GAMMA: A Strict yet Fair Programming Language

Ben Caimano - blc2129@columbia.edu
Weiyuan Li - wl2453@columbia.edu
Matthew H Maycock - mhm2159@columbia.edu
Arthy Padma Anandhi Sundaram - as4304@columbia.edu

A Project for Programming Languages and Translators,
taught by Stephen Edwards

Why GAMMA – or Why Orient the Object this way?

is a programming language of the "Scandanvian School" of Object orientation. The language provided two primary beneficial features:

1. The Pattern is used as a unifying concept for every element of the program – Classes, Procedures, anonymous Objects, etc.
2. Refinement and specialization is used instead of overriding to ensure that super-pattern behavior is respected by sub-patterns. A simple way analogy to understand this is to think how you would program in Java if all your methods had to be declared *abstract* or *final*. just provides this structuring and a little help to make it easier to use.

A consequence of the fact that patterns are refined and everything is a pattern is that classes can have nested classes and *those* nested classes can themselves be refined; classes can be *virtual* in the sense of virtual functions, etc in other languages.

The primary negative about however is that the syntax is horrible – absolutely atrocious. While the language provides a nice theoretical setting for object oriented development given it's pro's, the fact that the syntax is such a significant con clearly more than negates any of the benefits. In a world where LISP, the most beautiful language, is derided for its parentheses, a language like cannot succeed where Python, Ruby, and other easy to read languages exist.

And so this is the goal of the GAMMA programing language project; to make a new language with a feature set similar to what provides but with a readable syntax that is programmer friendly.

& GAMMA Differences

If we are making GAMMA as a replacement for then we need to talk about the changes this replacement represents. is written without much syntactic sugar; the Pattern syntax is used pervasively despite the fact that there are multiple repeated uses of the syntax. This is where a LISPer would say "Stop, Macro Time!" We have repeated syntactic patterns but a serious line clutter in their use.

This leads us to actually have two goals with respect to updating the syntax and usability of : first we will create a more readable version of the Pattern mechanism. As in , this will be GAMMA's primary way to define program entities, and so will still act as a unifying force and will be available to the programmer as the "under the hood" language for when it is needed.

The second goal will be to provide syntactic sugar for common Pattern syntax patterns (please blame developers for choosing Pattern as the name of their unifying concept, and making these sentences harder to read). We will provide syntax for classes, methods, and probably several other common patterns as we develop the language specification.

GAMMA Feature Set

GAMMA will provide the following features:

- Unifying Pattern Concept
- Refinement / Specialization
- Anonymous Pattern Instantiation
- Access specifiers respect object boundaries, not class boundaries

ray: The GAMMA Compiler

The compiler will proceed in two steps. First, the compiler will transform the source containing the syntactic sugar described above into a file consisting only of actual patterns. After this the compiler will transform general patterns into (hopefully portable) C code, and compile this to machine code with whatever compiler the user specifies.

Code Examples

Example 1: The All-mighty Pattern

```
Class Person:
  ## constructors take arguments, initialize state
  _init(String first_name, String last_name):
    this.first_name = first_name
    this.last_name = last_name

  ## the run method is a fundamental method that makes an object callable
  _run:
    println(this.first_name + " " + this.last_name + " is being stringified!")
    result = refine
    println(this.first_name + " " + this.last_name + " is strinfigied into " + result + "!'
    return result

  ## Simple member objects as variables
  Protected:
    String first_name
    String last_name

  Public:
    ## This is a common pattern that is equivalent
    ## to declaring a method virtual / abstract
    ## we need to think about this more (necessary
    ## to implement via compiler check; maybe returns
    ## NULL object if not implemented, etc, etc...
    String citizenID:
      return refine


class Student extends Person:
  ## Subclasses have to invoke their superclass,
  ## just like in common OOP languages (java/etc)
  _init(string first, string last, string grade):
    ## We can interpret _init because it should only be called through the Person() constru
    ## Since it is called inside, it must refer to the super
    self._init(first, last)
    this.level = grade

  ## Now we can refine!
  _run:
    return this.last_name + ", " + this.first_name + " is a " + this.level

  ## subclasses have their own data
  Private:
    String level # Freshman, etc...

  Public:
    String citizenID:
```