $p in footballLinebacker($p);
    run(Jim);
    ? $p in footballLinebacker($p);
    ? count();
}
```

This program will print:

```
Jim, Michael
Jim is no longer a player.
Michael
1
```