Writing a Domain-Specific Language for Interactive Fiction

Evan Silberman

January 2007

Abstract

The creation of IFAlpha, a domain-specific language for the creation of interactive fiction games, is discussed. IFAlpha is written in the Ruby programming language, and in fact is fully hosted within it. The design principles of the project, including expressiveness and straightforward syntax, and the degree to which they have been fulfilled are examined.

1 Purpose

The purpose of the IFAlpha project is to create a domain-specific language for the creation of interactive fiction games. The language will be hosted within the Ruby programming language. The goal of the project is to create an intuitive declarative interface for the language that can be easily understood by non-programmers, and to implement this interface in such a way that a reasonably complete system for writing interactive fiction games is achieved.

2 Background

A domain-specific language (DSL) is a programming language that is limited in functionality compared to a general-purpose language (GPL), but is more expressive than a GPL for a certain set of tasks, its domain. DSLs exist in many forms, including within GPLs (for tasks like option parsing), as part of a software interface (such as the function and macro language used by a spreadsheet program), and as stand-alone languages for specialized purposes (such as PostScript, which describes documents to be output by a printer). By limiting functionality and scope, DSLs make tasks

within their problem domain easier to cope with from a programming standpoint, and are easier to use for non-programmers or inexperienced programmers.

Interactive fiction (IF) refers to a genre of games also known as "text adventures." Players take on the role of a single hero, and gameplay is characterized by puzzle-solving, exploration, and conversation with in-game characters. The interface is text-based, and accepts input in the form of player commands such as "`go north`" and "`get map`," and provides information to the player by way of describing the rooms, creatures, and objects the hero encounters.

Interactive fiction has been around for a long time. For about a decade, Inform 6 has been the standard language for enthusiasts writing IF games. It allows very complex games, but its syntax is not necessarily intuitive and is based on the C programming language to a large degree. In 2006, Inform 7 was released, encompassing an IDE, a language specification, and a compiler. Inform 7 has a syntax that is very close to natural language, operating within the paradigm of writing a book. Creating a system like this is beyond the scope of this project, but IFAlpha will hopefully encompass a reasonable portion of the functionality needed to create an IF game.

3 Procedure and Methodology

The backend for IFAlpha is based on a straightforward object model written in Ruby, a dynamic, high-level, interpreted programming language. Rooms, creatures, and things are instantiated as instances of the classes `Room`, `Creature`, and `Thing`. The game writer, however, is not exposed to these classes. Taking advantage of Ruby's flexibility and metaprogramming features, actual Ruby code in the files written by developers takes the form of simple declarative instructions.

Metaprogramming is my primary tool for implementing the language. In recent years, Ruby's metaprogramming features have been used to implement a variety of hosted DSLs, including some of the features of the Ruby on Rails web framework. Using metaprogramming, methods and instance variables can be created and set on the fly, and explicit receivers of method calls can be left undeclared, only to be provided later.

What follows is an example room declaration in the IFAlpha DSL. This declaration is all valid Ruby code, but the details of the object model are hidden from the programmer. The design of the language syntax is based on declarations of rooms, things, and creatures using top-level methods. Properties are assigned using what appear to be data fields, but are in fact instance methods of an instance of the `Room` class that is created by the backend.

```
room :living do
    name "Living room"
    desc "This is not Tom's favorite room"
    tom "Tom is nowhere nearby"
    exit :north, :fantasy
    fiesta "Tom frowns on fiestas in the living room"
end
```

This IFAlpha code isn't parsed or compiled by an external program. It's all valid Ruby code. The method `room` is being called, which takes as its arguments a symbol and a code block. The `room` method, defined in the backedn, creates a new instance of the `Room` class and evaluates the code block it receives in the context of that new instance of `Room` to an instance of the Room class.

```
def room(name,&block)
    newRoom = IFAlpha::Room.new
    newRoom.instance_eval &block
    IFAlpha::Room.rooms[name] = newRoom
    newRoom
end
```

The magic in this method is in `instance_eval`, which evaluates the code block passed to `room` as if the methods in it were instance methods of the new instance of `Room` that is created earlier.

Another example from the backend is the method in which exits from room to room are created. Obviously when the user declares exits from the room, the rooms may not yet all exist, depending on the order things are in the game file. So the backend creates a Proc, an object encapsulating a code block, which is evaluated later, adding exits to every room once every room exists.

```
def exit(*args)
    # The rooms may not all be in the hash yet,
    # so we have to save this for later.
    if args[0].class == Hash
        h = args[0]
    else
        h = Hash[*args]
    end
    p = Proc.new do
```

```
        for direction, room in h do
            theRoom = @@rooms[room]
            @exits[direction] = theRoom
        end
    end
    @exitproc = p
end
```

4 Evaluating Ease of Use

The evaluation phase of the IFAlpha project has yet to get underway. Since the goal of the IFAlpha project is to create an easy and intuitive interface for writing IF games, the true test of its effectiveness is not that the language works in combination with the backend, but that the language works in combination with the user. Testers will experiment with the language and provide feedback as to its ease of use and functionality. Users will also hopefully provide suggestions for features to implement and syntax improvements.

The intended procedure for ease of use evaluation is as follows. The complete IFAlpha system will be distributed to testers, along with an example game file and some brief instructions, which will advise users only on how to edit and run game files. Users will then be instructed to try making changes to the example game file and then to make their own short game from scratch, using only the example file as a guide to how the games are written. After they play around for a while, the testers will be given a survey asking them to rate the ease of use of the language and comment on their frustrations and successes. This feedback will allow for evaluation of the IFAlpha language and functionality and whether the design goals of the language have been fulfilled.

5 Conclusion

[No conclusion yet; the project isn't finished. My plans for third quarter include freezing the feature set, improving functionality, cleaning up sloppy code, and distributing the completed system for feedback on ease of use.]