A Domain-Specific Language for Interactive Fiction: Handout

Evan Silberman

January 2007

1 Introduction

A domain-specific language (DSL) is a programming language designed to be used for a specific and limited set of tasks. Compared to a general-purpose programming language (GPL), a DSL has limited expressiveness, but is more expressive than a GPL in its domain. I am designing a DSL hosted within Ruby for creating interactive fiction games. My goal is to create an intuitive and expressive language for creating IF games while hiding the details of implementation from the programmer.

2 The Language

What follows is an example room declaration in the DSL. This declaration is all valid Ruby code, but details like the object model are hidden from the programmer. My general paradigm for the language syntax is that the programmer declares rooms, creatures, and objects using top-level methods and assigns properties using what appear to be data fields, but are in reality instance methods of a Room class defined in 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
```

3 The Backend

The DSL isn't parsed or compiled, and it's not exactly interpreted either. It's all valid Ruby code. Using metaprogramming techniques, responsibility for dealing with methods is passed from the game file to the "room" method 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.

3.1 Events

The system for Events is based on looking at the signatures of all the events that have been declared and comparing them to the important parts of the command the player has typed in. The shell file, which connects the player interface to the backend, finds the first event that matches the command and, if necessary, applies it instead of the default verb response.

```
def report(*args)
    for event in Event.events.values
        if args == event.signature
            a = event.act
        end
    end
    return a
end
```

4 Ease-of-Use Testing

Now that the system is nearly feature-complete, I will soon be recruiting you to be ease-of-use testers. Since the goal of the 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. I will ask testers to experiment with the language and provide feedback as to its ease of use and functionality. I will also solicit suggestions for improvements to the language and possibly feature requests.