Gamma Language Reference Manual

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

October 27, 2013

Contents

1 Introduction

1.1 Why GAMMA? – The Core Concept

We propose to implement an elegant yet secure general purpose object-oriented programming language. Interesting features have been selected from the history of object-oriented programming and will be combined with the familiar ideas and style of modern languages.

GAMMA combines three disparate but equally important tenants:

1. Purely object-oriented

   GAMMA brings to the table a purely object oriented programming language where every type is modeled as an object–including the standard primitives. Integers, Strings, Arrays, and other types may be expressed in the standard fashion but are objects behind the scenes and can be treated as such.

2. Controllable

   GAMMA provides innate security by choosing object level access control as opposed to class level access specifiers. Private members of one object are inaccessible to other objects of the same type. Overloading is not allowed. No subclass can turn your functionality on its head.

3. Versatile

   GAMMA allows programmers to place "refinement methods" inside their code. Alone these methods do nothing, but may be defined by subclasses so as to extend functionality at certain important positions. Anonymous instantiation allows for extension of your classes in a quick easy fashion. Generic typing on method parameters allows for the same method to cover a variety of input types.

1.2 The Motivation Behind GAMMA

GAMMA is a reaction to the object-oriented languages before it. Obtuse syntax, flaws in security, and awkward implementations plague the average object-oriented language. GAMMA is intended as a step toward ease and comfort as an object-oriented programmer.

The first goal is to make an object-oriented language that is comfortable in its own skin. It should naturally lend itself to constructing API-layers and abstracting general models. It should serve the programmer towards their goal instead of exerting unnecessary effort through verbosity and awkwardness of structure.

The second goal is to make a language that is stable and controllable. The programmer in the lowest abstraction layer has control over how those higher may procede. Unexpected runtime behavior should be reduced through firmness of semantic structure and debugging should be a straight-forward process due to pure object and method nature of GAMMA.

2 Lexical Elements

2.1 Identifiers

Identifiers are used for the identification of variable, methods and types. An identifer is a sequence of alphanumeric characters, uppercase and lowercase, and underscores. A variable or method identifier must start with a lowercase alphabetic character. A type identifier must start with an uppercase alphabetic character.

2.2 Keywords

The following words are reserved keywords. They may not be used as identifiers:

and	class	else	elsif	extends	extends	false
if	init	nand	new	nor	null	or
private	protected	public	refinable	refine	refinement	return
super	this	to	true	while	void	xor

3 Grammar

• *Classs may extend another class or default to extending Object*
$\langle$class$\rangle \Rightarrow$
  class $\langle$class id$\rangle\langle$extend$\rangle$ **{** $\langle$class section$\rangle$* **}**
$\langle$extend$\rangle \Rightarrow$
  ϵ
 | **extends** $\langle$class id$\rangle$

• *Sections – private protected public refinements and main*
$\langle$class section$\rangle \Rightarrow$
  $\langle$refinement$\rangle$
 | $\langle$access group$\rangle$
 | $\langle$main$\rangle$

• *Refinements are named method dot refinement*
$\langle$refinement$\rangle \Rightarrow$
  refinement **{** $\langle$refine$\rangle$* **}**
$\langle$refine$\rangle \Rightarrow$
  $\langle$return type$\rangle\langle$var id$\rangle$**.**$\langle$var id$\rangle\langle$params$\rangle$ **{** $\langle$statement$\rangle$* **}**

• *Access groups contain all the members of a class*
$\langle$access group$\rangle \Rightarrow$
  $\langle$access type$\rangle$ **{** $\langle$member$\rangle$* **}**
$\langle$access type$\rangle \Rightarrow$
  private
 | **protected**
 | **public**
$\langle$member$\rangle \Rightarrow$
  $\langle$var decl$\rangle$
 | $\langle$method$\rangle$
 | $\langle$init$\rangle$
$\langle$method$\rangle \Rightarrow$
  $\langle$return type$\rangle\langle$var id$\rangle\langle$params$\rangle$ **{** $\langle$statement$\rangle$* **}**
$\langle$init$\rangle \Rightarrow$
  init $\langle$params$\rangle$ **{** $\langle$statement$\rangle$* **}**

• *Main is special – not instance data starts execution*
$\langle$main$\rangle \Rightarrow$
  main (String[] $\langle$var id$\rangle$ **) {** $\langle$statement$\rangle$* **}**

- *Finally the meat and potatoes*

⟨statement⟩ ⇒
    ⟨var decl⟩ **;**
  | ⟨super⟩ **;**
  | ⟨return⟩ **;**
  | ⟨conditional⟩
  | ⟨loop⟩
  | ⟨expression⟩ **;**

- *Super invocation is so we can do constructor chaining*

⟨super⟩ ⇒
    super (⟨args⟩ **)**

- *Methods need to be able to return something too*

⟨return⟩ ⇒
    return ⟨expression⟩

- *Basic control structures*

⟨conditional⟩ ⇒
    if (⟨expression⟩ **) {** ⟨statement⟩* **}** ⟨else⟩
⟨else⟩ ⇒
    ϵ
  | ⟨elseif⟩ **else {** ⟨statement⟩* **}**
⟨elseif⟩ ⇒
    ϵ
  | ⟨elseif⟩ **elsif (** ⟨expression⟩ **) {** ⟨statement⟩* **}**
⟨loop⟩ ⇒
    while (⟨expression⟩ **) {** ⟨statement⟩* **}**

- *Anything that can result in a value*

⟨expression⟩ ⇒
    ⟨assignment⟩
  | ⟨invocation⟩
  | ⟨field⟩
  | ⟨deref⟩
  | ⟨arithmetic⟩
  | ⟨test⟩
  | ⟨instantiate⟩
  | ⟨refine expr⟩
  | ⟨literal⟩
  | **(** ⟨expression⟩ **)**
  | **null**

• *Assignment – putting one thing in another*
⟨assignment⟩ ⇒
    ⟨expression⟩ **:=** ⟨expression⟩

• *Member / data access*
⟨invocation⟩ ⇒
    ⟨expression⟩ **.** ⟨invoke⟩
  | ⟨invoke⟩
⟨invoke⟩ ⇒
    ⟨var id⟩ **()**
  | ⟨var id⟩ **(** ⟨args⟩ **)**
⟨field⟩ ⇒
    ⟨expression⟩ **.** ⟨var id⟩
⟨deref⟩ ⇒
    ⟨expression⟩ [⟨expression⟩]

• *Basic arithmetic can and will be done!*
⟨arithmetic⟩ ⇒
    ⟨expression⟩⟨bin op⟩⟨expression⟩
  | ⟨unary op⟩⟨expression⟩
⟨bin op⟩ ⇒
    +
  | **-**
  | *
  | /
  | %
  | ^
⟨unary op⟩ ⇒
    -

• *Common boolean predicates*
⟨test⟩ ⇒
    ⟨expression⟩⟨bin pred⟩⟨expression⟩
  | ⟨unary pred⟩⟨expression⟩
  | **refinable (** ⟨var id⟩ **)**
⟨bin pred⟩ ⇒
    and
  | **or**
  | **xor**
  | **nand**
  | **nor**
  | <
  | <=

| **=**
| **!=**
| >=
| >
⟨unary pred⟩ ⇒
!

• *Making something*
⟨instantiate⟩ ⇒
⟨object instantiate⟩
| ⟨array instantiate⟩
⟨object instantiate⟩ ⇒
new ⟨class id⟩
| **new** ⟨class id⟩ **(** ⟨args⟩ **)**
⟨array instantiate⟩ ⇒
new ⟨type⟩ [⟨expression⟩]

• *Refinement takes a specific specialization and notes the required return type*
⟨refine expr⟩ ⇒
refine ⟨specialize⟩ **to** ⟨type⟩
⟨specialize⟩ ⇒
⟨var id⟩ **()**
| ⟨var id⟩ **(** ⟨args⟩ **)**

• *Literally necessary*
⟨literal⟩ ⇒
⟨int lit⟩
| ⟨bool lit⟩
| ⟨float lit⟩
| ⟨string lit⟩
⟨float lit⟩ ⇒
⟨digit⟩+ **.** ⟨digit⟩+
⟨int lit⟩ ⇒
⟨digits⟩+
⟨bool lit⟩ ⇒
true
| **false**
⟨string lit⟩ ⇒
“⟨string escape seq⟩**”**

• *Params and args are as expected*
⟨params⟩ ⇒

($\langle$paramlist$\rangle$ **)**

$\langle$paramlist$\rangle \Rightarrow$

$\langle$var decl$\rangle$

| $\langle$paramlist$\rangle$ **,** $\langle$var decl$\rangle$

$\langle$args$\rangle \Rightarrow$

$\langle$expression$\rangle$

| $\langle$args$\rangle$ **,** $\langle$expression$\rangle$

- *All the basic stuff we've been saving up until now*

$\langle$var decl$\rangle \Rightarrow$

$\langle$type$\rangle\langle$var id$\rangle$

$\langle$return type$\rangle \Rightarrow$

unit

| $\langle$type$\rangle$

$\langle$type$\rangle \Rightarrow$

$\langle$class id$\rangle$

| $\langle$type$\rangle$[]

$\langle$class id$\rangle \Rightarrow$

$\langle$upper$\rangle\langle$ualphanum$\rangle$*

$\langle$var id$\rangle \Rightarrow$

$\langle$lower$\rangle\langle$ualphanum$\rangle$*