A Domain-Specific Language for Interactive Fiction: Handout

Evan Silberman

All Hallows' Eve, 2006

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.

One more example from the backend: exits. 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
```