

GAMMA

Γ α γ

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

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 tenets:

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.

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.

1.3 GAMMA Feature Set

GAMMA will provide the following features:

- Universal objecthood
- Optional "refinement" functions to extend superclass functionality
- Anonymous class instantiation
- Static typing
- Access specifiers that respect object boundaries, not class boundaries

1.4 ray: The GAMMA Compiler

The compiler will proceed in two steps. First, the compiler will interpret the source containing possible syntactic shorthand into a file consisting only of the most concise and structurally sound GAMMA core. 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.

Contents

1	Introduction	3
1.1	Why GAMMA? – The Core Concept	3
1.2	The Motivation Behind GAMMA	3
1.3	GAMMA Feature Set	3
1.4	ray: The GAMMA Compiler	4
2	Language Tutorial	8
3	LRM	10
3.1	Lexical Elements	10
3.1.1	Whitespace	10
3.1.2	Identifiers	10
3.1.3	Keywords	10
3.1.4	Operators	10
3.1.5	Literal Classes	10
3.1.6	Comments	12
3.1.7	Separators	12
3.2	Semantics	12
3.2.1	Types and Variables	12
3.2.2	Classes, Subclasses, and Their Members	12
3.2.3	Methods	13
3.2.4	Refinements	13
3.2.5	Constructors (init)	14
3.2.6	Main	14
3.2.7	Expressions and Statements	14
3.3	Syntax	14
3.3.1	Statement Grouping via Bodies	14
3.3.2	Variables	15
3.3.3	Methods	16
3.3.4	Classes	17
3.3.5	Conditional Structures	19
3.3.6	Refinements	19
3.4	Operators and Literal Types	20
3.4.1	The Operator =	20
3.4.2	The Operators != and <>	20
3.4.3	The Operator <	20
3.4.4	The Operator >	21
3.4.5	The Operator <=	21

3.4.6	The Operator <code>>=</code>	21
3.4.7	The Operator <code>+</code>	21
3.4.8	The Operator <code>-</code>	21
3.4.9	The Operator <code>*</code>	21
3.4.10	The Operator <code>/</code>	21
3.4.11	The Operator <code>%</code>	21
3.4.12	The Operator <code>^</code>	22
3.4.13	The Operator <code>:=</code>	22
3.4.14	The Operators <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code> , and <code>^=</code>	22
3.4.15	The Operator <code>and</code>	22
3.4.16	The Operator <code>or</code>	22
3.4.17	The Operator <code>not</code>	22
3.4.18	The Operator <code>nand</code>	22
3.4.19	The Operator <code>nor</code>	22
3.4.20	The Operator <code>xor</code>	22
3.4.21	The Operator <code>refinable</code>	22
3.5	Grammar	22
4	Project Planning	28
4.1	Planning Techniques	28
4.2	Ocaml Style Guide for the Development of the Ray Compiler	28
4.3	Project Timeline	30
4.4	Team Roles	31
4.5	Development Environment	32
4.5.1	Programming Languages	32
4.5.2	Development Tools	32
4.6	Project Log	33
5	Architectural Design	35
5.1	Block Diagrams	35
5.1.1	Structure by Module	35
5.1.2	Structure by Toplevel Ocaml Function	36
5.2	Component Connective Interfaces	36
5.3	Component Authorship	37
6	Test Plan	39
6.1	Examples Gamma Programs	39
6.1.1	Hello World	39
6.1.2	I/O	42
6.1.3	Argument Reading	46

6.2	Test Suites	50
6.2.1	Desired Failure Testing	50
6.2.2	Statement Testing	53
6.2.3	Expression Testing	55
6.2.4	Structure Testing	64
6.2.5	A Complex Test	65
7	Lessons Learned	67
8	Appendix	69

2 Language Tutorial

The structure of the example below should be intimately familiar to any student of Object-Oriented Programming.

```
1 class IOTest:
2     public:
3         init():
4             super()
5
6         void interact():
7             Printer p := system.out
8             Integer i := promptInteger("Please enter an integer")
9             Float f := promptFloat("Please enter a float")
10            p.printString("Sum of integer + float = ")
11            p.printFloat(i.toF() + f)
12            p.printString("\n")
13
14        private:
15            void prompt(String msg):
16                system.out.printString(msg)
17                system.out.printString(": ")
18
19            Integer promptInteger(String msg):
20                prompt(msg)
21                return system.in.scanInteger()
22
23            Float promptFloat(String msg):
24                prompt(msg)
25                return system.in.scanFloat()
26
27        main(System system, String[] args):
28            IOTest test := new IOTest()
29            test.interact()
```

Example 1: "A simple I/O example"

We start with a definition of our class.

```
1 class IOTest:
```

We follow by starting a **public** access level, defining an **init** method for our class, and calling the **super** method inside the **init** method. (Since we have not indicated a superclass for **IOTest**, this **super** method is for **Object**.)

```
1     public:
2         init():
3             super()
```

We also define the **private** access level with three methods: a generic method that prints a prompt message and two prompts for **Integers** and **Floats** respectively. These prompts call the generic message and then read from **system.in**.


```

1 private:
2     void prompt(String msg):
3         system.out.println(msg)
4         system.out.println(": ")
5
6     Integer promptInteger(String msg):
7         prompt(msg)
8         return system.in.scanInteger()
9
10    Float promptFloat(String msg):
11        prompt(msg)
12        return system.in.scanFloat()

```

We then write a method under the **public** access level. This calls our **private** level methods, convert our **Integer** to a **Float** and print our operation.

```

1     void interact():
2         Printer p := system.out
3         Integer i := promptInteger("Please enter an integer")
4         Float f := promptFloat("Please enter a float")
5         p.println("Sum of integer + float = ")
6         p.printFloat(i.toFloat() + f)
7         p.println("\n")

```

Finally, we define the **main** method for our class. We just make a new object of our class in that method and call our sole public method on it.

```

1     main(System system, String[] args):
2         IOTest test := new IOTest()
3         test.interact()

```

3 LRM

3.1 Lexical Elements

3.1.1 Whitespace

The new line (line feed), form feed, carriage return, and vertical tab characters will all be treated equivalently as vertical whitespace. Tokens are separated by horizontal (space, tab) and vertical (see previous remark) whitespace of any length (including zero).

3.1.2 Identifiers

Identifiers are used for the identification of variables, methods and types. An identifier is a sequence of alphanumeric characters, uppercase and lowercase, and underscores. A type identifier must start with an uppercase letter; all others must start with a lower case letter. Additionally, the lexeme of a left bracket followed immediately by a right bracket – `[]` – may appear at the end of a type identifier in certain contexts, and that there may be multiple present in this case (denoting arrays, etc). The legal contexts for such will be described later.

3.1.3 Keywords

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

and	class	else	elsif	extends	false
if	init	main	nand	new	nor
not	or	private	protected	public	refinable
refine	refinement	return	super	this	to
true	void	while	xor		

3.1.4 Operators

There are a large number of (mostly binary) operators:

=	!=	<>	<	<=	>	>=
+	-	*	/	%	^	:=
+=	-=	*=	/=	%=	^=	
and	or	not	nand	nor	xor	refinable

3.1.5 Literal Classes

A literal class is a value that may be expressed in code without the use of the new keyword. These are the fundamental units of program.

Integer Literals An integer literal is a sequence of digits. It may be prefaced by a unary minus symbol. For example:

- 777
- 42
- 2

- -999
- 0001

Float Literals A float literal is a sequence of digits and exactly one decimal point/period. It must have at least one digit before the decimal point and at least one digit after the decimal point. It may also be prefaced by a unary minus symbol. For example:

- 1.0
- -0.567
- 10000.1
- 00004.70000
- 12345.6789

Boolean Literals A boolean literal is a single keyword, either `true` or `false`.

String Literals A string literal consists of a sequence of characters enclosed in double quotes. Note that a string literal can have the new line escape sequence within it (among others, see below), but cannot have a new line (line feed), form feed, carriage return, or vertical tab within it; nor can it have the end of file. Please note that the sequence may be of length zero. For example:

- `"Yellow matter custard"`
- `"`
- `"Dripping\n from a dead"`
- `"'s 3y3"`

The following are the escape sequences available within a string literal; a backslash followed by a character outside of those below is an error.

- `\a` - u0007/alert/BEL
- `\b` - u0008/backspace/BB
- `\f` - u000c/form feed/FF
- `\n` - u000a/linefeed/LF
- `\r` - u000d/carriage return/CR
- `\t` - u0009/horizontal tab/HT
- `\v` - u000b/vertical tab/VT
- `\'` - u0027/single quote
- `\"` - u0022/double quote
- `\\` - u005c/backslash
- `\0` - u0000/null character/NUL

3.1.6 Comments

Comments begin with the sequence `/*` and end with `*/`. Comments nest within each other. Comments must be closed before the end of file is reached.

3.1.7 Separators

The following characters delineate various aspects of program organization (such as method arguments, array indexing, blocks, and expressions):

[] () { ,

A notable exception is that `[]` itself is a lexeme related to array types and there can be no space between the two characters in this regard.

3.2 Semantics

3.2.1 Types and Variables

Every *variable* in Gamma is declared with a *type* and an *identifier*. The typing is static and will always be known at compile time for every variable. The variable itself holds a reference to an instance of that type. At compile time, each variable reserves space for one reference to an instance of that type; during run time, each instantiation reserves space for one instance of that type (i.e. *not* a reference but the actual object). To be an instance of a type, an instance must be an instance of the class of the same name as that type or an instance of one of the set of descendants (i.e. a subclass defined via **extends** or within the transitive closure therein) of that class. For the purposes of method and refinement return types there is a special keyword, **void**, that allows a method or refinement to use the **return** keyword without an expression and thus not produce a value.

Array Types When specifying the type of a variable, the type identifier may be followed by one or more `[]` lexemes. The lexeme implies that the type is an *array type* of the *element type* that precedes it in the identifier. Elements of an array are accessed via an expression resulting in an array followed by a left bracket `[`, an expression producing an offset index of zero or greater, and a right bracket `]`. Elements are of one dimension less and so are themselves either arrays or are individual instances of the overall class/type involved (i.e. **BankAccount**).

3.2.2 Classes, Subclasses, and Their Members

GAMMA is a pure object-oriented language, which means every value is an object – with the exception that **this** is a special reference for the object of the current context; the use of **this** is only useful inside the context of a method, **init**, or refinement and so cannot be used in a **main**. **init** and **main** are defined later.

A class always extends another class; a class inherits all of its superclass's methods and may refine the methods of its superclass. A class must contain a constructor routine named *init* and it must invoke its superclass's constructor via the **super** keyword – either directly or transitively by referring to other constructors within the class. In the scope of every class, the keyword **this** explicitly refers to the instance itself. Additionally, a class contains three sets of *members* organized in *private*, *protected*, and *public* sections. Members may be either variables or methods. Members in the public section may be accessed (see syntax) by any other object. Members of the protected section may be accessed only by an object of that type or a descendant (i.e. a subtype defined transitively via the **extends** relation). Private members are only accessible by the members defined in that class (and are not accessible to descendants). Note that access

is enforced at object boundaries, not class boundaries – two `BankAccount` objects of the same exact type cannot access each other’s balance, which is in fact possible in both Java & C++, among others. Likewise if `SavingsAccount` extends `BankAccount`, an object of savings account can access the protected instance members of `SavingsAccount` related to its own data, but *cannot* access those of another object of similar type (`BankAccount` or a type derived from it).

The Object Class The Object class is the superclass of the entire class hierarchy in GAMMA. All objects directly or indirectly inherit from it and share its methods. By default, class declarations without extending explicitly are subclasses of Object.

The Literal Classes There are several *literal classes* that contain uniquely identified members (via their literal representation). These classes come with methods developed for most operators. They are also all subclasses of Object.

Anonymous Classes A class can be anonymously subclassed (such must happen in the context of instantiation) via refinements. They are a subclass of the class they refine, and the objects are a subtype of that type. Note that references are copied at anonymous instantiation, not values.

3.2.3 Methods

A method is a reusable subdivision of code that takes multiple (possibly zero) values as arguments and can either return a value of the type specified for the method, or not return any value in the case that the return type is `void`.

It is a semantic error for two methods of a class to have the same signature – which is the return type, the name, and the type sequence for the arguments. It is also a semantic error for two method signatures to only differ in return type in a given class.

Operators Since all variables are objects, every operator is in truth a method called from one of its operands with the other operands as arguments – with the notable exception of the assignment operators which operate at the language level as they deal not with operations but with the maintenance of references (but even then they use methods as `+=` uses the method for `+` – but the assignment part itself does not use any methods). If an operator is not usable with a certain literal class, then it will not have the method implemented as a member.

3.2.4 Refinements

Methods and constructors of a class can have *refine* statements placed in their bodies. Subclasses must implement *refinements*, special methods that are called in place of their superclass’ refine statements, unless the refinements are guarded with a boolean check via the `refinable` operator for their existence – in which case their implementation is optional.

It is a semantic error for two refinements of a method to have the same signature – which is the return type, the method they refine, the refinement name, and the type sequence for the arguments. It is also a semantic error for two method signatures to only differ in return type in a given class.

A refinement cannot be implemented in a class derived by a subclass, it must be provided if at all in the subclass. If it is desired that further subclassing should handle refinement, then these further refinements can be invoked inside the refinements themselves (syntactic sugar will make this easier in future releases). Note that refining within a refinement results in a refinement of the same method. That is, using `refine extra(someArg) to String` inside the refinement `String toString.extra(someType someArg)`

will (possibly, if not guarded) require the next level of subclassing to implement the extra refinement for `toString`.

3.2.5 Constructors (`init`)

Constructors are invoked to arrange the state of an object during instantiation and accept the arguments used for such. It is a semantic error for two constructors to have the same signature – that is the same type sequence.

3.2.6 Main

Each class can define at most one `main` method to be executed when that class will ‘start the program execution’ so to speak. Main methods are not instance methods and cannot refer to instance data. These are the only ‘static’ methods allowed in the Java sense of the word. It is a semantic error for the main to have a set of arguments other than a system object and a String array.

3.2.7 Expressions and Statements

The fundamental nature of an expression is that it generates a value. A statement can be a call to an expression, thus a method or a variable. Not every statement is an expression, however.

3.3 Syntax

The syntactic structures presented in this section may have optional elements. If an element is optional, it will be wrapped in the lexemes `<<` and `>>`. This grouping may nest. On rare occasions, a feature of the syntax will allow for truly alternate elements. The elements are presented in the lexemes `{` and `}`, each feature is separated by the lexeme `|`. If an optional element may be repeated without limit, it will finish with the lexeme `...`.

3.3.1 Statement Grouping via Bodies

A body of statements is a series of statements at the same level of indentation.

```
1  <<stmt1_statement>>
2  <<stmt2_statement>>
3  <<...>>
```

This pattern is elementary to write.

```
1  Mouse mouse = new Mouse()
2  mouse.click()
3  mouse.click_fast()
4  mouse.click("Screen won't respond")
5  mouse.defenestrate()
```

Example 2: Statement Grouping of a Typical Interface Simulator

3.3.2 Variables

Variable Assignment Assigning an instance to a variable requires an expression and a variable identifier:

```
1 var_identifier := val_expr
```

If we wanted to assign instances of Integer for our pythagorean theorem, we'd do it like so:

```
1 a := 3
2 b := 4
```

Example 3: Variable Assignment for the Pythagorean Theorem

Variable Declaration Declaring a variable requires a type and a list of identifiers delimited by commas. Each identifier may be followed by the assignment operator and an expression so as to combine assignment and declaration.

```
1 var_type var1_identifier << := val1_expr >> << , var2_identifier << := val2_expr >> >>
   <<...>>
```

If we wanted to declare variables for the pythagorean theorem, we would do it like so:

```
1 Float a, b, c
```

Example 4: Variable Initialization for the Pythagorean Theorem

Array Declaration Declaring an array is almost the same as declaring a normal variable, simply add square brackets after the type. Note that the dimension need be given.

```
1 element_type []...[] array_identifier << := new element_type [] (dim1_expr, ..., dimN_expr) >>
```

If we wanted a set of triangles to operate on, for instance:

```
1 Triangle [] triangles := new Triangle [] (42)
```

Example 5: Array Declaration and Instantiation of Many Triangles

Or perhaps, we want to index them by their short sides and initialize them later:

```
1 Triangle [] [] triangles
```

Example 6: Array Declaration of a 2-Degree Triangle Array

Array Dereferencing To dereference an instance of an array type down to an instance its element type, place the index of the element instance inside the array instance between [and] lexemes after the variable identifier. This syntax can be used to provide a variable for use in assignment or expressions.

```
1 var_identifier [ dim1_index ] ... [ dimN_index ]
```

Perhaps we care about the fifth triangle in our array from before for some reason.

```
1 Triangle my_triangle := triangles[4]
```

Example 7: Array Dereferencing a Triangle

3.3.3 Methods

Method Invocation Invoking a method requires at least an identifier for the method of the current context (i.e. implicit **this** receiver). The instance that the method is invoked upon can be provided as an expression. If it is not provided, the method is invoked upon **this**.

```
1 << instance_expr.>>method_identifier(<<arg1_expr>> <<, arg2_expr>> <<...>>)
```

Finishing our pythagorean example, we use method invocations and assignment to calculate the length of our third side, c.

```
1 c := ((a.power(2)).plus(b.power(2))).power(0.5)
```

Example 8: Method Invocation for the Pythagorean Theorem Using Methods

Method Invocation Using Operators Alternatively, certain base methods allow for the use of more familiar binary operators in place of a method invocation.

```
1 op1_expr operator op2_expr
```

Using operators has advantages in clarity and succinctness even if the end result is the same.

```
1 c := ( a^2 + b^2 )^0.5
```

Example 9: Method Invocation for the Pythagorean Theorem Using Operators

Operator Precedence In the previous examples, parentheses were used heavily in a context not directly related to method invocation. Parentheses have one additional function: they modify precedence among operators. Every operator has a precedence in relation to its fellow operators. Operators of higher precedence are enacted first. Please consider the following table for determining precedence:

:=	+=	-=	*=	/=	%=	^=
or	xor	nor				
and	nand					
=	<>	=/=				
>	<	>=	<=			
+	-					
*	/	%				
unary minus						
not	^					
array dereferencing	()				
method invocation						

Table 1: Operator Precedence

Method Declaration & Definition A method definition begins with the return type – either a type (possibly an n-dimensional array) or void. There is one type and one identifier for each parameter; and they are delimited by commas. Following the parentheses is a colon before the body of the method at an increased level of indentaiton. There can be zero or more statements in the body. Additionally, refinements may be placed throughout the statements.

```
1  {{return_type | Void}} method_identifier (<<arg1_type arg1_identifier>> <<, arg2_type
   arg2_identifier>> <<...>>): method_body
```

Finally, we may define a method to do our pythagorean theorem calculation.

```
1  Float pythagorean_theorem(Float a, Float b):
2      Float c
3      c := ( a^2 + b^2 ) ^0.5
4      return c
```

Example 10: Method Definition for the Pythagorean Theorem

3.3.4 Classes

Section Definition Every class always has at least one section that denotes members in a certain access level. A section resembles a body, it has a unified level of indentation throughout a set of variable and method declarations, including `init` methods.

```
1  <<{{method1_decl | var1_decl | init1_decl}}>>
2  <<{{method2_decl | var2_decl | init2_decl}}>>
3  <<...>>
```

Class Declaration & Definition A class definition always starts with the keyword `class` followed by a type (i.e. capitalized) identifier. There can be no brackets at the end of the identifier, and so this is a case where the type must be purely alphanumeric mixed with underscores. It optionally has the keyword `extends` followed by the identifier of the superclass. What follows is the class body at consistent indentation: an

optional `main` method, the three access-level member sections, and refinements. There may be `init` methods in any of the three sections, and there must be (semantically enforced, not syntactically) an `init` method either in the protected or public section (for otherwise there would be no way to generate instances).

While the grammar allows multiple main methods to be defined in a class, any more than one will result in an error during compilation.

```

1  class class_identifier <<extends superclass_identifier>>:
2      <<main.method>>
3      <<{{private | protected | public | refinement}} section1>>
4      <<{{private | protected | public | refinement}} section1>>
5      <<...>>

```

Let's make a basic geometric shape class in anticipation of later examples. We have private members, two access-level sections and an `init` method. No `extends` is specified, so it is assumed to inherit from `Object`.

```

1  class Geometric.Shape:
2      private:
3          String name
4          Float area
5          Float circumference
6      public:
7          init (String name):
8              this.name = name
9              if (refinable(improve_name)):
10                 this.name += refine improve_name() to String
11
12             return
13
14         Float get_area():
15             Float area
16             area := refine custom_area() to Float

```

Example 11: Class Declaration for a Geometric Shape class

Class Instantiation Making a new instance of a class is simple.

```

1  new class_identifier(<<arg1_expr>> <<arg2_expr>> <<...>>)

```

For instance:

```

1  Geometric.Shape = new Geometric.Shape("circle")

```

Example 12: Class Instantiation for a Geometric Shape class

Anonymous Classes An anonymous class definition is used in the instantiation of the class and can only provide refinements, no additional public, protected, or private members. Additionally no `init` or `main` can be given.

```

1 new superclass_identifier(<<arg1_expr>> <<,arg2_expr>> <<...>>):
2   <<refinements>>

```

3.3.5 Conditional Structures

If Statements The fundamental unit of an if statement is a keyword, followed by an expression between parentheses to test, and then a body of statements at an increased level of indentaiton. The first keyword is always **if**, each additional condition to be tested in sequence has the keyword **elsif** and a final body of statements may optionally come after the keyword **else**.

```

1 if (test1_expr): if1_body
2 <<elsif (test2_expr) if2_body>>
3 <<elsif (test3_expr) if3_body>>
4 <<...>>
5 <<else if4_body>>

```

While Statements A while statement consists of only the **while** keyword, a test expression and a body.

```

1 while (test_expr): while_body

```

3.3.6 Refinements

The Refine Invocation A refine invocation will eventually evaluate to an expression as long as the appropriate refinement is implemented. It is formed by using the keyword **refine**, the identifier for the refinement, the keyword **to**, and the type for the desired expression. Note that a method can only invoke its own refinements, not others – but refinements defined *within* a class can be called. This is done in addition to normal invocation. Also note that all overloaded methods of the same name share the same refinements.

```

1 refine refine_identifier to refine_type

```

The Refinable Test The original programmer cannot guarantee that future extenders will implement the refinement. If it is allowable that the refinement does not happen, then the programmer can use the **refinable** keyword as a callable identifier that evaluates to a Boolean instance. If the programmer contrives a situation where the compiler recognizes that a refinement is guarded but still executes a refine despite the refinement not existing, a runtime error will result.

```

1 refinable(refinement_identifier)

```

The Refinement Declaration To declare a refinement, declare a method in your subclass' refinement section with the special identifier `supermethod_identifier.refinement_identifier`.

3.4 Operators and Literal Types

The following defines the approved behaviour for each combination of operator and literal type. If the literal type is not listed for a certain operator, the operator's behaviour for the literal is undefined. These operators never take operands of different types.

3.4.1 The Operator =

Integer If two Integer instances have the same value, = returns **true**. If they do not have the same value, it returns **false**.

Float If two Float instances have an absolute difference of less than or equal to an epsilon of 2^{-24} , = returns **true**. If the absolute difference is greater than that epsilon, it returns **false**.

Boolean If two Boolean instances have the same keyword, either **true** or **false**, = returns **true**. If their keyword differs, it returns **false**.

String If two String instances have the same sequence of characters, = returns **true**. If their sequence of characters differs, it returns **false**.

3.4.2 The Operators != and <>

Integer If two Integer instances have a different value, != and <> return **true**. If they do have the same value, they return **false**.

Float If two Float instances have an absolute difference of greater than an epsilon of 2^{-24} , != returns **true**. If the absolute difference is less than or equal to that epsilon, it returns **false**.

Boolean If two Boolean instances have different keywords, != and <> return **true**. If their keywords are the same, they return **false**.

String If two String instances have the different sequences of characters, != and <> return **true**. If their sequence of characters is the same, they return **false**.

3.4.3 The Operator <

Integer and float If the left operand is less than the right operand, < returns **true**. If the right operand is less than or equal to the left operand, it returns **false**.

String If the left operand comes before the right operand in dictionary order, < returns **true**. If the left operand comes after the right operand in dictionary order, it returns **false**. If the two operands have the same sequence of characters, it returns **false**.

3.4.4 The Operator >

Integer and float If the left operand is greater than the right operand, > returns **true**. If the right operand is greater than or equal to the left operand, it returns **false**.

String If the left operand comes after the right operand in dictionary order, < returns **true**. If the left operand comes before the right operand in dictionary order, it returns **false**. If the two operands have the same sequence of characters, it returns **false**.

3.4.5 The Operator <=

Integer and float If the left operand is less than or equal to the right operand, < returns **true**. If the right operand is less than the left operand, it returns **false**.

String If the left operand comes before the right operand in dictionary order, < returns **true**. If the left operand comes after the right operand in dictionary order, it returns **false**. If the two operands have the same sequence of characters, it returns **true**.

3.4.6 The Operator >=

Integer and float If the left operand is greater than or equal to the right operand, > returns **true**. If the right operand is greater than the left operand, it returns **false**.

String If the left operand comes after the right operand in dictionary order, < returns **true**. If the left operand comes before the right operand in dictionary order, it returns **false**. If the two operands have the same sequence of characters, it returns **true**. `iiiii` HEAD

3.4.7 The Operator +

Integer and Float + returns the sum of the two operands.

String + returns the concatenation of the right operand onto the end of the left operand.

3.4.8 The Operator -

Integer and Float - returns the right operand subtracted from the left operand.

3.4.9 The Operator *

Integer and Float * returns the product of the two operands.

3.4.10 The Operator /

Integer and Float / returns the left operand divided by the right operand.

3.4.11 The Operator %

Integer and Float % returns the modulo of the left operand by the right operand.

3.4.12 The Operator `^`

Integer and Float `^` returns the left operand raised to the power of the right operand.

3.4.13 The Operator `:=`

Integer, Float, Boolean, and String `:=` assigns the right operand to the left operand and returns the value of the the right operand. This is the sole right precedence operator.

3.4.14 The Operators `+=`, `-=`, `*=`, `/=`, `%=`, and `^=`

Integer, Float, Boolean, and String This set of operators first applies the operator indicated by the first character of each operator as normal on the operands. It then assigns this value to its left operand.

3.4.15 The Operator `and`

Boolean `and` returns the conjunction of the operands.

3.4.16 The Operator `or`

Boolean `or` returns the disjunction of the operands.

3.4.17 The Operator `not`

Boolean `not` returns the negation of the operands.

3.4.18 The Operator `nand`

Boolean `nand` returns the negation of the conjunction of the operands.

3.4.19 The Operator `nor`

Boolean `nor` returns the negation of the disjunction of the operands.

3.4.20 The Operator `xor`

Boolean `xor` returns the exclusive disjunction of the operands.

3.4.21 The Operator `refinable`

Boolean `refinable` returns `true` if the refinement is implemented in the current subclass. It returns `false` otherwise.

3.5 Grammar

The following conventions are taken:

- Sequential semicolons (even separated by whitespace) are treated as one.

- the ‘digit’ class of characters are the numerical digits zero through nine
- the ‘upper’ class of characters are the upper case roman letters
- the ‘lower’ class of characters are the lower case roman letters
- the ‘ualphanum’ class of characters consists of the digit, upper, and lower classes together with the underscore
- a program is a collection of classes; this grammar describes solely classes
- the argument to main is semantically enforced after parsing; its presence here is meant to increase readability

The grammar follows:

-
- *Classes may extend another class or default to extending Object*

$\langle \text{class} \rangle \Rightarrow$
class $\langle \text{class id} \rangle \langle \text{extend} \rangle : \langle \text{class section} \rangle^*$
 $\langle \text{extend} \rangle \Rightarrow$
 ϵ
 | **extends** $\langle \text{class id} \rangle$

- *Sections – private protected public refinements and main*

$\langle \text{class section} \rangle \Rightarrow$
 $\langle \text{refinement} \rangle$
 | $\langle \text{access group} \rangle$
 | $\langle \text{main} \rangle$

- *Refinements are named method dot refinement*

$\langle \text{refinement} \rangle \Rightarrow$
refinement $\langle \text{refine} \rangle^*$
 $\langle \text{refine} \rangle \Rightarrow$
 $\langle \text{return type} \rangle \langle \text{var id} \rangle . \langle \text{var id} \rangle \langle \text{params} \rangle : \langle \text{statement} \rangle^*$

- *Access groups contain all the members of a class*

$\langle \text{access group} \rangle \Rightarrow$
 $\langle \text{access type} \rangle : \langle \text{member} \rangle^*$
 $\langle \text{access type} \rangle \Rightarrow$
private
 | **protected**
 | **public**
 $\langle \text{member} \rangle \Rightarrow$
 $\langle \text{var decl} \rangle$
 | $\langle \text{method} \rangle$
 | $\langle \text{init} \rangle$
 $\langle \text{method} \rangle \Rightarrow$
 $\langle \text{return type} \rangle \langle \text{var id} \rangle \langle \text{params} \rangle : \langle \text{statement} \rangle^*$
 $\langle \text{init} \rangle \Rightarrow$
init $\langle \text{params} \rangle : \langle \text{statement} \rangle^*$

- *Main is special – not instance data starts execution*

$\langle \text{main} \rangle \Rightarrow$
main (**System** **system**, **String**[] $\langle \text{var id} \rangle$) : $\langle \text{statement} \rangle^*$

- *Finally the meat and potatoes*

$\langle \text{statement} \rangle \Rightarrow$
 $\langle \text{var decl} \rangle$
| $\langle \text{var decl} \rangle := \langle \text{expression} \rangle$
| $\langle \text{super} \rangle$
| $\langle \text{return} \rangle$
| $\langle \text{conditional} \rangle$
| $\langle \text{loop} \rangle$
| $\langle \text{expression} \rangle$

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

$\langle \text{super} \rangle \Rightarrow$
super $\langle \text{args} \rangle$

- *Methods yield values (or just exit for void/init/main)*

$\langle \text{return} \rangle \Rightarrow$
return
| **return** $\langle \text{expression} \rangle$

- *Basic control structures*

$\langle \text{conditional} \rangle \Rightarrow$
if ($\langle \text{expression} \rangle$) : $\langle \text{statement} \rangle^*$ **else** $\langle \text{else} \rangle \Rightarrow$
 ϵ
| $\langle \text{elseif} \rangle$ **else** : $\langle \text{statement} \rangle^*$
 $\langle \text{elseif} \rangle \Rightarrow$
 ϵ
| $\langle \text{elseif} \rangle$ **elsif** ($\langle \text{expression} \rangle$) : $\langle \text{statement} \rangle^*$
 $\langle \text{loop} \rangle \Rightarrow$
while ($\langle \text{expression} \rangle$) : $\langle \text{statement} \rangle^*$

- *Anything that can result in a value*

$\langle \text{expression} \rangle \Rightarrow$
 $\langle \text{assignment} \rangle$
| $\langle \text{invocation} \rangle$
| $\langle \text{field} \rangle$
| $\langle \text{var id} \rangle$
| $\langle \text{deref} \rangle$
| $\langle \text{arithmetic} \rangle$
| $\langle \text{test} \rangle$
| $\langle \text{instantiate} \rangle$
| $\langle \text{refine expr} \rangle$
| $\langle \text{literal} \rangle$
| ($\langle \text{expression} \rangle$)
| **this**

- *Assignment – putting one thing in another*

$\langle \text{assignment} \rangle \Rightarrow$
 $\langle \text{expression} \rangle \langle \text{assign op} \rangle \langle \text{expression} \rangle$
 $\langle \text{assign op} \rangle \Rightarrow$
 :=
 $| \text{ +=}$
 $| \text{ -=}$
 $| \text{ *=}$
 $| \text{ /=}$
 $| \text{ \%=}$
 $| \text{ ^=}$

- *Member / data access*

$\langle \text{invocation} \rangle \Rightarrow$
 $\langle \text{expression} \rangle . \langle \text{var id} \rangle \langle \text{args} \rangle$
 $| \langle \text{var id} \rangle \langle \text{args} \rangle$
 $\langle \text{field} \rangle \Rightarrow$
 $\langle \text{expression} \rangle . \langle \text{var id} \rangle$
 $\langle \text{deref} \rangle \Rightarrow$
 $\langle \text{expression} \rangle [\langle \text{expression} \rangle]$

- *Basic arithmetic can and will be done!*

$\langle \text{arithmetic} \rangle \Rightarrow$
 $\langle \text{expression} \rangle \langle \text{bin op} \rangle \langle \text{expression} \rangle$
 $| \langle \text{unary op} \rangle \langle \text{expression} \rangle$
 $\langle \text{bin op} \rangle \Rightarrow$
 $+$
 $| -$
 $| *$
 $| /$
 $| \%$
 $| ^$
 $\langle \text{unary op} \rangle \Rightarrow$
 $-$

- *Common boolean predicates*

$\langle \text{test} \rangle \Rightarrow$
 $\langle \text{expression} \rangle \langle \text{bin pred} \rangle \langle \text{expression} \rangle$
 $| \langle \text{unary pred} \rangle \langle \text{expression} \rangle$
 $| \text{refinable} (\langle \text{var id} \rangle)$
 $\langle \text{bin pred} \rangle \Rightarrow$
 and
 $| \text{or}$
 $| \text{xor}$
 $| \text{nand}$
 $| \text{nor}$
 $| <$
 $| <=$
 $| =$
 $| <>$
 $| /=$

| \geq
 | $>$
 $\langle \text{unary pred} \rangle \Rightarrow$
 not

- *Making something*

$\langle \text{instantiate} \rangle \Rightarrow$
 new $\langle \text{type} \rangle \langle \text{args} \rangle \langle \text{optional refinements} \rangle$
 $\langle \text{optional refinements} \rangle \Rightarrow$
 ϵ
 | **{** $\langle \text{refine} \rangle^*$ **}**

- *Refinement takes a specialization and notes the required return type*

$\langle \text{refine expr} \rangle \Rightarrow$
 refine $\langle \text{var id} \rangle \langle \text{args} \rangle$ **to** $\langle \text{type} \rangle$

- *Literally necessary*

$\langle \text{literal} \rangle \Rightarrow$
 $\langle \text{int lit} \rangle$
 | $\langle \text{bool lit} \rangle$
 | $\langle \text{float lit} \rangle$
 | $\langle \text{string lit} \rangle$
 $\langle \text{float lit} \rangle \Rightarrow$
 $\langle \text{digit} \rangle^+ \cdot \langle \text{digit} \rangle^+$
 $\langle \text{int lit} \rangle \Rightarrow$
 $\langle \text{digits} \rangle^+$
 $\langle \text{bool lit} \rangle \Rightarrow$
 true
 | **false**
 $\langle \text{string lit} \rangle \Rightarrow$
 “ $\langle \text{string escape seq} \rangle$ ”

- *Params and args are as expected*

$\langle \text{params} \rangle \Rightarrow$
 ()
 | ($\langle \text{paramlist} \rangle$)
 $\langle \text{paramlist} \rangle \Rightarrow$
 $\langle \text{var decl} \rangle$
 | $\langle \text{paramlist} \rangle$, $\langle \text{var decl} \rangle$
 $\langle \text{args} \rangle \Rightarrow$
 ()
 | ($\langle \text{arglist} \rangle$)
 $\langle \text{arglist} \rangle \Rightarrow$
 $\langle \text{expression} \rangle$
 | $\langle \text{arglist} \rangle$, $\langle \text{expression} \rangle$

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

$\langle \text{var decl} \rangle \Rightarrow$
 $\langle \text{type} \rangle \langle \text{var id} \rangle$
 $\langle \text{return type} \rangle \Rightarrow$

```

    void
    |  $\langle \text{type} \rangle$ 
 $\langle \text{type} \rangle \Rightarrow$ 
     $\langle \text{class id} \rangle$ 
    |  $\langle \text{type} \rangle []$ 
 $\langle \text{class id} \rangle \Rightarrow$ 
     $\langle \text{upper} \rangle \langle \text{ualphanum} \rangle^*$ 
 $\langle \text{var id} \rangle \Rightarrow$ 
     $\langle \text{lower} \rangle \langle \text{ualphanum} \rangle^*$ 

```

4 Project Planning

4.1 Planning Techniques

The vast majority of all planning happened over a combination of email and google hangouts. The team experimented with a variety of communication methods. We found some success with using Glip late in our process. Zoho docs and google docs were also used without major utility.

The specification of new elements was routinely proposed via an email to all members with an example of the concept and a description of the concepts involved behind it. This proved surprisingly effective at achieving a consensus.

Development was heavily facilitated through the use of a shared git repository. Topical google hangouts would be started involving all members. Team members would describe what they were working on with the immediate tasks. Any given team member could only afford to work at the same time as any one other generally, so conflicts over work were rare.

Testing suites were developed concurrently with code. Given the well-traversed nature of object oriented programming, the necessary tests were fairly obvious.

4.2 Ocaml Style Guide for the Development of the Ray Compiler

Expert Ocaml technique is not expected for the development of ray, however there are some basic stylistic tendencies that are preferred at all times.

All indentation should be increments of four spaces. Tabs and two space increment indentation are not acceptable.

```
1  let x = 2
2  let z =
3      let add5 a =
4          + a 5 in
5      add5 x
```

When constructing a `let...in` statement, the associated `in` must not be alone on the final line. For a large `let` statement that defines a variable, store the final operational call in a dummy variable and return that dummy. For all but the shortest right-hand sides of `let` statements, the right-hand side should be placed at increased indentation on the next line.

```
1  let get_x =
2      ...
3      let n = 2 in
4      let x =
5          x_functor1 (x_functor2 y z) n in
6      x
```

`match` statements should always include a `|` for the first item. The `|` operators that are used should have aligned indentation, as should `->` operators, functors that follow such operators and comments. Exceedingly long functors should be placed at increased indentaiton on the next line. (These rules also apply to `type` definitions.)

```

1  let unify_it var =
2    match var with
3    | X(y)      -> y (* pop out *)
4    | Y(y) :: _ -> to_X y (* convert *)
5    | Z(y)      ->
6      to_X (to_Y (List.hd (List.rest y))) (* mangle *)

```

All records should maintain a basic standard of alignment and indentation for readability. (Field names, colons, and type specs should all be aligned to like.)

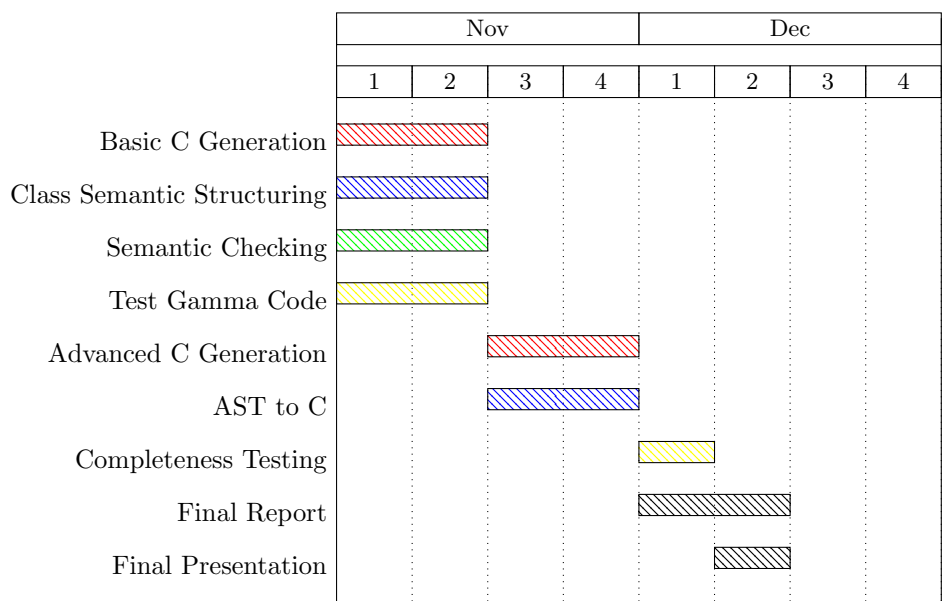
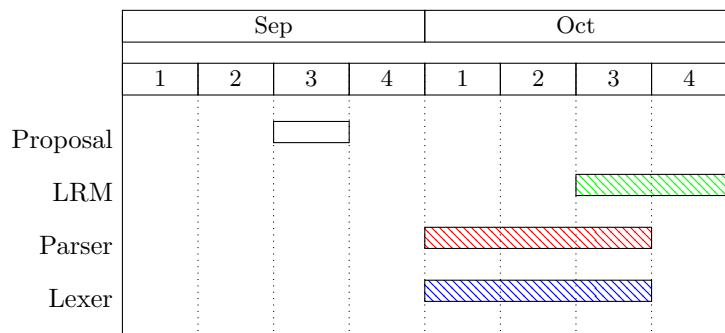
```

1  type person = {
2    names  : string list;
3    job    : string option; (* Not everybody has one *)
4    family : person list;
5    female : bool;
6    age    : int;
7  }

```

4.3 Project Timeline

The following gantt charts show the intended project timeline broken down by weeks of the four months of this semester. The loose units were intended to make our schedules more workable.



4.4 Team Roles

Ben Caimano

- Primary Documentation Officer
- Co-Organizer
- Parser Contributor
- Cast/C Contributor

Weiyuan Li

- Lexer Contributor
- Sast Contributor
- Cast/C Contributor
- Test Suite Contributor

Mathew H. Maycock

- Programming Lead
- Grammar Designer
- Quality Assurance Officer
- Lt. Documentation Officer
- Parser Contributor
- Sast Contributor
- Cast/C Contributor
- Test Suite Contributor

Arthy Sundaram

- Co-Organizer/President
- Parser Contributor
- Sast Contributor
- Cast/C Contributor
- Test Suite Contributor

4.5 Development Environment

4.5.1 Programming Languages

All Gamma code is compiled by the ray compiler to an intermediary file of C (ANSI ISO C90) code which is subsequently compiled to a binary file. Lexographical scanning, semantic parsing and checking, and compilation to C is all done by custom-written code in Ocaml 4.01.

The Ocaml code is compiled using the Ocaml bytecode compiler (`ocamlc`), the Ocaml parser generator (`ocamlyacc`), and the Ocaml lexer generator (`ocamllex`). Incidentally, documentation of the Ocaml code for internal use is done using the Ocaml documentation generator (`ocamldoc`). The compilation from intermediary C to bytecode is done using the GNU project C and C++ compiler (GCC) 4.7.3.

Scripting of our Ocaml compilation and other useful command-level tasks is done through a combination of the GNU make utility (a Makefile) and the dash command interpreter (shell scripts).

4.5.2 Development Tools

Our development tools were minimalistic. Each team member had a code editor of choice (emacs, vim, etc.). Content management and collaboration was done via git. Our git repository was hosted on BitBucket by Atlassian Inc. The ocaml interpreter shell was used for testing purposes, as was a large suite of testing utilities written in ocaml for the task. Among these created tools were:

- `canonical` - Takes an input stream of brace-style code and outputs the whitespace-style equivalent
- `cannonize` - Takes an input stream of whitespace-style code and outputs the brace-style equivalent
- `classinfo` - Analyzes the defined members (methods and variables) for a given class
- `freevars` - Lists the variables that remain unbound in the program
- `inspect` - Stringify a given AST
- `prettify` - Same as above but with formatting
- `streams` - Check a whitespace-style source for formatting issues

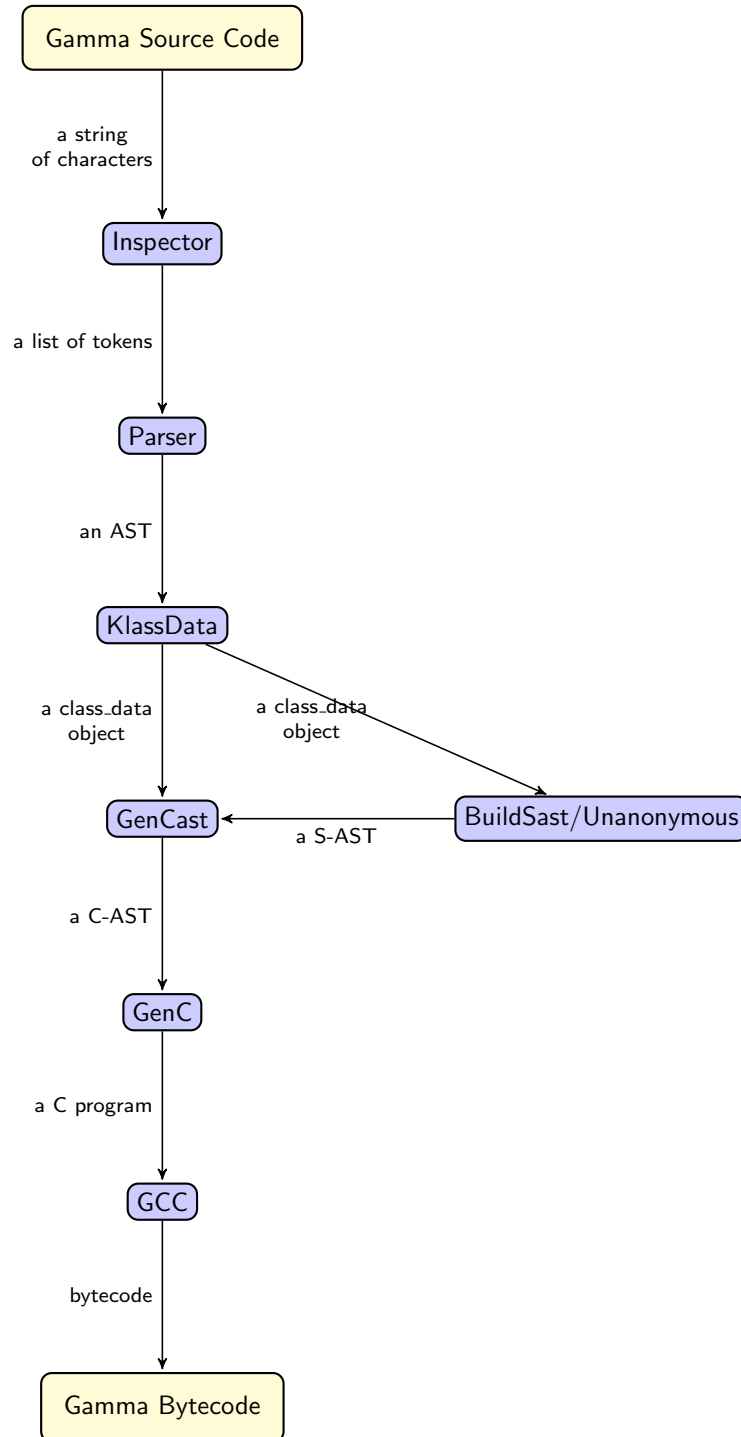
4.6 Project Log

- September 9th - Team Formed
- September 18th - Proposal drafting begins
- September 19th - A consensus is reached, basic form of the language is hashed out as a Beta-derived object oriented language.
- September 24-25th - Propose written, language essentials described
- October 9-10th - Grammar written
- October 18-20th - Bulk of the lexer/parser is written
- October 24th - Inspector written
- October 26th - Parser officially compiled for first time
- October 29th - Language resource manual finished, language structure semi-rigidly defined
- November 11th - General schedule set, promptly falls apart under the mutual stress of projects and midterms
- November 24th - Class data collection implemented
- November 30th - SAST structure defined
- December 8-10th - Team drama happens
- December 10th - SAST generation code written
- December 12th - CAST and CAST generation begun
- December 14th - C generation development started
- December 15th - Approximate CAST generation written
- December 16th - First ray binary made
- December 19th - Ray compilation of basic code successful
- December 22nd - Ray passes the test suite

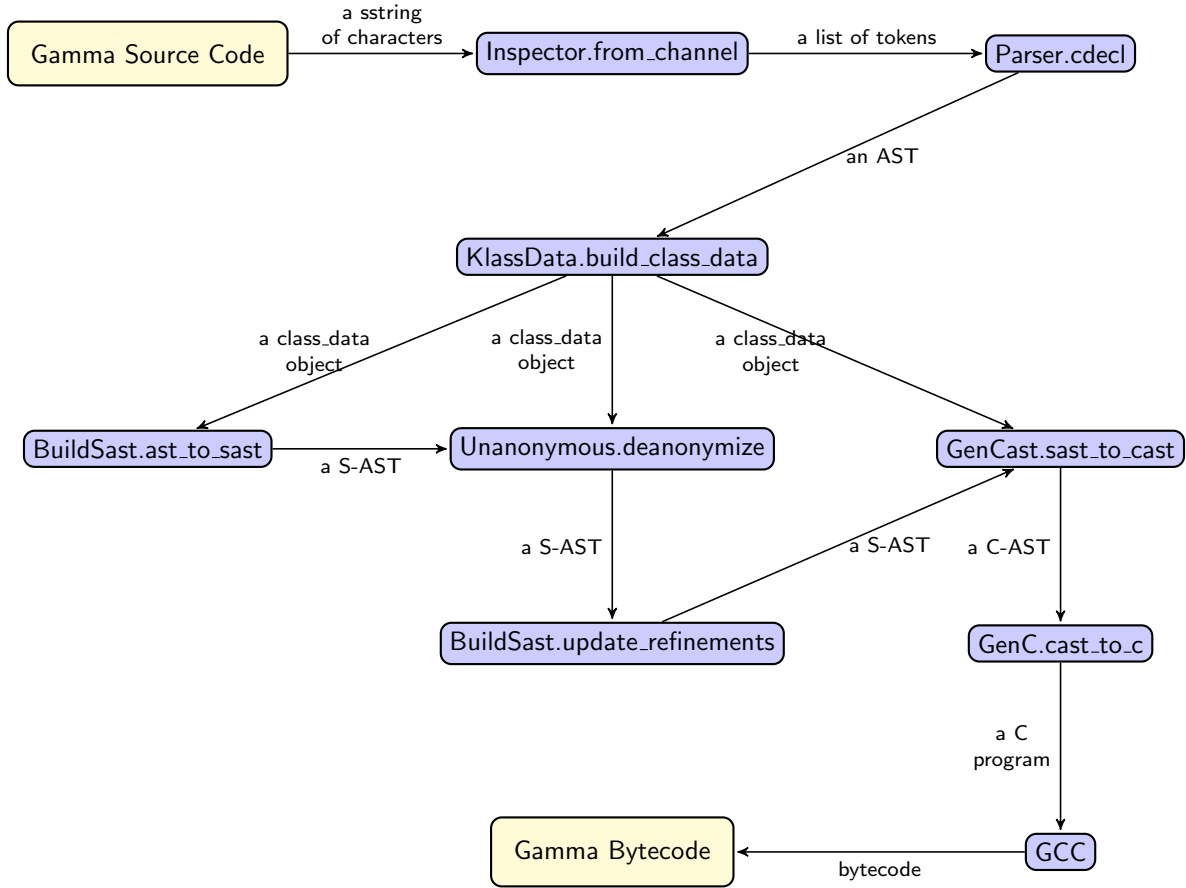
5 Architectural Design

5.1 Block Diagrams

5.1.1 Structure by Module



5.1.2 Structure by Toplevel Ocaml Function



5.2 Component Connective Interfaces

```

let get_data ast =
  let (which, builder) = if (Array.length Sys.argv <= 2)
    then ("Normal", KlassData.build_class_data)
    else ("Experimental", KlassData.build_class_data_test) in
  output_string (Format.sprintf " * Using %s KlassData Builder" which);
  match builder ast with
  | Left(data) -> data
  | Right(issue) -> Printf.fprintf stderr "%s\n" (KlassData.errstr issue); exit 1

let do_deanon klass_data sast = match Unanonymous.deanonymize klass_data sast with
  | Left(result) -> result
  | Right(issue) -> Printf.fprintf stderr "Error Deanonymizing:\n%s\n" (KlassData.errstr issue); exit 1

let source_cast _ =
  output_string " * Reading Tokens...";
  let tokens = with_file Inspector.from_channel Sys.argv.(1) in
  output_string " * Parsing Tokens...";
  let ast = Parser.cdecls (WhiteSpace.lextoks tokens) (Lexing.from_string "") in
  output_string " * Generating Global Data...";
  let klass_data = get_data ast in

```

```

output_string " * Building Semantic AST...";
let sast = BuildSast.ast_to_sast klass_data in
output_string " * Deanonymizing Anonymous Classes.";
let (klass_data, sast) = do_deanon klass_data sast in
output_string " * Rebinding refinements.";
let sast = BuildSast.update_refinements klass_data sast in
output_string " * Generating C AST...";
GenCast.sast_to_cast klass_data sast

let main _ =
  Printexc.record_backtrace true;
  output_string "/* Starting Build Process...";
  try
    let source = source_cast () in
    output_string " * Generating C...";
    output_string " */";
    GenC.cast_to_c source stdout;
    print_newline ();
    exit 0
  with excn ->
    let backtrace = Printexc.get_backtrace () in
    let reraise = ref false in
    let out = match excn with
      | Failure(reason) -> Format.sprintf "Failed: %s\n" reason
      | Invalid_argument(msg) -> Format.sprintf "Argument issue somewhere: %s\n"
msg
      | Parsing.Parse_error -> "Parsing error."
      | _ -> reraise := true; "Unknown Exception" in
    Printf.fprintf stderr "%s\n%s\n" out backtrace;
    if !reraise then raise(excn) else exit 1

```

Example 13: The Main Ray Compiler Ocaml (Trimmed)

The primary functionality of the compiler is collected into convenient ocaml modules. From the lexer to the C-AST to C conversion, the connections are the passing of data representations of the current step to the main function of the following module. We utilize as data representations three ASTs (basic, semantic, and C-oriented), a more searchable tabulation of class data, and, of course, a source string and a list of tokens. The presence of Anonymous classes complicates the building of the array of class data and the sast as can be seen by the functor `do_deanon`. Our testing experiences also lead to a more verbose form of AST generation for experimental features, hence `get_data`. In all other cases, the result of the previous step is simply stored in a variable by `let` and passed to the next step. The output of ray is a C file. The user must manually do the final step of compiling this file to bytecode using GCC.

5.3 Component Authorship

Each component was a combined effort. This is expressed somewhat in the project role section. However, for clarity, it will be reexpressed in terms of the module architecture above:

- Inspector - Weiyuan/Arthy
- Parser - Ben/Arthy/Matthew
- KlassData - Matthew
- Unanonymouse - Matthew
- BuildSast - Matthew/Weiyuan/Arthy
- GenCast - Matthew/Weiyuan/Ben/Arthy

- GenC - Matthew/Weiyuan/Ben/Arthy
- GCC - GNU

6 Test Plan

6.1 Examples Gamma Programs

6.1.1 Hello World

This program simply prints "Hello World". It demonstrates the fundamentals needed to write a Gamma program.

```
1  class HelloWorld:
2      public:
3          String greeting
4          init():
5              super()
6              greeting := "Hello World!"
7
8      main(System system, String[] args):
9          HelloWorld hw := new HelloWorld()
10         system.out.printString(hw.greeting)
11         system.out.printString("\n")
```

Example 14: "Hello World in Gamma"

```
1  /* Starting Build Process...
2     * Reading Tokens...
3     * Parsing Tokens...
4     * Generating Global Data...
5     * Using Normal KlassData Builder
6     * Building Semantic AST...
7     * Deanonymizing Anonymous Classes.
8     * Rebinding refinements.
9     * Generating C AST...
10     * Generating C...
11     */
12
13
14     /*
15     * Passing over code to find dispatch data.
16     */
17
18
19     /*
20     * Gamma preamble — macros and such needed by various things
21     */
22     #include "gamma-preamble.h"
23
24
25
26     /*
27     * Ancestry meta-info to link to later.
28     */
29     char *m_classes[] = {
30         "t_Boolean", "t_Float", "t_HelloWorld", "t_Integer", "t_Object", "t_Printer",
31         "t_Scanner", "t_String", "t_System"
32     };
33
34
35     /*
36     * Enums used to reference into ancestry meta-info strings.
37     */
```

```

38 enum m_class_idx {
39     T_BOOLEAN = 0, T_FLOAT, T_HELLOWORLD, T_INTEGER, T_OBJECT, T_PRINTER, T_SCANNER,
40     T_STRING, T_SYSTEM
41 };
42
43
44 /*
45  * Header file containing meta information for built in classes.
46  */
47 #include "gamma-builtin-meta.h"
48
49
50
51 /*
52  * Meta structures for each class.
53  */
54 ClassInfo M_HelloWorld;
55
56 void init_class_infos() {
57     init_built_in_infos();
58     class_info_init(&M_HelloWorld, 2, m_classes[T_OBJECT], m_classes[T_HELLOWORLD]);
59 }
60
61
62
63 /*
64  * Header file containing structure information for built in classes.
65  */
66 #include "gamma-builtin-struct.h"
67
68
69
70 /*
71  * Structures for each of the objects.
72  */
73 struct t_HelloWorld {
74     ClassInfo *meta;
75
76     struct {
77         struct t_System *v_system;
78     } Object;
79
80     struct {
81         struct t_String *v_greeting;
82     } HelloWorld;
83 };
84
85 };
86
87
88
89
90 /*
91  * Header file containing information regarding built in functions.
92  */
93 #include "gamma-builtin-functions.h"
94
95
96
97 /*
98  * All of the function prototypes we need to do magic.
99  */
100 struct t_HelloWorld *f_00000001_init(struct t_HelloWorld *);
101 void f_00000002_main(struct t_System *, struct t_String **);
102

```



```

103
104 /*
105  * All the dispatching functions we need to continue the magic.
106  */
107
108
109 /*
110  * Array allocators also do magic.
111  */
112
113
114 /*
115  * All of the functions we need to run the program.
116  */
117 /* Place-holder for struct t_Boolean *boolean_init(struct t_Boolean *this) */
118 /* Place-holder for struct t_Float *float_init(struct t_Float *this) */
119 /* Place-holder for struct t_Integer *float_to_i(struct t_Float *this) */
120 /* Place-holder for struct t_Integer *integer_init(struct t_Integer *this) */
121 /* Place-holder for struct t_Float *integer_to_f(struct t_Integer *this) */
122 /* Place-holder for struct t_Object *object_init(struct t_Object *this) */
123 /* Place-holder for struct t_Printer *printer_init(struct t_Printer *this, struct
124    t_Boolean *v_stdout) */
125 /* Place-holder for void printer_print_float(struct t_Printer *this, struct t_Float *
126    v_arg) */
127 /* Place-holder for void printer_print_integer(struct t_Printer *this, struct t_Integer *
128    v_arg) */
129 /* Place-holder for void printer_print_string(struct t_Printer *this, struct t_String *
130    v_arg) */
131 /* Place-holder for struct t_Scanner *scanner_init(struct t_Scanner *this) */
132 /* Place-holder for struct t_Float *scanner_scan_float(struct t_Scanner *this) */
133 /* Place-holder for struct t_Integer *scanner_scan_integer(struct t_Scanner *this) */
134 /* Place-holder for struct t_String *scanner_scan_string(struct t_Scanner *this) */
135 /* Place-holder for struct t_String *string_init(struct t_String *this) */
136 /* Place-holder for void system_exit(struct t_System *this, struct t_Integer *v_code) */
137 /* Place-holder for struct t_System *system_init(struct t_System *this) */
138
139 struct t_HelloWorld *f_00000001_init(struct t_HelloWorld *this)
140 {
141     object_init((struct t_Object *) (this));
142     ( (this->HelloWorld).v_greeting = ((struct t_String *) (LIT_STRING(" Hello World!"))) )
143     ;
144     return ( this );
145 }
146
147 void f_00000002_main(struct t_System *v_system, struct t_String **v_args)
148 {
149     struct t_HelloWorld *v_hw = ((struct t_HelloWorld *) (f_00000001_init(MAKE_NEW(
150     HelloWorld)))));
151     ( printer_print_string(((struct t_Printer *) ((v_system)->System.v_out)), (v_hw)->
152     HelloWorld.v_greeting) );
153     ( printer_print_string(((struct t_Printer *) ((v_system)->System.v_out)), LIT_STRING("
154     \n"))) );
155 }
156
157 /*
158  * Dispatch looks like this.
159  */
160
161 /*
162  * Array allocators.
163  */

```

```

160
161
162  /*
163  * The main.
164  */
165  #define CASES "HelloWorld"
166
167  int main(int argc, char **argv) {
168      INIT_MAIN(CASES)
169      if (!strcmp(gmain, "HelloWorld", 11)) { f_00000002_main(&global_system, str_args);
170          return 0; }
171      FAIL_MAIN(CASES)
172      return 1;
173  }

```

Example 15: "Hello World in Compiled C"

6.1.2 I/O

This program prompts the user for an integer and a float. It converts the integer to a float and adds the two together. It then prints the equation and result. (You might recognize this from the tutorial.)

```

1  class IOTest:
2      public:
3          init():
4              super()
5
6          void interact():
7              Printer p := system.out
8              Integer i := promptInteger("Please enter an integer")
9              Float f := promptFloat("Please enter a float")
10             p.printString("Sum of integer + float = ")
11             p.printFloat(i.toFloat() + f)
12             p.printString("\n")
13
14         private:
15             void prompt(String msg):
16                 system.out.printString(msg)
17                 system.out.printString(": ")
18
19             Integer promptInteger(String msg):
20                 prompt(msg)
21                 return system.in.scanInteger()
22
23             Float promptFloat(String msg):
24                 prompt(msg)
25                 return system.in.scanFloat()
26
27         main(System system, String[] args):
28             IOTest test := new IOTest()
29             test.interact()

```

Example 16: "I/O in Gamma"

```

1  /* Starting Build Process...
2  * Reading Tokens...
3  * Parsing Tokens...
4  * Generating Global Data...
5  * Using Normal KlassData Builder

```

```

6  * Building Semantic AST...
7  * Deanonymizing Anonymous Classes.
8  * Rebinding refinements.
9  * Generating C AST...
10 * Generating C...
11 */
12
13
14 /*
15  * Passing over code to find dispatch data.
16  */
17
18
19 /*
20  * Gamma preamble — macros and such needed by various things
21  */
22 #include "gamma-preamble.h"
23
24
25
26 /*
27  * Ancestry meta-info to link to later.
28  */
29 char *m_classes[] = {
30     "t_Boolean", "t_Float", "t_IOTest", "t_Integer", "t_Object", "t_Printer", "t_Scanner"
31     ,
32     "t_String", "t_System"
33 };
34
35 /*
36  * Enums used to reference into ancestry meta-info strings.
37  */
38 enum m_class_idx {
39     T_BOOLEAN = 0, T_FLOAT, T_IOTEST, T_INTEGER, T_OBJECT, T_PRINTER, T_SCANNER,
40     T_STRING, T_SYSTEM
41 };
42
43
44 /*
45  * Header file containing meta information for built in classes.
46  */
47 #include "gamma-builtin-meta.h"
48
49
50
51 /*
52  * Meta structures for each class.
53  */
54 ClassInfo M_IOTest;
55
56 void init_class_infos() {
57     init_built_in_infos();
58     class_info_init(&M_IOTest, 2, m_classes[T_OBJECT], m_classes[T_IOTEST]);
59 }
60
61
62
63 /*
64  * Header file containing structure information for built in classes.
65  */
66 #include "gamma-builtin-struct.h"
67
68
69

```

```

70  /*
71  * Structures for each of the objects.
72  */
73  struct t_IOTest {
74      ClassInfo *meta;
75
76      struct {
77          struct t_System *v_system;
78      } Object;
79
80
81      struct { BYTE empty_vars; } IOTest;
82  };
83
84
85
86
87  /*
88  * Header file containing information regarding built in functions.
89  */
90  #include "gamma-builtin-functions.h"
91
92
93
94  /*
95  * All of the function prototypes we need to do magic.
96  */
97  struct t_IOTest *f_00000001_init(struct t_IOTest *);
98  void f_00000002_interact(struct t_IOTest *);
99  void f_00000003_prompt(struct t_IOTest *, struct t_String *);
100 struct t_Integer *f_00000004_promptInteger(struct t_IOTest *, struct t_String *);
101 struct t_Float *f_00000005_promptFloat(struct t_IOTest *, struct t_String *);
102 void f_00000006_main(struct t_System *, struct t_String **);
103
104
105 /*
106 * All the dispatching functions we need to continue the magic.
107 */
108
109
110 /*
111 * Array allocators also do magic.
112 */
113
114
115 /*
116 * All of the functions we need to run the program.
117 */
118 /* Place-holder for struct t_Boolean *boolean_init(struct t_Boolean *this) */
119 /* Place-holder for struct t_Float *float_init(struct t_Float *this) */
120 /* Place-holder for struct t_Integer *float_to_i(struct t_Float *this) */
121 /* Place-holder for struct t_Integer *integer_init(struct t_Integer *this) */
122 /* Place-holder for struct t_Float *integer_to_f(struct t_Integer *this) */
123 /* Place-holder for struct t_Object *object_init(struct t_Object *this) */
124 /* Place-holder for struct t_Printer *printer_init(struct t_Printer *this, struct
    t_Boolean *v_stdout) */
125 /* Place-holder for void printer_print_float(struct t_Printer *this, struct t_Float *
    v_arg) */
126 /* Place-holder for void printer_print_integer(struct t_Printer *this, struct t_Integer *
    v_arg) */
127 /* Place-holder for void printer_print_string(struct t_Printer *this, struct t_String *
    v_arg) */
128 /* Place-holder for struct t_Scanner *scanner_init(struct t_Scanner *this) */
129 /* Place-holder for struct t_Float *scanner_scan_float(struct t_Scanner *this) */
130 /* Place-holder for struct t_Integer *scanner_scan_integer(struct t_Scanner *this) */

```

```

131 /* Place-holder for struct t_String *scanner_scan_string(struct t_Scanner *this) */
132 /* Place-holder for struct t_String *string_init(struct t_String *this) */
133 /* Place-holder for void system_exit(struct t_System *this, struct t_Integer *v_code) */
134 /* Place-holder for struct t_System *system_init(struct t_System *this) */
135
136 struct t_IOTest *f_00000001_init(struct t_IOTest *this)
137 {
138     object_init((struct t_Object *) (this));
139     return ( this );
140 }
141
142
143 void f_00000002_interact(struct t_IOTest *this)
144 {
145     struct t_Printer *v_p = ((struct t_Printer *) (((this->Object).v_system)->System.v_out
146 ));
147     struct t_Integer *v_i = ((struct t_Integer *) (f_00000004_promptInteger(((struct
148 t_IOTest *) (this)), LIT_STRING("Please enter an integer"))));
149     struct t_Float *v_f = ((struct t_Float *) (f_00000005_promptFloat(((struct t_IOTest *)
150 (this)), LIT_STRING("Please enter a float"))));
151     ( printer_print_string(((struct t_Printer *) (v_p)), LIT_STRING("Sum of integer +
152 float = ")) );
153     ( printer_print_float(((struct t_Printer *) (v_p)), ADD_FLOAT_FLOAT( integer_to_f(((
154 struct t_Integer *) (v_i))) , v_f )) );
155     ( printer_print_string(((struct t_Printer *) (v_p)), LIT_STRING("\n")) );
156 }
157
158 void f_00000003_prompt(struct t_IOTest *this, struct t_String *v_msg)
159 {
160     ( printer_print_string(((struct t_Printer *) (((this->Object).v_system)->System.v_out
161 ), v_msg) );
162     ( printer_print_string(((struct t_Printer *) (((this->Object).v_system)->System.v_out
163 ), LIT_STRING(": ") ) );
164 }
165
166 struct t_Integer *f_00000004_promptInteger(struct t_IOTest *this, struct t_String *v_msg)
167 {
168     ( f_00000003_prompt(((struct t_IOTest *) (this)), v_msg) );
169     return ( scanner_scan_integer(((struct t_Scanner *) (((this->Object).v_system)->System
170 .v_in))) );
171 }
172
173 struct t_Float *f_00000005_promptFloat(struct t_IOTest *this, struct t_String *v_msg)
174 {
175     ( f_00000003_prompt(((struct t_IOTest *) (this)), v_msg) );
176     return ( scanner_scan_float(((struct t_Scanner *) (((this->Object).v_system)->System.
177 v_in))) );
178 }
179
180 void f_00000006_main(struct t_System *v_system, struct t_String **v_args)
181 {
182     struct t_IOTest *v_test = ((struct t_IOTest *) (f_00000001_init(MAKENEW(IOTest))));
183     ( f_00000002_interact(((struct t_IOTest *) (v_test))) );
184 }
185
186 /*
187  * Dispatch looks like this.
188  */

```

```

187
188 /*
189  * Array allocators.
190  */
191
192
193 /*
194  * The main.
195  */
196 #define CASES "IOTest"
197
198 int main(int argc, char **argv) {
199     INIT_MAIN(CASES)
200     if (!strcmp(gmain, "IOTest", 7)) { f_00000006.main(&global_system, str_args); return
201         0; }
202     FAIL_MAIN(CASES)
203     return 1;
204 }

```

Example 17: "I/O in Compiled C"

6.1.3 Argument Reading

This program prints out each argument passed to the program.

```

1  class Test:
2      public:
3          init():
4              super()
5
6      main(System sys, String[] args):
7          Integer i := 0
8          Printer p := sys.out
9
10         while (i < sys.args):
11             p.printString("arg[")
12             p.printInteger(i)
13             p.printString("] = ")
14             p.printString(args[i])
15             p.printString("\n")
16             i += 1

```

Example 18: "Argument Reading in Gamma"

```

1  /* Starting Build Process...
2  * Reading Tokens...
3  * Parsing Tokens...
4  * Generating Global Data...
5  * Using Normal KlassData Builder
6  * Building Semantic AST...
7  * Deanonymizing Anonymous Classes.
8  * Rebinding refinements.
9  * Generating C AST...
10 * Generating C...
11 */
12
13
14 /*
15 * Passing over code to find dispatch data.

```

```

16  */
17
18
19  /*
20  * Gamma preamble — macros and such needed by various things
21  */
22  #include "gamma-preamble.h"
23
24
25
26  /*
27  * Ancestry meta-info to link to later.
28  */
29  char *m_classes[] = {
30      "t_Boolean", "t_Float", "t_Integer", "t_Object", "t_Printer", "t_Scanner",
31      "t_String", "t_System", "t_Test"
32  };
33
34
35  /*
36  * Enums used to reference into ancestry meta-info strings.
37  */
38  enum m_class_idx {
39      T_BOOLEAN = 0, T_FLOAT, T_INTEGER, T_OBJECT, T_PRINTER, T_SCANNER, T_STRING,
40      T_SYSTEM, T_TEST
41  };
42
43
44  /*
45  * Header file containing meta information for built in classes.
46  */
47  #include "gamma-builtin-meta.h"
48
49
50
51  /*
52  * Meta structures for each class.
53  */
54  ClassInfo M_Test;
55
56  void init_class_infos() {
57      init_built_in_infos();
58      class_info_init(&M_Test, 2, m_classes[T_OBJECT], m_classes[T_TEST]);
59  }
60
61
62
63  /*
64  * Header file containing structure information for built in classes.
65  */
66  #include "gamma-builtin-struct.h"
67
68
69
70  /*
71  * Structures for each of the objects.
72  */
73  struct t_Test {
74      ClassInfo *meta;
75
76      struct {
77          struct t_System *v_system;
78      } Object;
79
80

```

```

81     struct { BYTE empty-vars; } Test;
82 };
83
84
85
86
87 /*
88  * Header file containing information regarding built in functions.
89  */
90 #include "gamma-builtin-functions.h"
91
92
93
94 /*
95  * All of the function prototypes we need to do magic.
96  */
97 struct t_Test *f_00000001_init(struct t_Test *);
98 void f_00000002_main(struct t_System *, struct t_String **);
99
100
101 /*
102  * All the dispatching functions we need to continue the magic.
103  */
104
105
106 /*
107  * Array allocators also do magic.
108  */
109
110
111 /*
112  * All of the functions we need to run the program.
113  */
114 /* Place-holder for struct t_Boolean *boolean_init(struct t_Boolean *this) */
115 /* Place-holder for struct t_Float *float_init(struct t_Float *this) */
116 /* Place-holder for struct t_Integer *float_to_i(struct t_Float *this) */
117 /* Place-holder for struct t_Integer *integer_init(struct t_Integer *this) */
118 /* Place-holder for struct t_Float *integer_to_f(struct t_Integer *this) */
119 /* Place-holder for struct t_Object *object_init(struct t_Object *this) */
120 /* Place-holder for struct t_Printer *printer_init(struct t_Printer *this, struct
    t_Boolean *v_stdout) */
121 /* Place-holder for void printer_print_float(struct t_Printer *this, struct t_Float *
    v_arg) */
122 /* Place-holder for void printer_print_integer(struct t_Printer *this, struct t_Integer *
    v_arg) */
123 /* Place-holder for void printer_print_string(struct t_Printer *this, struct t_String *
    v_arg) */
124 /* Place-holder for struct t_Scanner *scanner_init(struct t_Scanner *this) */
125 /* Place-holder for struct t_Float *scanner_scan_float(struct t_Scanner *this) */
126 /* Place-holder for struct t_Integer *scanner_scan_integer(struct t_Scanner *this) */
127 /* Place-holder for struct t_String *scanner_scan_string(struct t_Scanner *this) */
128 /* Place-holder for struct t_String *string_init(struct t_String *this) */
129 /* Place-holder for void system_exit(struct t_System *this, struct t_Integer *v_code) */
130 /* Place-holder for struct t_System *system_init(struct t_System *this) */
131
132 struct t_Test *f_00000001_init(struct t_Test *this)
133 {
134     object_init((struct t_Object *) (this));
135     return ( this );
136 }
137
138
139 void f_00000002_main(struct t_System *v_sys, struct t_String **v_args)
140 {
141     struct t_Integer *v_i = ((struct t_Integer *) (LIT_INT(0)));

```



```

142     struct t_Printer *v_p = ((struct t_Printer *)((v_sys)->System.v_out));
143     while ( BOOLOF( NTEST_LESS_INT_INT( v_i , (v_sys)->System.v_argc ) ) ) {
144         ( printer_print_string(((struct t_Printer *) (v_p)), LIT_STRING("arg[")) );
145         ( printer_print_integer(((struct t_Printer *) (v_p)), v_i) );
146         ( printer_print_string(((struct t_Printer *) (v_p)), LIT_STRING("] = ") ) );
147         ( printer_print_string(((struct t_Printer *) (v_p)), ((struct t_String **)(v_args)
148         ) [INTEGER_OF((v_i))]) );
149         ( printer_print_string(((struct t_Printer *) (v_p)), LIT_STRING("\n")) );
150         ( v_i = ((struct t_Integer *) (ADD_INT_INT( v_i , LIT_INT(1) )) ) );
151     }
152
153
154
155     /*
156     * Dispatch looks like this.
157     */
158
159
160     /*
161     * Array allocators.
162     */
163
164
165     /*
166     * The main.
167     */
168     #define CASES "Test"
169
170     int main(int argc, char **argv) {
171         INIT_MAIN(CASES)
172         if (!strcmp(gmain, "Test", 5)) { f_00000002_main(&global_system, str_args); return
173         0; }
174         FAIL_MAIN(CASES)
175         return 1;
176     }

```

Example 19: "Argument Reading in Compiled C"

6.2 Test Suites

All tests suites involved Gamma source code that was compiled through ray and GCC to check for desired functionality. This was done as a communal effort towards the end of the project.

6.2.1 Desired Failure Testing

This suite of tests made sure that bad code did not compile.

```
1  class Parent:
2      public:
3          init():
4              super()
5
6  class Child extends Parent:
7      public:
8          init():
9              super()
10
11 class Test:
12     public:
13         init():
14             super()
15
16     main(System system, String[] args):
17         Child child := new Parent()
```

Test Source 1: "Superclass Typed to Subclass"

While a subclass can be stored in a variable typed to its parent, the reverse should not be possible.

```
1  class BadDecl:
2      public:
3          init():
4              super()
5              Integer a := 3.4
```

Test Source 2: "Improper Variable Declaration/Assignment"

A Float should never be allowed to be stored in an Integer variable.

```
1  class Test:
2      public:
3          Float a
4          Float b
5          Integer c
6
7          init():
8              super()
9              a := 1.5
10             b := 2.2
11             c := 3
12
13         Float overview():
14             Float success := a+b+c
15             return success
16
17     main(System system, String[] args):
18         Test ab := new Test()
```

```

19     Printer p := system.out
20     p.printString("Sum of integer = ")
21     p.printFloat(ab.overview())
22     p.printString("\n")

```

Test Source 3: "Binary Operations Between Incompatible Types"

A Float should not be allowed to be added to an Integer.

```

1  class BadReturn:
2      public:
3          init():
4              super()
5
6          Integer badReturn():
7              return "Hey There"

```

Test Source 4: "Return Variable of the Wrong Type"

It is not allowed for a function to return a variable of a different type than its declared return type.

```

1  class BadReturn:
2      public:
3          init():
4              super()
5
6          Integer badReturn():
7              return

```

Test Source 5: "Empty Return Statement"

A return statement should return something.

```

1  class BadReturn:
2      public:
3          init():
4              super()
5
6          void badReturn():
7              return "Hey There"

```

Test Source 6: "Return Statement in a Void Method"

A method with a return type of void should have no return statement.

```

1  class BadAssign:
2      public:
3          init():
4              super()
5              Integer a
6              a := 3.4

```

Test Source 7: "Improper Literal Assignment"

A literal object cannot be assigned to a variable of the wrong type.

```

1 class BadStatic:
2   public:
3     Integer getZero():
4       return 0
5     init():
6       super()
7   main(System system, String[] args):
8     getZero() /* This is supposed to fail. DON'T CHANGE */

```

Test Source 8: "Static Method Calls"

A method must be called on an object.

```

1 class Parent:
2   public:
3     Integer a
4     Integer b
5     Integer c
6
7     init():
8       super()
9       a := 1
10      b := 2
11      c := 0
12
13     Integer overview():
14       Integer success := refine toExtra(a,b) to Integer
15       return success
16
17 class Child extends Parent:
18   refinement:
19     Integer overview.toExtra(Integer a, Integer b):
20       Integer success := a + b
21       Printer p := new Printer(true)
22       p.printInteger(a)
23       p.printInteger(b)
24       p.printInteger(c)
25       return success
26   public:
27     Integer a1
28     Integer b1
29     Integer c1
30
31     init():
32       super()
33       a1 := 1
34       b1 := 2
35       c1 := 0
36
37 class Test:
38   public:
39     init():
40       super()
41
42   main(System system, String[] args):
43     Parent ab := new Parent
44     Printer p := system.out
45     p.printString("Sum of integer = ")
46     p.printInteger(ab.overview())
47     p.printString("\n")

```

Test Source 9: "Unimplemented Refinement"

A method that has a refinement must be called from a subclass of the original class that implements the refinement.

```

1  class Parent:
2      public:
3          Integer a
4          Integer b
5          Integer c
6
7      init():
8          super()
9          a := 1
10         b := 2
11         c := 0
12
13     Integer overview():
14         Integer success := -1
15         if (refinable(toExtra)) {
16             success := refine toExtra(a,b) to Integer;
17         }
18         return success
19
20 class Child extends Parent:
21     refinement:
22         Integer overview.toExtra(Integer a, Integer b):
23             Integer success := a + b
24             Printer p := new Printer(true)
25             p.printInteger(a)
26             p.printInteger(b)
27             p.printInteger(c)
28             return success
29     public:
30         Integer a1
31         Integer b1
32         Integer c1
33
34     init():
35         super()
36         a1 := 1
37         b1 := 2
38         c1 := 0
39
40 class Test:
41     public:
42         init():
43             super()
44
45     main(System system, String[] args):
46         Parent ab := new Parent()
47         Printer p := system.out
48         p.printString("Sum of integer = ")
49         p.printInteger(ab.overview())
50         p.printString("\n")

```

Test Source 10: "unimplemented Refinement with Refinable"

This case uses refinable to avoid paths with unimplemented refinements. It should function.

6.2.2 Statement Testing

This suite of test case makes sure that basic statements do compile.

```

1
2 class WhileLoopTest:
3     public:
4         init():
5             super()
6             Integer a := 0
7             while((a>=0) and (a<10)):
8                 system.out.printInteger(a)
9                 system.out.printString("\n")
10                a := a + 1
11
12 main(System system, String [] args):
13     new WhileLoopTest()

```

Test Source 11: "Conditioned While Statements"

This test makes sure while loops function.

```

1
2 class WhileLoopTest:
3     public:
4         init():
5             super()
6             Integer a := 0
7             while(true):
8                 system.out.printInteger(a)
9                 system.out.printString("\n")
10                a := a + 1
11
12 main(System system, String [] args):
13     new WhileLoopTest()

```

Test Source 12: "Infinite While Statement"

This test makes sure that while loops can continue within the bounds of memory.

```

1 class IfTest:
2     private:
3         void line():
4             system.out.printString("\n")
5
6         void out(String msg):
7             system.out.printString(msg)
8             line()
9
10        void yes():
11            out("This should print.")
12        void no():
13            out("This should not print.")
14
15    public:
16        init():
17            super()
18
19            out("Simple (1/2)")
20            if (true) { yes(); }
21            if (false) { no(); }
22            line()
23
24            out("Basic (2/2)")
25            if (true) { yes(); } else { no(); }

```

```

26     if (false) { no(); } else { yes(); }
27     line()
28
29     out("Multiple (3/3)")
30     if (true) { yes(); } elseif (false) { no(); } else { no (); }
31     if (false) { no(); } elseif (true) { yes(); } else { no (); }
32     if (false) { no(); } elseif (false) { no(); } else { yes (); }
33     line()
34
35     out("Non-exhaustive (2/3)")
36     if (true) { yes(); } elseif (false) { no(); }
37     if (false) { no(); } elseif (true) { yes(); }
38     if (false) { no(); } elseif (false) { no(); }
39
40     main(System system, String[] args):
41         IfTest theif := new IfTest()

```

Test Source 13: "If Statements"

This test makes sure if statements function.

6.2.3 Expression Testing

This suite of test case makes sure that basic expressions do compile.

```

1  class Test:
2      public:
3          Integer a
4          Integer b
5          Integer c
6
7      init():
8          super()
9          a := 1
10         b := 2
11         c := 3
12
13     Integer overview():
14         Integer success := a+b
15         return success
16
17     main(System system, String[] args):
18         Test ab := new Test()
19         Printer p := system.out
20         p.printString("Sum of integer = ")
21         p.printInteger(ab.overview())
22         p.printString("\n")

```

Test Source 14: "Add Integers"

```

1  class Test:
2      public:
3          Float a
4          Float b
5          Integer c
6
7      init():
8          super()
9          a := 1.5
10         b := 2.2

```

```

11         c := 0
12
13     Float overview():
14         Float success := a+b
15         return success
16
17 main(System system, String[] args):
18     Test ab := new Test()
19     Printer p := system.out
20     p.printString("Sum of integer = ")
21     p.printFloat(ab.overview())
22     p.printString("\n")

```

Test Source 15: "Add Floats"

These tests add numeric literal objects together.

```

1 class Test:
2     public:
3         Integer a
4         Float b
5
6         init():
7             super()
8
9         Integer add():
10            a := 10 * 2 * 9
11            b := 6.0 * 0.5 * (-2.0)
12            return 0
13
14 main(System sys, String[] args):

```

Test Source 16: "Multiplication"

```

1 class Test:
2     public:
3         Integer a
4         Float b
5
6         init():
7             super()
8
9         Integer add():
10            a := (10 / 5) / -2
11            b := (10.0 / 5.0) / -2.0
12            return 0
13
14 main(System sys, String[] args):
15     Test t := new Test()
16     Printer p := sys.out
17
18     t.add()
19     p.printString("A is ")
20     p.printInteger(t.a)
21     p.printString(", B is ")
22     p.printFloat(t.b)
23     p.printString("\n")

```

Test Source 17: "Divition"

These tests form products/quotions of Floats/Integers.

```
1 class Test:
2     public:
3         Integer a
4         Integer b
5         Integer c
6
7         init():
8             super()
9             a := 1
10            b := 2
11            c := 3
12
13            Integer overview():
14                Integer success := a%b
15                return success
16
17            main(System system, String[] args):
18                Test ab := new Test()
19                Printer p := system.out
20                p.printString(" 1 % 2 = ")
21                p.printInteger(ab.overview())
22                p.printString("\n")
```

Test Source 18: "Modulus"

This test forms the modulus of Integers.

```
1 class Test:
2     public:
3         init():
4             super()
5
6         void interact():
7             Printer p := system.out
8             Integer i := 5
9             Float f := 1.5
10            p.printString("Sum of integer + float = ")
11            p.printFloat(i.toF() + f)
12            p.printString("\n")
13
14            main(System system, String[] args):
15                Test test := new Test()
16                test.interact()
```

Test Source 19: "Literal Casting and Addition"

```
1 class Test:
2     public:
3         init():
4             super()
5
6         void interact():
7             Printer p := system.out
8             Integer i := 5
9             Float f := 1.5
10            p.printString("integer - float = ")
11            p.printFloat(i.toF() - f)
12            p.printString("\n")
```

```

13
14 main(System system, String[] args):
15     Test test := new Test()
16     test.interact()

```

Test Source 20: "Literal Casting and Subtraction"

```

1 class Test:
2     public:
3         init():
4             super()
5
6         void interact():
7             Printer p := system.out
8             Integer i := 5
9             Float f := 1.5
10            p.printString("integer * float = ")
11            p.printFloat(i.toF() * f)
12            p.printString("\n")
13
14 main(System system, String[] args):
15     Test test := new Test()
16     test.interact()

```

Test Source 21: "Literal Casting and Multiplication"

```

1 class Test:
2     public:
3         init():
4             super()
5
6         void interact():
7             Printer p := system.out
8             Integer i := 5
9             Float f := 1.5
10            p.printString("float/Integer = ")
11            p.printFloat(f/i.toF())
12            p.printString("\n")
13
14 main(System system, String[] args):
15     Test test := new Test()
16     test.interact()

```

Test Source 22: "Literal Casting and Division"

```

1 class Test:
2     public:
3         init():
4             super()
5
6         void interact():
7             Printer p := system.out
8             Integer i := 5
9             Float f := 1.5
10            p.printString("integer ^ float = ")
11            p.printFloat(i.toF() ^ f)
12            p.printString("\n")
13

```

```

14  main(System system, String[] args):
15      Test test := new Test()
16      test.interact()

```

Test Source 23: "Literal Casting and Exponentiation"

These tests check that numerical literal objects can be cast to allow mathematic operations.

```

1  class Parent:
2      public:
3          init():
4              super()
5
6  class Child extends Parent:
7      public:
8          init():
9              super()
10
11 class Test:
12     public:
13         init():
14             super()
15
16     main(System system, String[] args):
17         Parent child := new Child()

```

Test Source 24: "Superclass Typing"

This test assigns a subclass to a variable typed to its parent.

```

1  class Test:
2      private:
3          void line():
4              system.out.println("\n")
5
6          void out(String msg):
7              system.out.println(msg)
8              line()
9
10     public:
11         init():
12             super()
13             Integer a:=2
14             Integer b:=3
15             Integer c
16
17         /* less and less and equal*/
18         if (a<2) { system.out.println("1. a=2 a<2 shouldnot print\n"); }
19         elsif (a<=2) { system.out.println("1. a=2 a<=2 success\n"); }
20         else { system.out.println("1. should never hit here\n"); }
21
22
23         /* greater and greater than equal */
24         if (b>3) { system.out.println("2. b=3 b>3 shouldnot print\n"); }
25         else { system.out.println("2. b=3 b>=3 success\n"); }
26
27         /*Equal and not equal*/
28         if (a <> b) { system.out.println("3. a!=b success \n"); }
29         a:=b
30         if (a=b) { system.out.println("4. a=b success\n"); }
31

```

```

32      /*And or */
33      if(a=3 and b=3) { system.out.println("5. a=3 and b=3 success\n"); }
34
35      b:=5
36      if(b=3 or a=3) { system.out.println("6. b=3 or a=3 success\n"); }
37
38      /*nand and nor and not*/
39      b:=4
40      a:=4
41      if(b=3 nor a=3) { system.out.println("7. b=10 nor a=10 success\n"); }
42      if(not(b=4 nand a=4)) { system.out.println("8. not(b=4 nand a=4) success\n"); }
43  }
44      b:=3
45      if(b=4 nand a=4) { system.out.println("9. b=4 nand a=4 success\n"); }
46      if(b=3 xor a=3) { system.out.println("10. b=3 xor a=3 success\n"); }
47      c:=10
48      if((a<>b or b=c) and c=10) { system.out.println("11. (a<>b or b=c) and c=10
49      success\n"); }
50      line()
51
52  main(System system, String[] args):
53      Test thief := new Test()

```

Test Source 25: "Boolean Comparison"

This test performs boolean comparisons between numeric literal objects.

```

1
2  class Person:
3      protected:
4          String name
5
6      public:
7          init(String name):
8              super()
9              this.name := name
10
11      void introduce():
12          Printer p := system.out
13          p.println("Hello, my name is ")
14          p.println(name)
15          p.println(", and I am from ")
16          p.println(refine origin() to String)
17          p.println("I am ")
18          p.println(Integer(refine age() to Integer))
19          p.println("years old. My occupation is ")
20          p.println(refine work() to String)
21          p.println("It was nice meeting you.\n")
22
23  class Test:
24      protected:
25          init():
26              super()
27
28      main(System sys, String[] args):
29          (new Person("Matthew")) {
30              String introduce.origin() { return "New Jersey"; }
31              Integer introduce.age() { return 33; }
32              String introduce.work() { return "Student"; }
33          }.introduce()
34
35          (new Person("Arthy")) {
36              String introduce.origin() { return "India"; }

```

```

37     Integer introduce.age() { return 57; }
38     String introduce.work() { return "Student"; }
39 }).introduce()
40
41     (new Person("Weiyuan") {
42         String introduce.origin() { return "China"; }
43         Integer introduce.age() { return 24; }
44         String introduce.work() { return "Student"; }
45     }).introduce()
46
47     (new Person("Ben") {
48         String introduce.origin() { return "New York"; }
49         Integer introduce.age() { return 24; }
50         String introduce.work() { return "Student"; }
51     }).introduce()

```

Test Source 26: "Anonymous objects"

This tests forms anonymous objects.

```

1  class Test:
2      private:
3          void print(Integer i):
4              Printer p := system.out
5              p.printString("a[")
6              p.printInteger(i)
7              p.printString("] = ")
8              p.printInteger(a[i])
9              p.printString("\n")
10
11     public:
12         Integer[] a
13         init():
14             super()
15             a := new Integer[] (4)
16             a[0] := 3
17             a[1] := 2
18             a[2] := 1
19             a[3] := 0
20
21         void print():
22             Integer i := 0
23             while (i < 4):
24                 print(i)
25                 i += 1
26
27     main(System system, String[] args):
28         Test f
29         f := new Test()
30         f.print()

```

Test Source 27: "Arrays"

This test forms an array.

```

1  class Parent:
2      public:
3          Integer a
4          Integer b
5          Integer c
6

```

```

7      init():
8          super()
9          a := 1
10         b := 2
11         c := 0
12
13     Integer overview():
14         Integer success := refine toExtra(a,b) to Integer
15         return success
16
17 class Child extends Parent:
18     refinement:
19         Integer overview.toExtra(Integer a, Integer b):
20             Integer success := a + b
21             Printer p := new Printer(true)
22             p.printInteger(a)
23             p.printInteger(b)
24             p.printInteger(c)
25             return success
26     public:
27         Integer a1
28         Integer b1
29         Integer c1
30
31     init():
32         super()
33         a1 := 1
34         b1 := 2
35         c1 := 0
36
37 class Test:
38     public:
39         init():
40             super()
41
42     main(System system, String[] args):
43         Parent ab := new Child()
44         Printer p := system.out
45         p.printString("Sum of integer = ")
46         p.printInteger(ab.overview())
47         p.printString("\n")

```

Test Source 28: "Refinement"

This test checks that basic refinement works.

```

1 class Parent:
2     public:
3         Integer a
4         Integer b
5         Integer c
6
7     init():
8         super()
9         a := 1
10        b := 2
11        c := 0
12
13    Integer overview():
14        Integer success := -1
15        if (refinable(toExtra)) {
16            success := refine toExtra(a,b) to Integer;
17        }

```

```

18         return success
19
20 class Child extends Parent:
21     refinement:
22         Integer overview.toExtra(Integer a, Integer b):
23             Integer success := a + b
24             Printer p := new Printer(true)
25             p.printInteger(a)
26             p.printInteger(b)
27             p.printInteger(c)
28             return success
29     public:
30         Integer a1
31         Integer b1
32         Integer c1
33
34         init():
35             super()
36             a1 := 1
37             b1 := 2
38             c1 := 0
39
40 class Test:
41     public:
42         init():
43             super()
44
45     main(System system, String[] args):
46         Parent ab := new Child()
47         Printer p := system.out
48         p.printString("Sum of integer = ")
49         p.printInteger(ab.overview())
50         p.printString("\n")

```

Test Source 29: "Refinable"

This test checks that the refinable keyword works.

```

1 class Parent:
2     protected:
3         Integer a
4         Integer b
5         String name
6
7     public:
8         init(String name):
9             super()
10
11             this.name := name
12             a := 1
13             b := 2
14
15         void print():
16             Printer p := system.out
17             p.printString(name)
18             p.printString(": A is ")
19             p.printInteger(a)
20             p.printString(", B is ")
21             p.printInteger(b)
22             p.printString("\n")
23
24         void update():
25             if (refinable(setA)):

```

```

26     a := refine setA() to Integer
27     if (refinable(setB)):
28         b := refine setB() to Integer
29
30 class Son extends Parent:
31     public:
32         init(String name):
33             super(name)
34
35     refinement:
36         Integer update.setA():
37             return -1
38         Integer update.setB():
39             return -2
40
41 class Daughter extends Parent:
42     public:
43         init(String name):
44             super(name)
45
46     refinement:
47         Integer update.setA():
48             return 10
49         Integer update.setB():
50             return -5
51
52
53 class Test:
54     protected:
55         init():
56             super()
57
58     main(System sys, String[] args):
59         Parent pop := new Parent("Father")
60         Son son := new Son("Son")
61         Daughter daughter := new Daughter("Daughter")
62
63         pop.print()
64         son.print()
65         daughter.print()
66         sys.out.printString("—————\n")
67         pop.update()
68         son.update()
69         daughter.update()
70
71         pop.print()
72         son.print()
73         daughter.print()

```

Test Source 30: "Refinements"

This test makes multiple trivial refinements.

6.2.4 Structure Testing

```

1 class MainTest:
2     public:
3         init():
4             super()
5     main(System system, String[] args):
6         Integer a

```



```
7   a := 0
8   a += 1
```

Test Source 31: "Main Method"

This test forms a main method

```
1  class Math:
2      private:
3          Float xyz
4      public:
5          init():
6              super()
7          Integer add(Integer a, Integer b):
8              return 6
9          Integer sub(Integer a, Integer c):
10             return 4
11     main(System sys, String[] args):
12
13  class NonMath:
14      private:
15          String shakespeare
16      public:
17          init():
18              super()
19          String recite():
20              return "hey"
21     main(System sys, String[] hey):
```

Test Source 32: "Empty Bodies"

This test presents minimalistic bodies for a variety of methods.

```
1  class FuncTest:
2      public:
3          Integer a
4
5          init():
6              super()
7              a := 1
8
9      private:
10         Integer incre_a(Integer b):
11             a := a + b
12             return a
13
14         Integer incre_a_twice(Integer b):
15             incre_a(b)
16             incre_a(b)
17             return a
18
19     main(System system, String[] args):
20         FuncTest test := new FuncTest()
```

Test Source 33: "Functions"

This test probes function scope.

6.2.5 A Complex Test

```

1  class IOTest:
2      public:
3          Integer a
4          Integer b
5          Integer c
6          init():
7              super()
8              a := 1
9              b := 2
10             c := 0
11         void overview():
12             Printer p := new Printer(true)
13             p.printInteger(a)
14             p.printInteger(b)
15             p.printInteger(c)
16         Integer incre_ab():
17             Scanner s := new Scanner()
18             Integer delta
19             delta := s.scanInteger()
20             a := a + delta
21             b := b + delta
22             return c
23         Integer arith():
24             c := -(a + b)
25             return c
26
27     class Main:
28         public:
29             init():
30                 super()
31             main(String[] args):
32                 IOTest ab := new IOTest()
33                 ab.overview()
34                 ab.incre_ab()
35                 ab.overview()
36                 ab.arith()
37                 ab.overview()

```

Test Source 34: "Complex Scanning"

This test does a series of more advanced tasks in Gamma.

7 Lessons Learned

Arthy

First of all, I should thank my wonderful team mates and I enjoyed every bit working with them. Be it clearly silly questions on the language or design or OCAML anything and everything they were always there! And without them it would have certainly not been possible to have pulled this project i must confess well yea at the last moment. Thanks guys!

Thanks to Professor Edwards for making this course so much fun - you never feel the pressure of taking a theoretical course as this - as he puts it - "...in how many other theoretical courses have you had a lecture that ends with a tatoood hand.."

As any team projects we had our own idiosyncracies that left us with missing deadlines and extending demo deadline and what not - so we were not that one off team which miraculously fit well - we were just like any other team but a team that learnt lessons quickly applied them - left ego outside the door - and worked for the fun of the project! If the team has such a spirit that's all that is required.

Advice 1. Do have a team lead 2. Do have one person who is good in OCAML if possible or at least has had experiences with modern programming languages. 3. Have one who is good in programming language theory 4. Ensure you have team meetings - if people do not turn up or go missing - do open up talk to them 5. Ensure everyone is comfortable with the project and is at the same pace as yours early on 6. Discuss the design and make a combined decision - different people think differently that definitely will help. 7. This is definitely a fun course and do not spoil it by procastration - with OCAML you just have few lines to code why not start early and get it done early (Smiley) 8. I may want to say do not be ambitious - but in retrospect - I learnt a lot - and may be wish some more - so try something cool - after all that's what is grad school for!

Good luck

Ben

This class has been amazing in terms of a practical experience in writting low-level programing and forming a platform for others to write at a higher more abstract-level. I came into this expecting a lot of what the others say they have learned, the most important learning for me is how vital it is to understand your team as much as possible. We are four people with a very diverse set of talents and styles. Applied properly, we probably could have done just about anything with our collective talents. (Spoiler, we did not apply our group talents effectively as would have been hoped.)

My advice to future teams is to get to know each other as computer scientists and people first. If you have the time, do a small (day-long) project together like a mini hackathon. Figure out if your styles differ and write a style guide on which you can all agree. Realistically look at who will have time when. This is not the only thing on anyone's plate, you might have to front-load one member and back-load another. Establish clear leadership and a division of tasks. We just pushed people at the task at hand and were delaying by half-days for a given component to be ready. Write in parallel, it's easier to make your code match up than write linearly and mix schedules and styles. (If you could see the amount of formatting and style correction commits on our repository...)

Good luck. This course is worth it but a real challenge.

Matthew

I had a beginning of an idea of how OOP stuff worked underneath the hood, but this really opened my eyes up to how much work was going on.

It also taught me a lot about making design decisions, and how it's never a good idea to say "this time we'll just use strings and marker values cause we need it done sooner than later" – if Algebraic Data Types are available, use them. Even if it means you have to go back and adjust old code because of previous ideas fall out of line with new ones.

I learned how annoying the idea of a NULL value in a typed system can be when we don't give casting as an option (something we should have thought about before), and how smart python is by having methods accept and name the implicit parameter themselves. Good job, GvR.

Advice

- Start early and procrastinate less
- Have a team leader and communicate better
- Enjoy it

Weiyuan

First I would like to say that this is a very cool, educational and fun project.

One thing I learned from this project is that I take modern programming languages for granted. I enjoyed many comfortable features and syntactic sugar but never realized there is so much craziness under the hood. We had a long list of ambitious goals at the beginning. Many of them had to be given up as the project went on. From parsing to code generation, I faced a lot of design decisions that I did not even know existed. I gained a much better understanding of how programming languages work and why they are designed the way they are. Also, now I have a completely refreshed view when I see posts titled "Java vs. C++" on the Internet.

Another thing I learned is that proper task division, time management and effective communication are extremely important for a team project. Doing things in parallel and communicating smoothly can save you a lot of trouble.

Finally, I learned my first functional programming language OCaml and I do like it, though I still feel it's weird sometimes.

8 Appendix

```
1  Int    addInt.gamma
2  Float  addFloat.gamma
3  Bool
4  String
5
6  Binop
7
8  arith
9  Add    addInt.gamma addFloat.gamma addMix.gamma addMix.gamma_err
10 Sub    subMix.gamma
11 Prod   prodMix.gamma
12 Div    divMix.gamma
13 Mod    mod.gamma
14 Neg
15 Pow    powMix.gamma
16
17 numtest ifeq.gamma ( will be renamed)
18 Eq      pass
19 Neq     pass
20 Less    pass
21 Grtr    pass
22 Leq     pass
23 Geq     pass
24 And     pass
25 Or      pass
26 Nand    pass
27 Nor     pass
28 Xor     pass
29 Not     pass
30 nested conditions not tested
31
32 This    ANY
33 Null
34 Id      ANY
35 NewObj  MANY
36 Anonymous
37 Literal ANY
38 Assign  ANY
39 Deref
40 Field   ANY
41 Invoc   ANY
42 Unop
43
44 refinement
45 Refine   refine_refinable.gamma refine_unrefinable.gamma
46 Refinable refineable.gamma refineable.gamma_err
47
48 Decl     addInt.gamma
49 If       ifeq.gamma nested-if-to-tested
50 While
51 Expr
52 Return   ANY
53 Super
54
55 private
56 public
57 protect
58 main
59 extends  refinement testcase
```

Source 1: "compiler-tests/testcaseregistry"

```

1  class IOTest:
2      public:
3          Integer a
4          Integer b
5          Integer c
6          init():
7              super()
8              a := 1
9              b := 2
10             c := 0
11         void overview():
12             Printer p := new Printer(true)
13             p.printInteger(a)
14             p.printInteger(b)
15             p.printInteger(c)
16         Integer incre_ab():
17             Scanner s := new Scanner()
18             Integer delta
19             delta := s.scanInteger()
20             a := a + delta
21             b := b + delta
22             return c
23         Integer arith():
24             c := -(a + b)
25             return c
26
27     class Main:
28         public:
29             init():
30                 super()
31             main(String[] args):
32                 IOTest ab := new IOTest()
33                 ab.overview()
34                 ab.incre_ab()
35                 ab.overview()
36                 ab.arith()
37                 ab.overview()

```

Source 2: "compiler-tests/mix.gamma"

```

1  class IOTest:
2      public:
3          init():
4              super()
5
6          void interact():
7              Printer p := system.out
8              Integer i := promptInteger("Please enter an integer")
9              Float f := promptFloat("Please enter a float")
10             p.printString("Sum of integer + float = ")
11             p.printFloat(i.toF() + f)
12             p.printString("\n")
13
14         private:
15             void prompt(String msg):
16                 system.out.printString(msg)
17                 system.out.printString(": ")

```

```

18 Integer promptInteger(String msg):
19     prompt(msg)
20     return system.in.scanInteger()
21
22
23 Float promptFloat(String msg):
24     prompt(msg)
25     return system.in.scanFloat()
26
27 main(System system, String[] args):
28     IOTest test := new IOTest()
29     test.interact()

```

Source 3: "compiler-tests/programs/io.gamma"

```

1 class HelloWorld:
2     public:
3         String greeting
4         init():
5             super()
6             greeting := "Hello World!"
7
8 main(System system, String[] args):
9     HelloWorld hw := new HelloWorld()
10    system.out.printString(hw.greeting)
11    system.out.printString("\n")

```

Source 4: "compiler-tests/programs/helloworld.gamma"

```

1 class Test:
2     public:
3         init():
4             super()
5
6 main(System sys, String[] args):
7     Integer i := 0
8     Printer p := sys.out
9
10    while (i < sys.argv):
11        p.printString("arg[")
12        p.printInteger(i)
13        p.printString("] = ")
14        p.printString(args[i])
15        p.printString("\n")
16        i += 1

```

Source 5: "compiler-tests/programs/args.gamma"

```

1 class Parent:
2     public:
3         init():
4             super()
5
6 class Child extends Parent:
7     public:
8         init():
9             super()
10

```

```

11 class Test:
12     public:
13         init():
14             super()
15
16     main(System system, String[] args):
17         Child child := new Parent()

```

Source 6: "compiler-tests/bad/super-assign.gamma"

```

1 class BadDecl:
2     public:
3         init():
4             super()
5             Integer a := 3.4

```

Source 7: "compiler-tests/bad/decl.gamma"

```

1 class Test:
2     public:
3         Float a
4         Float b
5         Integer c
6
7         init():
8             super()
9             a := 1.5
10            b := 2.2
11            c := 3
12
13        Float overview():
14            Float success := a+b+c
15            return success
16
17        main(System system, String[] args):
18            Test ab := new Test()
19            Printer p := system.out
20            p.printString("Sum of integer = ")
21            p.printFloat(ab.overview())
22            p.printString("\n")

```

Source 8: "compiler-tests/bad/addMix.gamma"

```

1 class BadReturn:
2     public:
3         init():
4             super()
5
6         Integer badReturn():
7             return "Hey There"

```

Source 9: "compiler-tests/bad/return1.gamma"

```

1 class BadAssign:
2     public:
3         init():

```



```

4      super()
5      Integer a
6      a := 3.4

```

Source 10: "compiler-tests/bad/assign.gamma"

```

1  class BadStatic:
2      public:
3          Integer getZero():
4              return 0
5          init():
6              super()
7          main(System system, String[] args):
8              getZero() /* This is supposed to fail. DON'T CHANGE */

```

Source 11: "compiler-tests/bad/static.gamma"

```

1  class Parent:
2      public:
3          Integer a
4          Integer b
5          Integer c
6
7      init():
8          super()
9          a := 1
10         b := 2
11         c := 0
12
13         Integer overview():
14             Integer success := refine toExtra(a,b) to Integer
15             return success
16
17  class Child extends Parent:
18      refinement:
19          Integer overview.toExtra(Integer a, Integer b):
20              Integer success := a + b
21              Printer p := new Printer(true)
22              p.printInteger(a)
23              p.printInteger(b)
24              p.printInteger(c)
25              return success
26      public:
27          Integer a1
28          Integer b1
29          Integer c1
30
31      init():
32          super()
33          a1 := 1
34          b1 := 2
35          c1 := 0
36
37  class Test:
38      public:
39          init():
40              super()
41
42          main(System system, String[] args):
43              Parent ab := new Parent
44              Printer p := system.out

```

```

45     p.println("Sum of integer = ")
46     p.printInteger(ab.overview())
47     p.println("\n")

```

Source 12: "compiler-tests/bad/refine_refinable.gamma"

```

1  class BadReturn:
2      public:
3          init():
4              super()
5
6          Integer badReturn():
7              return

```

Source 13: "compiler-tests/bad/return2.gamma"

```

1  class BadReturn:
2      public:
3          init():
4              super()
5
6          void badReturn():
7              return "Hey There"

```

Source 14: "compiler-tests/bad/return3.gamma"

```

1  class Parent:
2      public:
3          Integer a
4          Integer b
5          Integer c
6
7          init():
8              super()
9              a := 1
10             b := 2
11             c := 0
12
13             Integer overview():
14                 Integer success := -1
15                 if (refinable(toExtra)) {
16                     success := refine toExtra(a,b) to Integer;
17                 }
18                 return success
19
20  class Child extends Parent:
21      refinement:
22          Integer overview.toExtra(Integer a, Integer b):
23              Integer success := a + b
24              Printer p := new Printer(true)
25              p.printInteger(a)
26              p.printInteger(b)
27              p.printInteger(c)
28              return success
29      public:
30          Integer a1
31          Integer b1
32          Integer c1

```

```

33
34     init():
35         super()
36         a1 := 1
37         b1 := 2
38         c1 := 0
39
40 class Test:
41     public:
42         init():
43             super()
44
45     main(System system, String[] args):
46         Parent ab := new Parent()
47         Printer p := system.out
48         p.printString("Sum of integer = ")
49         p.printInteger(ab.overview())
50         p.printString("\n")

```

Source 15: "compiler-tests/bad/refinable.gamma"

```

1
2 class WhileLoopTest:
3     public:
4         init():
5             super()
6             Integer a := 0
7             while((a>=0) and (a<10)):
8                 system.out.printInteger(a)
9                 system.out.printString("\n")
10                a := a + 1
11
12     main(System system, String[] args):
13         new WhileLoopTest()

```

Source 16: "compiler-tests/stmts/while_condn.gamma"

```

1
2 class WhileLoopTest:
3     public:
4         init():
5             super()
6             Integer a := 0
7             while(true):
8                 system.out.printInteger(a)
9                 system.out.printString("\n")
10                a := a + 1
11
12     main(System system, String[] args):
13         new WhileLoopTest()

```

Source 17: "compiler-tests/stmts/while.gamma"

```

1 class IfTest:
2     private:
3         void line():
4             system.out.printString("\n")
5

```

```

6      void out(String msg):
7          system.out.println(msg)
8          line()
9
10     void yes():
11         out("This should print.")
12     void no():
13         out("This should not print.")
14
15     public:
16         init():
17             super()
18
19             out("Simple (1/2)")
20             if (true) { yes(); }
21             if (false) { no(); }
22             line()
23
24             out("Basic (2/2)")
25             if (true) { yes(); } else { no(); }
26             if (false) { no(); } else { yes(); }
27             line()
28
29             out("Multiple (3/3)")
30             if (true) { yes(); } elseif (false) { no(); } else { no(); }
31             if (false) { no(); } elseif (true) { yes(); } else { no(); }
32             if (false) { no(); } elseif (false) { no(); } else { yes(); }
33             line()
34
35             out("Non-exhaustive (2/3)")
36             if (true) { yes(); } elseif (false) { no(); }
37             if (false) { no(); } elseif (true) { yes(); }
38             if (false) { no(); } elseif (false) { no(); }
39
40     main(System system, String[] args):
41         IfTest theif := new IfTest()

```

Source 18: "compiler-tests/stmts/if.gamma"

```

1      class Test:
2          public:
3              Integer a
4              Integer b
5              Integer c
6
7          init():
8              super()
9              a := 1
10             b := 2
11             c := 3
12
13             Integer overview():
14                 Integer success := a+b
15                 return success
16
17     main(System system, String[] args):
18         Test ab := new Test()
19         Printer p := system.out
20         p.println("Sum of integer = ")
21         p.printInteger(ab.overview())
22         p.println("\n")

```

Source 19: "compiler-tests/exprs/addInt.gamma"

```
1 class Test:
2   public:
3     Integer a
4     Float b
5
6     init():
7       super()
8
9     Integer add():
10      a := 10 * 2 * 9
11      b := 6.0 * 0.5 * (-2.0)
12      return 0
13
14 main(System sys, String[] args):
```

Source 20: "compiler-tests/exprs/prod.gamma"

```
1 class Test:
2   public:
3     init():
4       super()
5
6     void interact():
7       Printer p := system.out
8       Integer i := 5
9       Float f := 1.5
10      p.printString("integer - float = ")
11      p.printFloat(i.toF() - f)
12      p.printString("\n")
13
14 main(System system, String[] args):
15   Test test := new Test()
16   test.interact()
```

Source 21: "compiler-tests/exprs/subMix.gamma"

```
1 class Parent:
2   public:
3     init():
4       super()
5
6 class Child extends Parent:
7   public:
8     init():
9       super()
10
11 class Test:
12   public:
13     init():
14       super()
15
16 main(System system, String[] args):
17   Parent child := new Child()
```

Source 22: "compiler-tests/exprs/super-assign.gamma"

```
1 class Test:
2   public:
3     init():
4       super()
5
6     void interact():
7       Printer p := system.out
8       Integer i := 5
9       Float f := 1.5
10      p.printString("float/Integer = ")
11      p.printFloat(f/i.toF())
12      p.printString("\n")
13
14 main(System system, String[] args):
15   Test test := new Test()
16   test.interact()
```

Source 23: "compiler-tests/exprs/divMix.gamma"

```
1 class Test:
2   public:
3     init():
4       super()
5
6     void interact():
7       Printer p := system.out
8       Integer i := 5
9       Float f := 1.5
10      p.printString("Sum of integer + float = ")
11      p.printFloat(i.toF() + f)
12      p.printString("\n")
13
14 main(System system, String[] args):
15   Test test := new Test()
16   test.interact()
```

Source 24: "compiler-tests/exprs/addMix.gamma"

```
1 class Test:
2   private:
3     void line():
4       system.out.printString("\n")
5
6     void out(String msg):
7       system.out.printString(msg)
8       line()
9
10  public:
11    init():
12      super()
13      Integer a:=2
14      Integer b:=3
15      Integer c
16
17    /* less and less and equal*/
```

```

18     if (a<2) { system.out.println("1. a=2 a<2 shouldnot print\n"); }
19     elsif (a<=2) { system.out.println("1. a=2 a<=2 success\n"); }
20     else { system.out.println("1. should never hit here\n"); }
21
22
23     /* greater and greater than equal */
24     if (b>3) { system.out.println("2. b=3 b>3 shouldnot print\n"); }
25     else { system.out.println("2. b=3 b>=3 success\n"); }
26
27     /*Equal and not equal*/
28     if (a <> b) { system.out.println("3. a!=b success \n"); }
29     a:=b
30     if (a=b) { system.out.println("4. a=b success\n"); }
31
32     /*And or */
33     if(a=3 and b=3) { system.out.println("5. a=3 and b=3 success\n"); }
34
35     b:=5
36     if(b=3 or a=3) { system.out.println("6. b=3 or a=3 success\n"); }
37
38     /*nand and nor and not*/
39     b:=4
40     a:=4
41     if(b=3 nor a=3) { system.out.println("7. b=10 nor a=10 success\n"); }
42     if(not(b=4 nand a=4)) { system.out.println("8. not(b=4 nand a=4) success\n"); }
43 }
44     b:=3
45     if(b=4 nand a=4) { system.out.println("9. b=4 nand a=4 success\n"); }
46     if(b=3 xor a=3) { system.out.println("10. b=3 xor a=3 success\n"); }
47     c:=10
48     if((a<>b or b=c) and c=10) { system.out.println("11. (a<>b or b=c) and c=10
49     success\n"); }
50     line()
51
52 main(System system, String[] args):
53     Test theif := new Test()

```

Source 25: "compiler-tests/exprs/ifeq.gamma"

```

1 class Test:
2     public:
3         Integer a
4         Integer b
5         Integer c
6
7     init():
8         super()
9         a := 1
10        b := 2
11        c := 3
12
13    Integer overview():
14        Integer success := a%b
15        return success
16
17    main(System system, String[] args):
18        Test ab := new Test()
19        Printer p := system.out
20        p.println(" 1 % 2 = ")
21        p.printInteger(ab.overview())
22        p.println("\n")

```

Source 26: "compiler-tests/exprs/mod.gamma"

```

1
2 class Person:
3     protected:
4         String name
5
6     public:
7         init(String name):
8             super()
9             this.name := name
10
11         void introduce():
12             Printer p := system.out
13             p.printString("Hello, my name is ")
14             p.printString(name)
15             p.printString(", and I am from ")
16             p.printString(refine origin() to String)
17             p.printString(" I am ")
18             p.printInteger(refine age() to Integer)
19             p.printString(" years old. My occupation is ")
20             p.printString(refine work() to String)
21             p.printString(". It was nice meeting you.\n")
22
23 class Test:
24     protected:
25         init():
26             super()
27
28     main(System sys, String[] args):
29         (new Person("Matthew")) {
30             String introduce.origin() { return "New Jersey"; }
31             Integer introduce.age() { return 33; }
32             String introduce.work() { return "Student"; }
33         }.introduce()
34
35         (new Person("Arthy")) {
36             String introduce.origin() { return "India"; }
37             Integer introduce.age() { return 57; }
38             String introduce.work() { return "Student"; }
39         }.introduce()
40
41         (new Person("Weiyuan")) {
42             String introduce.origin() { return "China"; }
43             Integer introduce.age() { return 24; }
44             String introduce.work() { return "Student"; }
45         }.introduce()
46
47         (new Person("Ben")) {
48             String introduce.origin() { return "New York"; }
49             Integer introduce.age() { return 24; }
50             String introduce.work() { return "Student"; }
51         }.introduce()

```

Source 27: "compiler-tests/exprs/anonymous.gamma"

```

1 class Test:
2     public:
3         init():

```



```

4      super()
5
6      void interact():
7          Printer p := system.out
8          Integer i := 5
9          Float f := 1.5
10         p.printString("integer ^ float = ")
11         p.printFloat(i.toF() ^ f)
12         p.printString("\n")
13
14     main(System system, String[] args):
15         Test test := new Test()
16         test.interact()

```

Source 28: "compiler-tests/exprs/powMix.gamma"

```

1  class Test:
2      public:
3          init():
4              super()
5
6          void interact():
7              Printer p := system.out
8              Integer i := 5
9              Float f := 1.5
10             p.printString("integer * float = ")
11             p.printFloat(i.toF() * f)
12             p.printString("\n")
13
14     main(System system, String[] args):
15         Test test := new Test()
16         test.interact()

```

Source 29: "compiler-tests/exprs/prodMix.gamma"

```

1  class Parent:
2      protected:
3          Integer a
4          Integer b
5          String name
6
7      public:
8          init(String name):
9              super()
10
11             this.name := name
12             a := 1
13             b := 2
14
15          void print():
16              Printer p := system.out
17              p.printString(name)
18              p.printString(": A is ")
19              p.printInteger(a)
20              p.printString(", B is ")
21              p.printInteger(b)
22              p.printString("\n")
23
24          void update():
25              if (refinable(setA)):
26                  a := refine setA() to Integer

```

```

27         if (refinable(setB)):
28             b := refine setB() to Integer
29
30     class Son extends Parent:
31         public:
32             init (String name):
33                 super(name)
34
35         refinement:
36             Integer update.setA():
37                 return -1
38             Integer update.setB():
39                 return -2
40
41     class Daughter extends Parent:
42         public:
43             init (String name):
44                 super(name)
45
46         refinement:
47             Integer update.setA():
48                 return 10
49             Integer update.setB():
50                 return -5
51
52
53     class Test:
54         protected:
55             init():
56                 super()
57
58         main(System sys, String[] args):
59             Parent pop := new Parent("Father")
60             Son son := new Son("Son")
61             Daughter daughter := new Daughter("Daughter")
62
63             pop.print()
64             son.print()
65             daughter.print()
66             sys.out.printString("—————\n")
67             pop.update()
68             son.update()
69             daughter.update()
70
71             pop.print()
72             son.print()
73             daughter.print()

```

Source 30: "compiler-tests/exprs/simple-refine.gamma"

```

1     class Test:
2         private:
3             void print(Integer i):
4                 Printer p := system.out
5                 p.printString("a[")
6                 p.printInteger(i)
7                 p.printString("] = ")
8                 p.printInteger(a[i])
9                 p.printString("\n")
10
11         public:
12             Integer[] a
13             init():

```

```

14     super()
15     a := new Integer [] (4)
16     a[0] := 3
17     a[1] := 2
18     a[2] := 1
19     a[3] := 0
20
21     void print():
22         Integer i := 0
23         while (i < 4):
24             print(i)
25             i += 1
26
27     main(System system, String [] args):
28         Test f
29         f := new Test()
30         f.print()

```

Source 31: "compiler-tests/exprs/newarr.gamma"

```

1  class Test:
2      public:
3          Float a
4          Float b
5          Integer c
6
7      init():
8          super()
9          a := 1.5
10         b := 2.2
11         c := 0
12
13     Float overview():
14         Float success := a+b
15         return success
16
17     main(System system, String [] args):
18         Test ab := new Test()
19         Printer p := system.out
20         p.printString("Sum of integer = ")
21         p.printFloat(ab.overview())
22         p.printString("\n")

```

Source 32: "compiler-tests/exprs/addFloat.gamma"

```

1  class Test:
2      public:
3          Integer a
4          Float b
5
6      init():
7          super()
8
9      Integer add():
10         a := (10 / 5) / -2
11         b := (10.0 / 5.0) / -2.0
12         return 0
13
14     main(System sys, String [] args):
15         Test t := new Test()
16         Printer p := sys.out

```

```

17     t.add()
18     p.printString("A is ")
19     p.printInteger(t.a)
20     p.printString(", B is ")
21     p.printFloat(t.b)
22     p.printString("\n")
23

```

Source 33: "compiler-tests/exprs/div.gamma"

```

1  class Parent:
2      public:
3          Integer a
4          Integer b
5          Integer c
6
7      init():
8          super()
9          a := 1
10         b := 2
11         c := 0
12
13     Integer overview():
14         Integer success := refine toExtra(a,b) to Integer
15         return success
16
17 class Child extends Parent:
18     refinement:
19         Integer overview.toExtra(Integer a, Integer b):
20             Integer success := a + b
21             Printer p := new Printer(true)
22             p.printInteger(a)
23             p.printInteger(b)
24             p.printInteger(c)
25             return success
26     public:
27         Integer a1
28         Integer b1
29         Integer c1
30
31     init():
32         super()
33         a1 := 1
34         b1 := 2
35         c1 := 0
36
37 class Test:
38     public:
39         init():
40             super()
41
42     main(System system, String[] args):
43         Parent ab := new Child()
44         Printer p := system.out
45         p.printString("Sum of integer = ")
46         p.printInteger(ab.overview())
47         p.printString("\n")

```

Source 34: "compiler-tests/exprs/refine_refinable.gamma"

```

1  class Parent:

```

```

2   public:
3       Integer a
4       Integer b
5       Integer c
6
7       init():
8           super()
9           a := 1
10          b := 2
11          c := 0
12
13       Integer overview():
14           Integer success := -1
15           if (refinable(toExtra)) {
16               success := refine toExtra(a,b) to Integer;
17           }
18           return success
19
20   class Child extends Parent:
21       refinement:
22           Integer overview.toExtra(Integer a, Integer b):
23               Integer success := a + b
24               Printer p := new Printer(true)
25               p.printInteger(a)
26               p.printInteger(b)
27               p.printInteger(c)
28               return success
29       public:
30           Integer a1
31           Integer b1
32           Integer c1
33
34       init():
35           super()
36           a1 := 1
37           b1 := 2
38           c1 := 0
39
40   class Test:
41       public:
42           init():
43               super()
44
45       main(System system, String[] args):
46           Parent ab := new Child()
47           Printer p := system.out
48           p.printString("Sum of integer = ")
49           p.printInteger(ab.overview())
50           p.printString("\n")

```

Source 35: "compiler-tests/exprs/refinable.gamma"

```

1   class MainTest:
2       public:
3           init():
4               super()
5       main(System system, String[] args):
6           Integer a
7           a := 0
8           a += 1

```

Source 36: "compiler-tests/structure/main.gamma"

```

1 class Math:
2     private:
3         Float xyz
4     public:
5         init():
6             super()
7         Integer add(Integer a, Integer b):
8             return 6
9         Integer sub(Integer a, Integer c):
10            return 4
11     main(System sys, String[] args):
12
13 class NonMath:
14     private:
15         String shakespear
16     public:
17         init():
18             super()
19         String recite():
20             return "hey"
21     main(System sys, String[] hey):

```

Source 37: "compiler-tests/structure/no-bodies.gamma"

```

1 class FuncTest:
2     public:
3         Integer a
4
5         init():
6             super()
7             a := 1
8
9     private:
10        Integer incre_a(Integer b):
11            a := a + b
12            return a
13
14        Integer incre_a_twice(Integer b):
15            incre_a(b)
16            incre_a(b)
17            return a
18
19    main(System system, String[] args):
20        FuncTest test := new FuncTest()

```

Source 38: "compiler-tests/structure/func.gamma"

```

1 open Ast
2 open Klass
3
4 (** Functions to be used with testing in the interpreter (or test scripts we write later)
5  *)
6 let get_example_path dir example = String.concat Filename.dir_sep ["test"; "tests"; "
7   Brace"; dir; example]
8
9 let get_example_scan dir example =
10     let input = open_in (get_example_path dir example) in
11     let tokens = Inspector.from_channel input in
12     let _ = close_in input in

```

```

12 tokens
13
14 let get_example_parse dir example =
15   let tokens = get_example_scan dir example in
16   Parser.cdecls (WhiteSpace.lextoks tokens) (Lexing.from_string "")
17
18 let get_example_longest_body dir example =
19   let classes = get_example_parse dir example in
20   let methods aklass = List.flatten (List.map snd (Klass.klass_to_functions aklass)) in
21   let all_methods = List.flatten (List.map methods classes) in
22   let with_counts = List.map (function func -> (Util.get_statement_count func.body,
23   func)) all_methods in
24   let maximum = List.fold_left max 0 (List.map fst with_counts) in
25   List.map snd (List.filter (function (c, _) -> c == maximum) with_counts)

```

Source 39: "Debug.ml"

```

1  open Printf
2  open Util
3
4  let output_string whatever =
5    print_string whatever;
6    print_newline()
7
8  let load_file filename =
9    if Sys.file_exists filename
10     then open_in filename
11     else raise (Failure("Could not find file " ^ filename ^ "."))
12
13  let with_file f file =
14    let input = load_file file in
15    let result = f input in
16    close_in input;
17    result
18
19  let get_data ast =
20    let (which, builder) = if (Array.length Sys.argv <= 2)
21      then ("Normal", KlassData.build_class_data)
22      else ("Experimental", KlassData.build_class_data_test) in
23    output_string (Format.sprintf " * Using %s KlassData Builder" which);
24    match builder ast with
25    | Left(data) -> data
26    | Right(issue) -> Printf.fprintf stderr "%s\n" (KlassData.errstr issue); exit 1
27
28  let do_deanon klass_data sast = match Unanonymous.deanonimize klass_data sast with
29    | Left(result) -> result
30    | Right(issue) -> Printf.fprintf stderr "Error Deanonimizing:\n%s\n" (KlassData.
31    errstr issue); exit 1
32
33  let source_cast _ =
34    output_string " * Reading Tokens...";
35    let tokens = with_file Inspector.from_channel Sys.argv.(1) in
36    output_string " * Parsing Tokens...";
37    let ast = Parser.cdecls (WhiteSpace.lextoks tokens) (Lexing.from_string "") in
38    output_string " * Generating Global Data...";
39    let klass_data = get_data ast in
40    output_string " * Building Semantic AST...";
41    let sast = BuildSast.ast_to_sast klass_data in
42    output_string " * Deanonymizing Anonymous Classes.";
43    let (klass_data, sast) = do_deanon klass_data sast in
44    output_string " * Rebinding refinements.";
45    let sast = BuildSast.update_refinements klass_data sast in
46    output_string " * Generating C AST...";

```

```

46 GenCast.sast_to_cast klass_data sast
47
48 let main _ =
49   Printexc.record_backtrace true;
50   output_string "/* Starting Build Process...";
51   try
52     let source = source_cast () in
53     output_string " * Generating C...";
54     output_string " */";
55     GenC.cast_to_c source stdout;
56     print_newline ();
57     exit 0
58   with excn ->
59     let backtrace = Printexc.get_backtrace () in
60     let reraise = ref false in
61     let out = match excn with
62       | Failure(reason) -> Format.sprintf "Failed: %s\n" reason
63       | Invalid_argument(msg) -> Format.sprintf "Argument issue somewhere: %s\n"
64         msg
65       | Parsing.Parse_error -> "Parsing error."
66       | _ -> reraise := true; "Unknown Exception" in
67     Printf.fprintf stderr "%s\n%s\n" out backtrace;
68     if !reraise then raise(excn) else exit 1
69 let _ = main ()

```

Source 40: "ray.ml"

```

1  module StringMap = Map.Make (String);;
2
3  type class_def = { klass : string; parent : string option };;
4
5  let d1 = { klass = "myname"; parent = "Object" };;
6  let d3 = { klass = "myname2"; parent = "Object1" };;
7  let d4 = { klass = "myname3"; parent = "Object2" };;
8  let d2 = { klass = "myname1"; parent = "Object" };;
9
10 (*let myfunc cnameMap cdef =
11   if StringMap.mem cdef.parent cnameMap then
12     let cur = StringMap.find cdef.parent cnameMap in
13     StringMap.add cdef.parent (cdef.klass::cur) cnameMap
14   else
15     StringMap.add cdef.parent [cdef.klass] cnameMap;;
16
17 *)
18 let rec print_list = function
19   [] -> ()
20 | e::l -> print_string e ; print_string " " ; print_list l;;
21
22 let rec spitmap fst snd = print_string fst; print_list snd;;
23
24 let cnameMap =
25
26 let myfunc cnameMap cdef =
27   if StringMap.mem cdef.parent cnameMap then
28     let cur = StringMap.find cdef.parent cnameMap in
29     StringMap.add cdef.parent (cdef.klass::cur) cnameMap
30   else
31     StringMap.add cdef.parent [cdef.klass] cnameMap
32
33 in
34 List.fold_left
35   myfunc

```



```

36 StringMap.empty [d1;d2;d3;d4];;
37 StringMap.iter spitmap cnameMap;;
38
39 print_newline

```

Source 41: "unittest/bkup.ml"

```

1  module StringMap = Map.Make (String);;
2
3
4
5  type var_def = string * string;;
6  type func_def = {
7      returns : string option;
8      host    : string option;
9      name    : string;
10     static  : bool;
11     formals : var_def list;
12     (*body   : stmt list;*)
13 };;
14 type member_def = VarMem of var_def | MethodMem of func_def | InitMem of func_def;;
15
16 (* Things that can go in a class *)
17 type class_sections_def = {
18     privates : member_def list;
19     protects : member_def list;
20     publics  : member_def list;
21     (* refines : func_def list;
22        mains   : func_def list;*)
23 };;
24
25 type class_def = { klass : string; parent : string option; sections : class_sections_def;
26 };;
27
28 let sdef1 = {
29     privates = [VarMem("int", "a"); VarMem("int", "b")];;
30     protects = [VarMem("int", "c"); VarMem("int", "d")];;
31     publics  = [VarMem("int", "e"); VarMem("int", "f")];;
32 };;
33
34 let sdef2 = {
35     privates = [ VarMem("int", "g"); VarMem("int", "h")];;
36     protects = [ VarMem("int", "j"); VarMem("int", "i")];;
37     publics  = [ VarMem("int", "k"); VarMem("int", "l")];;
38 };;
39
40 let sdef3 = {
41     privates = [ VarMem("int", "m"); VarMem("int", "n")];;
42     protects = [ VarMem("int", "p"); VarMem("int", "o")];;
43     publics  = [ VarMem("int", "q"); VarMem("int", "r")];;
44 };;
45
46 let sdef4 = {
47     privates = [VarMem("int", "x"); VarMem("int", "s")];;
48     protects = [VarMem("int", "w"); VarMem("int", "t")];;
49     publics  = [VarMem("int", "v"); VarMem("int", "u")];;
50 };;
51 let d1 = { klass = "myname"; parent = Some("Object"); sections = sdef1 };;
52 let d3 = { klass = "myname2"; parent = Some("myname1"); sections = sdef3 };;
53 let d4 = { klass = "myname3"; parent = Some("myname2"); sections = sdef4 };;
54 let d2 = { klass = "myname1"; parent = Some("myname"); sections = sdef2 };;
55 (*
56 let myfunc cnameMap cdef =

```

```

56     if StringMap.mem cdef.parent cnameMap then
57         let cur = StringMap.find cdef.parent cnameMap in
58         StringMap.add cdef.parent (cdef.klass::cur) cnameMap
59     else
60         StringMap.add cdef.parent [cdef.klass] cnameMap;;
61
62 *)
63 let rec print_list = function
64 [] -> print_string "No more subclasses\n";
65 | e::l -> print_string e ; print_string "," ; print_list l;;
66
67 let rec spitmap fst scnd = print_string fst; print_string "->"; print_list scnd;;
68
69 let cnameMap =
70
71 let myfunc cnameMap cdef =
72
73     let cnameMap = StringMap.add cdef.klass [] cnameMap
74     in
75     let myparent =
76         match cdef.parent with
77         None -> "Object"
78         | Some str -> str
79     in
80     if StringMap.mem myparent cnameMap then
81         let cur = StringMap.find myparent cnameMap in
82         StringMap.add myparent (cdef.klass::cur) cnameMap
83     else
84         StringMap.add myparent [cdef.klass] cnameMap;
85
86
87 in
88     List.fold_left myfunc StringMap.empty [d1;d2;d3;d4];;
89 StringMap.iter spitmap cnameMap;;
90
91 let s2bmap =
92
93     let subtobase s2bmap cdef =
94         if StringMap.mem cdef.klass s2bmap then
95             (*how to raise exception*)
96             s2bmap
97         else
98             StringMap.add cdef.klass cdef.parent s2bmap
99
100     in
101     List.fold_left
102         subtobase
103         StringMap.empty [d1;d2;d3;d4];;
104
105 let rec spitmap fst snd = print_string fst; print_string "->";
106     match snd with
107     Some str -> print_string str; print_string "\n"
108     | None -> print_string "Object's parent is none\n";
109 in
110 StringMap.iter spitmap s2bmap;;
111
112 print_newline;;
113
114
115 print_string "getclassdef test\n\n";;
116 let rec getclassdef cname clist =
117     match clist with
118     [] -> None
119     | hd::tl -> if hd.klass = cname then Some(hd) else getclassdef cname tl;;
120

```

```

121 let print_cdef c = match c with None -> "No classdef" | Some c1 -> c1.klass;;
122 let print_pdef p = match p with None -> "No classdef" | Some p1 ->
123     (match p1.parent with None -> "No parent" | Some x -> x);;
124
125 let def1 = getclassdef "myname" [d1;d2;d3;d4];;
126 print_string (print_cdef def1);;
127 print_string "\n";;
128 print_string(print_pdef def1);;
129
130 print_string "\n\ngetmethoddef test\n";;
131
132
133
134 let rec getmemdef mname mlist =
135     match mlist with
136     [] -> None
137     | hd::tl -> match hd with
138         VarMem(typeid, varname) -> if varname = mname then Some(typeid) else
139             getmemdef mname tl
140         | _ -> None
141 ;;
142
143 (*Given a class definition and variable name, the lookupfield
144 lookup for the field in the privates, publics and protects list.
145 If found returns a (classname, accessspecifier, typeid, variablename) tuple
146 If not found returns a None*)
147 let lookupfield cdef vname =
148     let pmem = getmemdef vname cdef.sections.privates
149     in
150     match pmem with
151     Some def -> Some(cdef.klass, "private", vname, def)
152     | None ->
153         let pubmem = getmemdef vname cdef.sections.publics
154         in
155         match pubmem with
156         Some def -> Some(cdef.klass, "public", vname, def)
157         | None ->
158             let promem = getmemdef vname cdef.sections.protects
159             in
160             match promem with
161             Some def -> Some(cdef.klass, "protect", vname, def)
162             | None -> None
163 ;;
164
165 (*getfield takes classname and variablename;
166 looks for the class with the classname;
167 If classname found, lookup the variable in the class;
168 Else returns None
169 *)
170 let fstoffour (x,-,-,-) = x;;
171 let sndoffour (-,x,-,-) = x;;
172 let throffour (-,-,x,-) = x;;
173 let lstoffour (-,-,-,x) = x;;
174
175 let rec getfield cname vname cdeflist =
176     let classdef = getclassdef cname cdeflist
177     in
178     match classdef with
179     None ->
180         if cname = "Object" then
181             None
182         else
183             let basename = match(StringMap.find cname s2bmap) with Some b -> b | None ->

```

```

184         getfield basename vname cdeflist
185     | Some (cdef) -> lookupfield cdef vname;;
186
187 let field = getfield "myname3" "a" [d1;d2;d3;d4]
188 in
189 match field with
190 None -> print_string "field not found\n";
191 | Some tup -> print_string (fstoffour(tup));;

```

Source 42: "unittest/sast.ml"

```

1  %{
2  open Ast
3
4  (** Parser that reads from the scanner and produces an AST. *)
5
6  (** Set a single function to belong to a certain section *)
7  let set_func_section_to sect f = { f with section = sect }
8  (** Set a list of functions to belong to a certain section *)
9  let set_func_section sect = List.map (set_func_section_to sect)
10
11  (** Set a single member to belong to a certain subset of class memory.
12   This is necessary as a complicated function because init and main
13   can live in one of the several access levels. *)
14  let set_mem_section_to sect = function
15  | VarMem(v) -> VarMem(v)
16  | InitMem(func) -> InitMem({ func with section = sect })
17  | MethodMem(func) -> MethodMem({ func with section = sect })
18
19  (** Set a list of members to belong to a certain subset of class memory *)
20  let set_mem_section sect = List.map (set_mem_section_to sect)
21
22
23  (** Set the class of a func_def *)
24  let set_func_class aclass func = { func with inclass = aclass }
25
26  (** Set the class of a function member *)
27  let set_member_class aclass = function
28  | InitMem(func) -> InitMem(set_func_class aclass func)
29  | MethodMem(func) -> MethodMem(set_func_class aclass func)
30  | v -> v
31
32  (** Set the class of all sections *)
33  let set_func_class aclass sections =
34  let set_mems = List.map (set_member_class aclass) in
35  let set_funcs = List.map (set_func_class aclass) in
36  { privates = set_mems sections.privates;
37    publics = set_mems sections.publics;
38    protects = set_mems sections.protects;
39    refines = set_funcs sections.refines;
40    mains = set_funcs sections.mains }
41  %}
42
43  %token <int> SPACE
44  %token COLON NEWLINE
45  %token LPAREN RPAREN LBRACKET RBRACKET COMMA LBRACE RBRACE
46  %token PLUS MINUS TIMES DIVIDE MOD POWER
47  %token PLUSA MINUSA TIMESA DIVIDEA MODA POWERA
48  %token EQ NEQ GT LT GEQ LEQ AND OR NAND NOR XOR NOT
49  %token IF ELSE ELSIF WHILE
50  %token ASSIGN RETURN CLASS EXTEND SUPER INIT PRIVATE PROTECTED PUBLIC
51  %token NULL VOID THIS
52  %token NEW MAIN ARRAY

```

```

53 %token REFINABLE REFINE REFINES TO
54 %token SEMI COMMA DOT EOF
55
56 %token <string> TYPE
57 %token <int> ILIT
58 %token <float> FLIT
59 %token <bool> BLIT
60 %token <string> SLIT
61 %token <string> ID
62
63 /* Want to work on associativity when I'm a bit fresher */
64 %right ASSIGN PLUSA MINUSA TIMESA DIVIDEA MODA POWERA
65 %left OR NOR XOR
66 %left AND NAND
67 %left EQ NEQ
68 %left LT GT LEQ GEQ
69 %left PLUS MINUS
70 %left TIMES DIVIDE MOD
71 %nonassoc UMINUS
72 %left NOT POWER
73 %left LPAREN RPAREN LBRACKET RBRACKET
74 %left DOT
75
76 %start cdecls
77 %type <Ast.program> cdecls
78
79 %%
80
81 /* Classe and subclassing */
82 cdecls:
83     | cdecl { [$1] }
84     | cdecls cdecl { $2 :: $1 }
85 cdecl:
86     | CLASS TYPE extend_opt class_section_list
87     { { klass      = $2;
88         parent     = $3;
89         sections   = set_func_class $2 $4 } }
90 extend_opt:
91     | /* default */ { Some("Object") }
92     | EXTEND TYPE { Some($2) }
93
94 /* Class sections */
95 class_section_list:
96     | LBRACE class_sections RBRACE { $2 }
97 class_sections:
98     | /* Base Case */
99     { { privates = [];
100       protects = [];
101       publics = [];
102       refines = [];
103       mains = [] } }
104     | class_sections private_list { { $1 with privates = (set_mem_section Privates $2) @
105       $1.privates } }
105     | class_sections protect_list { { $1 with protects = (set_mem_section Protects $2) @
106       $1.protects } }
106     | class_sections public_list { { $1 with publics = (set_mem_section Publics $2) @
107       $1.publics } }
107     | class_sections refine_list { { $1 with refines = (set_func_section Refines $2) @
108       $1.refines } }
108     | class_sections main_method { { $1 with mains = (set_func_section_to Mains $2) ::
109       $1.mains } }
109
110 /* Refinements */
111 refine_list:
112     | REFINES LBRACE refinements RBRACE { $3 }

```

```

113 refinements:
114 | /* Can be empty */      { [] }
115 | refinements refinement { $2 :: $1 }
116 refinement:
117 | vartype ID DOT invocable { { $4 with returns = Some($1); host = Some($2) } }
118 | VOID ID DOT invocable   { { $4 with host = Some($2) } }
119
120 /* Private, protected, public members */
121 private_list:
122 | PRIVATE member_list    { $2 }
123 protect_list:
124 | PROTECTED member_list { $2 }
125 public_list:
126 | PUBLIC member_list     { $2 }
127
128 /* Members of such access groups */
129 member_list:
130 | LBRACE members RBRACE { $2 }
131 members:
132 | { [] }
133 | members member { $2 :: $1 }
134 member:
135 | vdecl semi { VarMem($1) }
136 | mdecl      { MethodMem($1) }
137 | init       { InitMem($1) }
138
139 /* Methods */
140 mdecl:
141 | vartype invocable { { $2 with returns = Some($1) } }
142 | VOID invocable   { $2 }
143
144 /* Constructors */
145 init:
146 | INIT callable { { $2 with name = "init" } }
147
148 /* Each class has an optional main */
149 main_method:
150 | MAIN callable { { $2 with name = "main"; static = true } }
151
152 /* Anything that is callable has these forms */
153 invocable:
154 | ID callable { { $2 with name = $1 } }
155 callable:
156 | formals stmt_block
157   { { returns = None;
158     host = None;
159     name = "";
160     static = false;
161     formals = $1;
162     body = $2;
163     section = Privates;
164     inclass = "";
165     uid = UID.uid_counter ();
166     builtin = false } }
167
168 /* Statements */
169 stmt_block:
170 | LBRACE stmt_list RBRACE { List.rev $2 }
171 stmt_list:
172 | /* nada */ { [] }
173 | stmt_list stmt { $2 :: $1 }
174 stmt:
175 | vdecl semi { Decl($1, None) }
176 | vdecl ASSIGN expr semi { Decl($1, Some($3)) }
177 | SUPER actuals semi { Super($2) }

```

```

178 | RETURN expr semi      { Return(Some($2)) }
179 | RETURN semi;          { Return(None) }
180 | conditional           { $1 }
181 | loop                  { $1 }
182 | expr semi             { Expr($1) }
183
184 /* Control Flow */
185 conditional:
186 | IF pred stmt_block else_list { If((Some($2), $3) :: $4) }
187 else_list:
188 | /* nada */              { [] }
189 | ELSE stmt_block          { [(None, $2)] }
190 | ELSIF pred stmt_block else_list { (Some($2), $3) :: $4 }
191 loop:
192 | WHILE pred stmt_block { While($2, $3) }
193 pred:
194 | LPAREN expr RPAREN { $2 }
195
196 /* Expressions */
197 expr:
198 | assignment             { $1 }
199 | invocation              { $1 }
200 | field                   { $1 }
201 | value                   { $1 }
202 | arithmetic              { $1 }
203 | test                    { $1 }
204 | instantiate             { $1 }
205 | refineexpr              { $1 }
206 | literal                 { $1 }
207 | LPAREN expr RPAREN     { $2 }
208 | THIS                    { This }
209 | NULL                    { Null }
210
211
212 assignment:
213 | expr ASSIGN expr        { Assign($1, $3) }
214 | expr PLUSA expr         { Assign($1, Binop($1, Arithmetic(Add), $3)) }
215 | expr MINUSA expr        { Assign($1, Binop($1, Arithmetic(Sub), $3)) }
216 | expr TIMESA expr        { Assign($1, Binop($1, Arithmetic(Prod), $3)) }
217 | expr DIVIDEA expr       { Assign($1, Binop($1, Arithmetic(Div), $3)) }
218 | expr MODA expr          { Assign($1, Binop($1, Arithmetic(Mod), $3)) }
219 | expr POWERA expr        { Assign($1, Binop($1, Arithmetic(Pow), $3)) }
220
221 invocation:
222 | expr DOT ID actuals { Invoc($1, $3, $4) }
223 | ID actuals { Invoc(This, $1, $2) }
224
225 field:
226 | expr DOT ID { Field($1, $3) }
227
228 value:
229 | ID { Id($1) }
230 | expr LBRACKET expr RBRACKET { Deref($1, $3) }
231
232 arithmetic:
233 | expr PLUS expr          { Binop($1, Arithmetic(Add), $3) }
234 | expr MINUS expr         { Binop($1, Arithmetic(Sub), $3) }
235 | expr TIMES expr         { Binop($1, Arithmetic(Prod), $3) }
236 | expr DIVIDE expr        { Binop($1, Arithmetic(Div), $3) }
237 | expr MOD expr           { Binop($1, Arithmetic(Mod), $3) }
238 | expr POWER expr         { Binop($1, Arithmetic(Pow), $3) }
239 | MINUS expr %prec UMINUS { Unop(Arithmetic(Neg), $2) }
240
241 test:
242 | expr AND expr { Binop($1, CombTest(And), $3) }

```

```

243 | expr OR expr      { Binop($1, CombTest(Or), $3) }
244 | expr XOR expr    { Binop($1, CombTest(Xor), $3) }
245 | expr NAND expr   { Binop($1, CombTest(Nand), $3) }
246 | expr NOR expr    { Binop($1, CombTest(Nor), $3) }
247 | expr LT expr     { Binop($1, NumTest(Less), $3) }
248 | expr LEQ expr    { Binop($1, NumTest(Leq), $3) }
249 | expr EQ expr     { Binop($1, NumTest(Eq), $3) }
250 | expr NEQ expr    { Binop($1, NumTest(Neq), $3) }
251 | expr GEQ expr    { Binop($1, NumTest(Geq), $3) }
252 | expr GT expr     { Binop($1, NumTest(Grtr), $3) }
253 | NOT expr        { Unop(CombTest(Not), $2) }
254 | REFINABLE LPAREN ID RPAREN { Refinable($3) }
255
256 instantiate:
257 | NEW vartype actuals { NewObj($2, $3) }
258 | NEW vartype actuals LBRACE refinements RBRACE { Anonymous($2, $3, List.map (
    set_func.class $2) $5) }
259
260 refineexpr:
261 | REFIN ID actuals TO vartype { Refine($2, $3, Some($5)) }
262 | REFIN ID actuals TO VOID    { Refine($2, $3, None) }
263
264 literal:
265 | lit { Literal($1) }
266
267 /* Literally necessary */
268 lit:
269 | SLIT { String($1) }
270 | ILIT { Int($1) }
271 | FLIT { Float($1) }
272 | BLIT { Bool($1) }
273
274 /* Parameter lists */
275 formals:
276 | LPAREN formals_opt RPAREN { $2 }
277 formals_opt:
278 | { [] }
279 | formals_list { List.rev $1 }
280 formals_list:
281 | vdecl { [$1] }
282 | formals_list COMMA vdecl { $3 :: $1 }
283
284 /* Arguments */
285 actuals:
286 | LPAREN actuals_opt RPAREN { $2 }
287 actuals_opt:
288 | { [] }
289 | actuals_list { List.rev $1 }
290 actuals_list:
291 | expr { [$1] }
292 | actuals_list COMMA expr { $3 :: $1 }
293
294 /* Variable declaration */
295 vdecl:
296 | vartype ID { ($1, $2) }
297 vartype:
298 | TYPE { $1 }
299 | vartype ARRAY { $1 ^ "[]" }
300
301 /* Eat multiple semis */
302 semi:
303 | SEMI {}
304 | semi SEMI {}

```


Source 43: "parser.mly"

```

1  open Ast
2  open Util
3  open StringModules
4  open GlobalData
5
6  (** Approximates a class *)
7  (**
8   From a class get the parent
9   @param aklass is a class_def to get the parent of
10  @return The name of the parent object
11  *)
12  let klass_to_parent aklass = match aklass with
13  | { klass = "Object" } -> raise(Invalid_argument("Cannot get parent of the root"))
14  | { parent = None; _ } -> "Object"
15  | { parent = Some(aklass); _ } -> aklass
16
17  (**
18   Utility function — place variables in left, methods (including init) in right
19   @param mem A member_def value (VarMem, MethodMem, InitMem)
20   @return Places the values held by VarMem in Left, values held by MethodMem or InitMem
21   in Right
22  *)
23  let member_split mem = match mem with
24  | VarMem(v) -> Left(v)
25  | MethodMem(m) -> Right(m)
26  | InitMem(i) -> Right(i)
27
28  (**
29   Stringify a section to be printed
30   @param section A class_section value (Privates, Protects, Publics, Refines, or Mains)
31   @return The stringification of the section for printing
32  *)
33  let section_string section = match section with
34  | Privates -> "private"
35  | Protects -> "protected"
36  | Publics -> "public"
37  | Refines -> "refinement"
38  | Mains -> "main"
39
40  (**
41   Return the variables of the class
42   @param aklass The class to explore
43   @return A list of ordered pairs representing different sections,
44   the first item of each pair is the type of the section, the second
45   is a list of the variables defs (type, name). Note that this only
46   returns pairs for Publics, Protects, and Privates as the others
47   cannot have variables
48  *)
49  let klass_to_variables aklass =
50  let vars members = fst (either_split (List.map member_split members)) in
51  let s = aklass.sections in
52  [(Publics, vars s.publics); (Protects, vars s.protects); (Privates, vars s.privates)]
53
54  (**
55   Return the methods of the class
56   @param aklass The class to explore
57   @return A list of ordered pairs representing different sections,
58   the first item of each pair is the type of the section, the second
59   is a list of the methods. Note that this only returns the methods

```

```

59     in Publics, Protects, or Privates as the other sections don't have
60     'normal' methods in them
61 *)
62 let klass_to_methods aklass =
63     let funcs members = snd (either_split (List.map member_split members)) in
64     let s = aklass.sections in
65     [(Publics, funcs s.publics); (Protects, funcs s.protects); (Privates, funcs s.
        privates)]
66
67 (**
68     Get anything that is invocable, not just instance methods
69     @param aklass The class to explore
70     @return The combined list of refinements, mains, and methods
71 *)
72 let klass_to_functions aklass =
73     let s = aklass.sections in
74     (Refines, s.refines) :: (Mains, s.mains) :: klass_to_methods aklass
75
76 (**
77     Return whether two function definitions have conflicting signatures
78     @param func1 A func_def
79     @param func2 A func_def
80     @return Whether the functions have the same name and the same parameter type sequence
81 *)
82 let conflicting_signatures func1 func2 =
83     let same_type (t1, _) (t2, _) = (t1 = t2) in
84     let same_name = (func1.name = func2.name) in
85     let same_params = try List.for_all2 same_type func1.formals func2.formals with
86         | Invalid_argument(-) -> false in
87     same_name && same_params
88
89 (**
90     Return a string that describes a function
91     @param func A func_def
92     @return A string showing the simple signature ([host.]name and arg types)
93 *)
94 let signature_string func =
95     let name = match func.host with
96     | None -> func.name
97     | Some(h) -> Format.sprintf "%s.%s" h func.name in
98     Format.sprintf "%s(%s)" name (String.concat ", " (List.map fst func.formals))
99
100 (**
101     Return a string representing the full signature of the function
102     @param func A func_def
103     @return A string showing the signature (section, [host.]name, arg types)
104 *)
105 let full_signature_string func =
106     let ret = match func.returns with
107     | None -> "Void"
108     | Some(t) -> t in
109     Format.sprintf "%s %s %s" (section_string func.section) ret (signature_string func)
110
111 (**
112     Given a class_data record, a class name, and a variable name, lookup the section and
113     type for that variable.
114     @param data A class_data record
115     @param klass_name The name of a class (string)
116     @param var_name The name of a variable (string)
117     @return Either None if the variable is not declared in the class or Some((section,
118     type))
119     where the variable is declared in section and has the given type.
120 *)
121 let class_var_lookup data klass_name var_name =

```

```

121     match map.lookup klass_name data.variables with
122     | Some(var_map) -> map.lookup var_name var_map
123     | - -> None
124
125 (**
126   Given a class_data record, a class_name, and a variable name, lookup the class in the
127   hierarchy
128   that provides access to that variable from within that class (i.e. private in that
129   class or
130   public / protected in an ancestor).
131   @param data A class_data record.
132   @param klass_name The name of a class (string)
133   @param var_name The name of a variable (string).
134   @return (class (string), type (string), class_section) option (None if not found).
135 *)
136 let class_field_lookup data klass_name var_name =
137   let var_lookup klass = class_var_lookup data klass var_name in
138   let rec lookup klass sections = match var_lookup klass, klass with
139   | Some((sect, vtype)), _ when List.mem sect sections -> Some((klass, vtype, sect))
140   | -, "Object" -> None
141   | -, _ -> lookup (StringMap.find klass data.parents) [Publics; Protects] in
142   lookup klass_name [Publics; Protects; Privates]
143
144 (**
145   Given a class_data record, a class name, a var_name, and whether the receiver of the
146   field lookup
147   is this, return the lookup of the field in the ancestry of the object. Note that this
148   restricts
149   things that should be kept protected (thus this thusly passed)
150   @param data A class_data record
151   @param klass_name The name of a class (string)
152   @param var_name The name of a variable (string)
153   @return Either the left of a triple (class found, type, section) or a Right of a
154   boolean, which
155   is true if the item was found but inaccessible and false otherwise.
156 *)
157 let class_field_far_lookup data klass_name var_name this =
158   match class_field_lookup data klass_name var_name with
159   | Some((klass, vtyp, section)) when this || section = Publics -> Left((klass,
160   vtyp, section))
161   | Some(_) -> Right(true)
162   | None -> Right(false)
163
164 (**
165   Given a class_data record, a class name, and a method name, lookup all the methods in
166   the
167   given class with that name.
168   @param data A class_data record
169   @param klass_name The name of a class (string)
170   @param func_name The name of a method (string)
171   @return A list of methods in the class with that name or the empty list if no such
172   method exists.
173 *)
174 let class_method_lookup data klass_name func_name =
175   match map.lookup klass_name data.methods with
176   | Some(method_map) -> map.lookup_list func_name method_map
177   | - -> []
178
179 (**
180   Given a class_data record, a class name, a method name, and whether the current
181   context is
182   'this' (i.e. if we want private / protected / etc), then return all methods in the
183   ancestry
184   of that class with that name (in the appropriate sections).

```

```

175     @param data A class_data record value
176     @param klass_name The name of a class.
177     @param method_name The name of a method to look up
178     @param this_search_mode — true means public/protected/private and then public/
    protected,
179     false is always public
180     @return A list of methods with the given name.
181 *)
182 let class_ancestor_method_lookup data klass_name method_name this =
183   let (startsects, recsects) = if this then ([Publics; Protects; Privates], [Publics;
    Protects]) else ([Publics], [Publics]) in
184   let rec find_methods found aklass sects =
185     let accessible f = List.mem f.section sects in
186     let funcs = List.filter accessible (class_method_lookup data aklass method_name)
    in
187     let found = funcs @ found in
188     if aklass = "Object" then found
189     else if method_name = "init" then found
190     else find_methods found (StringMap.find aklass data.parents) recsects in
191   find_methods [] klass_name startsects
192
193 (**
194   Given a class_data record, class name, method name, and refinement name, return the
    list of
195   refinements in that class for that method with that name.
196   @param data A class_data record value
197   @param klass_name A class name
198   @param method_name A method name
199   @param refinement_name A refinement name
200   @return A list of func_def values that match the given requirements. Note that this
    returns the
201   functions defined IN class name, not the ones that could be used INSIDE class name (
    via a refine
202   invocation). i.e. functions that may be invoked by the parent.
203 *)
204 let refine_lookup data klass_name method_name refinement_name =
205   match map_lookup klass_name data.refines with
206   | Some(map) -> map_lookup_list (method_name ^ "." ^ refinement_name) map
207   | - -> []
208
209 (**
210   Given a class_data record, a class name, a method name, and a refinement name, return
    the list
211   of refinements across all subclasses for the method with that name.
212   @param data A class_data record value
213   @param klass_name A class name
214   @param method_name A method name
215   @param refinement_name A refinement name
216   @return A list of func_def values that meet the criteria and may be invoked by this
    given method.
217   i.e. these are all functions residing in SUBCLASSES of the named class.
218 *)
219 let refinable_lookup data klass_name method_name refinement_name =
220   let refines = match map_lookup klass_name data.refinable with
221   | Some(map) -> map_lookup_list method_name map
222   | None -> [] in
223   List.filter (fun f -> f.name = refinement_name) refines
224
225 (**
226   Given a class_data record and two classes, returns the distance between them. If one
    is a proper
227   subtype of the other then Some(n) is returned where n is non-zero when the two
    classes are different
228   and comparable (one is a subtype of the other), zero when they are the same, and None
    when they are

```

```

229     incomparable (one is not a subtype of the other)
230     @param data A class_data record
231     @param klass1 A class to check the relation of to klass2
232     @param klass2 A class to check the relation of to klass1
233     @return An int option, None when the two classes are incomparable, Some(positive)
234     when klass2 is an
235     ancestor of klass1, Some(negative) when klass1 is an ancestor of klass2.
236 *)
237 let get_distance data klass1 klass2 =
238     (* We let these pop exceptions because that means bad programming on the compiler
239     * writers part, not on the GAMMA programmer's part (when klass1, klass2 aren't found
240     *)
241     let klass1_map = StringMap.find klass1 data.distance in
242     let klass2_map = StringMap.find klass2 data.distance in
243     match map_lookup klass2 klass1_map, map_lookup klass1 klass2_map with
244     | None, None -> None
245     | None, Some(n) -> Some(-n)
246     | res, _ -> res
247
248 (**
249     Check if a type exists in the class data — convenience function
250     @param data A class_data record
251     @param atype The name of a class (string)
252     @return True if the atype is a known type, false otherwise.
253 *)
254 let is_type data atype =
255     let lookup = try String.sub atype 0 (String.index atype '[') with
256     | Not_found -> atype in
257     StringSet.mem lookup data.known
258
259 (**
260     Check if a class is a subclass of another given a class_data record
261     @param data A class_data record
262     @param subtype A class name (string)
263     @param supertype A class name (string)
264     @return Whether subtype has supertype as an ancestor given data.
265     Note that this is true when the two are equal (trivial ancestor).
266 *)
267 let is_subtype data subtype supertype =
268     let basetype s = try let n = String.index s '[' in String.sub s 0 n with Not_found ->
269     s in
270     match get_distance data (basetype subtype) (basetype supertype) with
271     | Some(n) when n >= 0 -> true
272     | _ -> false
273
274 (**
275     Check if a class is a proper subclass of another given a class_data record
276     @param data A class_data record
277     @param subtype A class name (string)
278     @param supertype A class name (string)
279     @return Whether subtype has supertype as an ancestor given data.
280     Note that this IS NOT true when the two are equal (trivial ancestor).
281 *)
282 let is_proper_subtype data subtype supertype =
283     match get_distance data subtype supertype with
284     | Some(n) when n > 0 -> true
285     | _ -> false
286
287 (**
288     Return whether a list of actuals and a list of formals are compatible.
289     For this to be true, each actual must be a (not-necessarily-proper) subtype
290     of the formal at the same position. This requires that both be the same
291     in quantity, obviously.
292     @param data A class_data record (has type information)

```

```

291   @param actuals A list of the types (and just the types) of the actual arguments
292   @param formals A list of the types (and just the types) of the formal arguments
293   @return Whether the actual arguments are compatible with the formal arguments.
294   *)
295   let compatible_formals data actuals formals =
296     let compatible formal actual = is_subtype data actual formal in
297     try List.for_all2 compatible formals actuals with
298       | Invalid_argument(-) -> false
299
300   (**
301    Return whether a given func_def is compatible with a list of actual arguments.
302    This means making sure that it has the right number of formal arguments and that
303    each actual argument is a subtype of the corresponding formal argument.
304    @param data A class_data record (has type information)
305    @param actuals A list of the types (and just the types) of the actual arguments
306    @param func A func_def from which to get formals
307    @return Whether the given func_def is compatible with the actual arguments.
308   *)
309   let compatible_function data actuals func =
310     compatible_formals data actuals (List.map fst func.formals)
311
312   (**
313    Return whether a function's return type is compatible with a desired return type.
314    Note that if the desired return type is None then the function is compatible.
315    Otherwise if it is not None and the function's is, then it is not compatible.
316    Lastly, if the desired type is a supertype of the function's return type then the
317    function is compatible.
318    @param data A class_data record value
319    @param ret_type The desired return type
320    @param func A func_def to test.
321    @return True if compatible, false if not.
322   *)
323   let compatible_return data ret_type func =
324     match ret_type, func.returns with
325     | None, _ -> true
326     | _, None -> false
327     | Some(desired), Some(given) -> is_subtype data given desired
328
329   (**
330    Return whether a function's signature is completely compatible with a return type
331    and a set of actuals
332    @param data A class_data record value
333    @param ret_type The return type (string option)
334    @param actuals The list of actual types
335    @param func A func_def value
336    @return True if compatible, false if not.
337   *)
338   let compatible_signature data ret_type actuals func =
339     compatible_return data ret_type func && compatible_function data actuals func
340
341   (**
342    Filter a list of functions based on their section.
343    @param funcs a list of functions
344    @param sects a list of class_section values
345    @return a list of functions in the given sections
346   *)
347   let in_section sects funcs =
348     List.filter (fun f -> List.mem f.section sects) funcs
349
350   (**
351    Given a class_data record, a list of actual arguments, and a list of methods,
352    find the best matches for the actuals. Note that if there are multiple best
353    matches (i.e. ties) then a non-empty non-singleton list is returned.
354    Raises an error if somehow our list of compatible methods becomes incompatible
355    [i.e. there is a logic error in the compiler].

```

```

356 @param data A class_data record
357 @param actuals The list of types (and only types) for the actual arguments
358 @param funcs The list of candidate functions
359 @return The list of all best matching functions (should be at most one, we hope).
360 *)
361 let best_matching_signature data actuals funcs =
362   let funcs = List.filter (compatible.function data actuals) funcs in
363   let distance_of actual formal = match get_distance data actual formal with
364     | Some(n) when n >= 0 -> n
365     | _ -> raise(Invalid_argument("Compatible methods somehow incompatible: " ^
actual ^ " vs. " ^ formal ^ ". Compiler error.)) in
366   let to_distance func = List.map2 distance_of actuals (List.map fst func.formals) in
367   let with_distances = List.map (fun func -> (func, to_distance func)) funcs in
368   let lex_compare (_, lex1) (_, lex2) = lexical.compare lex1 lex2 in
369   List.map fst (find_all_min lex_compare with_distances)
370
371 (**
372   Given a class_data record, method name, and list of actuals, and a list of sections
373   to consider,
374   get the best matching method. Note that if there is more than one then an exception
375   is raised
376   as this should have been reported during collision detection [compiler error].
377   @param data A class_data record
378   @param method_name The name to lookup candidates for
379   @param actuals The list of types (and only types) for the actual arguments
380   @param sections The sections to filter on (only look in these sections)
381   @return Either None if no function is found, Some(f) if one function is found, or an
382   error is raised.
383   *)
384 let best_method data klass_name method_name actuals sections =
385   let methods = class.method.lookup data klass_name method_name in
386   let methods = in_section sections methods in
387   match best_matching_signature data actuals methods with
388   | [] -> None
389   | [func] -> Some(func)
390   | _ -> raise(Invalid_argument("Multiple methods named " ^ method_name ^ " of the
same signature in " ^ klass_name ^ "; Compiler error.))
391
392 let best_inherited_method data klass_name method_name actuals this =
393   let methods = class.ancestor_method.lookup data klass_name method_name this in
394   match best_matching_signature data actuals methods with
395   | [] -> None
396   | [func] -> Some(func)
397   | _ -> raise(Invalid_argument("Multiple methods named " ^ method_name ^ " of the
same signature inherited in " ^ klass_name ^ "; Compiler error.))
398
399 (**
400   Given the name of a refinement to apply, the list of actual types,
401   find the compatible refinements via the data / klass_name / method_name.
402   Partition the refinements by their inklass value and then return a list
403   of the best matches from each partition.
404   @param data A class_data record value
405   @param klass_name A class name
406   @param method_name A method name
407   @param refine_name A refinement name
408   @param actuals The types of the actual arguments
409   @return A list of functions to switch on based on the actuals.
410   *)
411 let refine_on data klass_name method_name refine_name actuals ret_type =
412   (* These are all the refinements available from subclasses *)
413   let refines = refinable.lookup data klass_name method_name refine_name in
414   (* Compatible functions *)
415   let compat = List.filter (compatible.signature data ret_type actuals) refines in

```

```

415 (* Organize by inclass *)
416 let to_class map f = add_map_list f.inclass f map in
417 let by_class = List.fold_left to_class StringMap.empty compat in
418
419 (* Now make a map of only the best *)
420 let best_funcs = match best_matching_signature data actuals_funcs with
421 | [func] -> func
422 | _ -> raise (Failure ("Compiler error finding a unique best refinement.")) in
423 let to_best klass funcs map = StringMap.add klass (best_funcs) map in
424 let best_map = StringMap.fold to_best by_class StringMap.empty in
425
426 (* Now just return the bindings from the best *)
427 List.map snd (StringMap.bindings best_map)
428
429 (**
430  Get the names of the classes in level order (i.e. from root down).
431  @param data A class_data record
432  @return The list of known classes, from the root down.
433  *)
434 let get_class_names data =
435   let kids aklass = map_lookup_list aklass data.children in
436   let rec append found = function
437     | [] -> List.rev found
438     | items -> let next = List.flatten (List.map kids items) in
439       append (items@found) next in
440   append [] ["Object"]
441
442
443 (**
444  Get leaf classes
445  @param data A class_data record
446  @return A list of leaf classes
447  *)
448 let get_leaves data =
449   let is_leaf f = match map_lookup_list f data.children with
450   | [] -> true
451   | _ -> false in
452   let leaves = StringSet.filter is_leaf data.known in
453   StringSet.elements leaves

```

Source 44: "Klass.ml"

```

1  all: compile _tools _ray _doc
2
3  compile:
4    #Generate the lexer and parser
5    ocamllex scanner.mll
6    ocamlyacc parser.mly
7
8    ocamlc -c -g Ast.mli
9    ocamlc -c -g UID.ml
10
11    ocamlc -c -g parser.mli
12    ocamlc -c -g scanner.ml
13    ocamlc -c -g parser.ml
14
15    ocamlc -c -g WhiteSpace.ml
16    ocamlc -c -g Inspector.mli
17    ocamlc -c -g Inspector.ml
18    ocamlc -c -g Pretty.ml
19
20    ocamlc -c -g Util.ml
21    ocamlc -c -g StringModules.ml

```



```

22     ocamlc -c -g GlobalData.mli
23     ocamlc -c -g Klass.mli
24     ocamlc -c -g KlassData.mli
25     ocamlc -c -g BuiltIns.mli
26     ocamlc -c -g BuiltIns.ml
27     ocamlc -c -g Klass.ml
28     ocamlc -c -g KlassData.ml
29     ocamlc -c -g Variables.ml
30     ocamlc -c -g Sast.mli
31     ocamlc -c -g BuildSast.mli
32     ocamlc -c -g BuildSast.ml
33     ocamlc -c -g Unanonymous.mli
34     ocamlc -c -g Unanonymous.ml
35     ocamlc -c -g Cast.mli
36     ocamlc -c -g GenCast.ml
37     ocamlc -c -g GenC.ml
38     ocamlc -c -g Debug.ml
39
40     ocamlc -c -g classinfo.ml
41     ocamlc -c -g inspect.ml
42     ocamlc -c -g prettify.ml
43     ocamlc -c -g streams.ml
44     ocamlc -c -g canonical.ml
45     ocamlc -c -g freevars.ml
46     ocamlc -c -g ray.ml
47
48     _tools:
49         #Make the tools
50         ocamlc -g -o tools/prettify UID.cmo scanner.cmo parser.cmo Inspector.cmo Pretty.cmo
51         WhiteSpace.cmo prettify.cmo
52         ocamlc -g -o tools/inspect UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.
53         cmo inspect.cmo
54         ocamlc -g -o tools/streams UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.
55         cmo streams.cmo
56         ocamlc -g -o tools/canonical UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.
57         cmo canonical.cmo
58         ocamlc -g -o tools/freevars UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.
59         cmo Util.cmo StringModules.cmo str.cma BuiltIns.cmo Klass.cmo KlassData.cmo Debug.cmo
60         Variables.cmo freevars.cmo
61         ocamlc -g -o tools/classinfo UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.
62         cmo Util.cmo StringModules.cmo str.cma BuiltIns.cmo Klass.cmo KlassData.cmo classinfo
63         .cmo
64
65     _ray:
66         #Make ray
67         mkdir -p bin
68         ocamlc -g -o bin/ray UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.cmo Util
69         .cmo StringModules.cmo str.cma BuiltIns.cmo Klass.cmo KlassData.cmo Debug.cmo
70         Variables.cmo BuildSast.cmo Unanonymous.cmo GenCast.cmo GenC.cmo ray.cmo
71
72     nodoc: compile _tools _ray
73
74     docsources = Ast.mli BuildSast.ml BuildSast.mli BuiltIns.ml BuiltIns.mli Cast.mli Debug.
75     ml GenCast.ml GenC.ml GlobalData.mli Inspector.ml Inspector.mli Klass.ml Klass.mli
76     KlassData.ml KlassData.mli Pretty.ml Sast.mli StringModules.ml UID.ml Unanonymous.ml
77     Unanonymous.mli Util.ml Variables.ml WhiteSpace.ml parser.ml parser.mli scanner.ml
78     docgen = ./doc/.docgen
79
80     _doc:
81         #Generate the documentation
82         mkdir -p doc
83         ocamlc -g -o doc/docgen $(docgen) -keep-code $(docsources)
84         ocamlc -g -o doc/docgen $(docgen) -d doc -t "The Ray Compiler" -html -colorize -
85         code -all-params
86         ocamlc -g -o doc/docgen $(docgen) -dot -o ".doc/ray-modules.dot"

```

```

73   ocaml doc -hide-warnings -load $(docgen) -dot -dot-types -o "./doc/ray-types.dot"
74
75   bleach:
76       rm *.cmi *.cmo parser.ml parser.mli scanner.ml
77       rm -r ./doc
78
79   clean:
80       rm *.cmi *.cmo parser.ml parser.mli scanner.ml
81
82   cleantools:
83       rm tools/{prettify,inspect,streams,canonical,freevars,classinfo}

```

Source 45: "Makefile"

```

1
2   val ast_to_sast_klass : GlobalData.class_data -> Ast.class_def -> Sast.class_def
3   val ast_to_sast : GlobalData.class_data -> Sast.class_def list
4   val update_refinements : GlobalData.class_data -> Sast.class_def list -> Sast.class_def
   list

```

Source 46: "BuildSast.mli"

```

1   /* N queens iterative solution */
2
3   class ChessBoard:
4       public:
5           init(Integer size):
6               super()
7               n := size
8               solution_count := 0
9               arrangement := new Integer[](n)
10              Integer i := 0
11              while(i < n):
12                  arrangement[i] := -1
13                  i += 1
14
15              Boolean test_column(Integer row):
16                  Integer i := 0
17                  while(i < row):
18                      if(arrangement[i] == arrangement[row]):
19                          return false
20                      i += 1
21                  return true
22
23              Boolean test_diag(Integer row):
24                  Integer i := 0
25                  while(i < row):
26                      if(((arrangement[row] - arrangement[i]) == row - i) or ((arrangement[row] -
arrangement[i]) == i - row)):
27                          return false
28                      i += 1
29                  return true
30
31              Boolean test(Integer row):
32                  if(test_column(row) and test_diag(row)):
33                      return true
34                  else:
35                      return false
36
37              Integer print_board():
38                  system.out.printString("\nSolution # ")

```

```

39     system.out.printInteger(solution_count)
40     system.out.printString("\n")
41     Integer r := 0
42     while(r < n):
43         Integer c := 0
44         while(c < n):
45             if(arrangement[r] == c):
46                 system.out.printString("Q ")
47             else:
48                 system.out.printString("* ")
49             c += 1
50         system.out.printString("\n")
51         r += 1
52     return 0
53
54 Integer get_solutions():
55     arrangement[0] := -1
56     Integer row := 0
57     while(row >= 0):
58         arrangement[row] += 1
59         while(arrangement[row] < n and not test(row)):
60             arrangement[row] += 1
61         if(arrangement[row] < n):
62             if(row == n - 1):
63                 solution_count += 1
64                 print_board()
65             else:
66                 row += 1
67                 arrangement[row] := -1
68         else:
69             row -= 1
70     return 0
71
72 private:
73     Integer n
74     Integer solution_count
75     Integer[] arrangement
76
77 main(System system, String[] args):
78     system.out.printString("Chess board size: ")
79     Integer size := system.in.scanInteger()
80     ChessBoard nqueens := new ChessBoard(size)
81     nqueens.get_solutions()

```

Source 47: "demo/nqueens.gamma"

```

1 class HelloWorld:
2     public:
3         String greeting
4         init():
5             super()
6             greeting := "Hello World!"
7
8     main(System system, String[] args):
9         HelloWorld hw := new HelloWorld()
10        system.out.printString(hw.greeting)
11        system.out.printString("\n")

```

Source 48: "demo/helloworld.gamma"

```

1 class Bank:

```

```

2 public:
3     init():
4         super()
5         id_counter := 0
6         accounts := new Account[(100)
7
8         /* Anonymous instantiation can 'get around' protected constructors */
9         Account president := (new Account(id_counter, "Bank President") {
10             Float apply_interest.rate() { return 0.10; }
11         })
12         accounts[id_counter] := president
13         id_counter += 1
14
15 Integer open_checking(String client_name):
16     Account new_account := new Checking(id_counter, client_name)
17     accounts[id_counter] := new_account
18     id_counter += 1
19     return id_counter-1
20
21 Integer open_savings(String client_name):
22     Account new_account := new Savings(id_counter, client_name)
23     accounts[id_counter] := new_account
24     id_counter += 1
25     return id_counter-1
26
27 Integer apply_interest(Integer id):
28     if(id > id_counter or id < 0):
29         return 1
30     accounts[id].apply_interest()
31     return 0
32
33 Float get_balance(Integer id):
34     if(id > id_counter):
35         system.out.println("Invalid account number.\n")
36         return -1.0
37     return accounts[id].get_balance()
38
39 Integer deposit(Integer id, Float amount):
40     if(id > id_counter):
41         system.out.println("Invalid account number.\n")
42         return 1
43
44     accounts[id].deposit(amount)
45     return 0
46
47 Integer withdraw(Integer id, Float amount):
48     if(id > id_counter):
49         system.out.println("Invalid account number.\n")
50         return 1
51     if(amount > accounts[id].get_balance()):
52         return 1
53
54     accounts[id].withdraw(amount)
55     return 0
56
57 Integer transfer(Integer from_id, Integer to_id, Float amount):
58     if(from_id > id_counter):
59         system.out.println("Invalid account number.\n")
60         return 1
61     if(accounts[from_id].get_balance() < amount):
62         system.out.println("Insufficient funds.\n")
63         return 1
64     accounts[from_id].withdraw(amount)
65     accounts[to_id].deposit(amount)
66     return 0

```

```

67     Float get_balance(Integer id, Float amount):
68         if(id > id_counter):
69             return -1.0
70         return accounts[id].get_balance()
71
72
73
74     protected:
75         Integer id_counter
76         Account[] accounts
77
78     /* Subclasses can come before classes if you like */
79     class Checking extends Account:
80         public:
81             init(Integer id, String name):
82                 super(id, name)
83
84         refinement:
85             Float apply_interest.rate():
86                 return 0.005
87
88     class Savings extends Account:
89         public:
90             init(Integer id, String name):
91                 super(id, name)
92
93         refinement:
94             Float apply_interest.rate():
95                 return 0.02
96
97     class Account:
98         protected:
99             void apply_interest(Boolean check):
100                 if (not (refinable(rate))):
101                     system.out.println("Account must have some interest rate.\n")
102                     system.exit(1)
103
104             init(Integer new_id, String name):
105                 super()
106                 apply_interest(false)
107
108                 id := new_id
109                 client := name
110                 balance := 0.0
111                 transactions := new Float[](100)
112                 trans_len := 0
113
114         public:
115             Integer get_id():
116                 return id
117
118             String get_client_name():
119                 return client
120
121             Float get_balance():
122                 return balance
123
124             void apply_interest():
125                 balance *= (1.0 + (refine rate() to Float))
126
127             Integer deposit(Float amount):
128                 if(amount < 0.0):
129                     return 1
130                 balance += amount
131                 transactions[trans_len] := amount

```

```

132         trans_len += 1
133         return 0
134
135     Integer withdraw(Float amount):
136         if (amount < 0.0):
137             system.out.println("Invalid number entered.\n")
138             return 1
139         if (balance < amount):
140             system.out.println("Insufficient funds.\n")
141             return 1
142         balance -= amount
143         return 0
144
145     private:
146         Integer id
147         String client
148         Float balance
149         Float[] transactions
150         Integer trans_len
151
152
153     class Main:
154     public:
155         init():
156             super()
157
158     main(System system, String[] args):
159         Bank citibank := new Bank()
160         Integer menu_lvl := 0
161         Integer menu_num := 0
162         Integer selection := new Integer()
163         Integer account_id := -1
164
165         while (true):
166             if (menu_lvl == 0):
167                 system.out.println("Please Select:\n1.Open New Account\n2.Manage Existing
Account\n3.I'm the President!\n-> ")
168                 selection := system.in.nextInt()
169                 account_id := -1
170                 menu_lvl := 1
171
172             if (menu_lvl == 1):
173                 if (selection == 1):
174                     system.out.println("Your Name Please:")
175                     String name := new String()
176                     name := system.in.nextLine()
177                     Integer checking_id := citibank.open_checking(name)
178                     Integer savings_id := citibank.open_savings(name)
179
180                     system.out.println("\nDear ")
181                     system.out.println(name)
182                     system.out.println("\n")
183                     system.out.println("Your new checking account number: ")
184                     system.out.println(checking_id)
185                     system.out.println("\n")
186                     system.out.println("Your new savings account number: ")
187                     system.out.println(savings_id)
188                     system.out.println("\n")
189                     selection := 0
190                     menu_lvl := 0
191                 else:
192                     if (selection == 2):
193                         if (account_id < 0):
194                             system.out.println("Your Account Number Please: ")
195                             account_id := system.in.nextInt()

```

```

196         citibank.apply_interest(account_id)
197         system.out.println(" Please Select:\n1.Check Balance\n2.Deposit\n3.
198 Withdraw\n4.Transfer\n5.Exit\n-> ")
199         menu_lvl := 2
200         selection := system.in.scanInteger()
201         if(selection == 5):
202             selection := 0
203             menu_lvl := 0
204         else:
205             if(selection == 3):
206                 selection := 2
207                 account_id := 0
208                 menu_lvl := 1
209
210         if(menu_lvl == 2):
211             if(selection == 1):
212                 system.out.println("Your current balance: ")
213                 system.out.printFloat(citibank.get_balance(account_id))
214                 system.out.println("\n")
215                 menu_lvl := 1
216                 selection := 2
217             else:
218                 if(selection == 2):
219                     system.out.println("Please enter the amount you want to deposit: ")
220                     Float amount := system.in.scanFloat()
221                     citibank.deposit(account_id, amount)
222                     menu_lvl := 1
223                     selection := 2
224                 else:
225                     if(selection == 3):
226                         system.out.println("Pleaser enter the amount you want to withdraw: ")
227                         Float amount := system.in.scanFloat()
228                         citibank.withdraw(account_id, amount)
229                         menu_lvl := 1
230                         selection := 2
231                     else:
232                         if(selection == 4):
233                             system.out.println("Please enter the account number you want to
234 transfer to: ")
235                             Integer to_account := system.in.scanInteger()
236                             system.out.println("Please enter the amount you want to transfer: ")
237                             Float amount := system.in.scanFloat()
238                             citibank.transfer(account_id, to_account, amount)
239                             menu_lvl := 1
240                             selection := 2

```

Source 49: "demo/bank.gamma"

```

1  open Parser
2
3  (** Convert a whitespace file into a brace file. *)
4
5  (**
6   Gracefully tell the programmer that they done goofed
7   @param msg The descriptive error message to convey to the programmer
8   *)
9  let wsfail msg = raise(Failure(msg))
10
11  (**
12   Only allow spacing that is at the start of a line
13   @param program A program as a list of tokens
14   @return a list of tokens where the only white space is indentation, newlines,

```

```

15     and colons (which count as a newline as it must be followed by them)
16 *)
17 let indenting_space program =
18   let rec space_indenting rtokens = function
19     | NEWLINE::SPACE(n)::rest -> space_indenting (SPACE(n)::NEWLINE::rtokens) rest
20     | COLON::SPACE(n)::rest -> space_indenting (SPACE(n)::COLON::rtokens) rest
21     | SPACE(n)::rest -> space_indenting rtokens rest
22     | token::rest -> space_indenting (token::rtokens) rest
23     | [] -> List.rev rtokens in
24   match (space_indenting [] (NEWLINE::program)) with
25   | NEWLINE::rest -> rest
26   | _ -> wsfail "Indenting should have left a NEWLINE at the start of program; did
27 not."
28
29 (**
30  Between LBRACE and RBRACE we ignore spaces and newlines; colons are errors in this
31  context.
32  It's not necessary that this be done after the above, but it is recommended.
33  @param program A program in the form of a list of tokens
34  @return A slightly slimmer program
35 *)
36 let despace_brace program =
37   let rec brace_despace depth tokens rtokens last =
38     if depth > 0 then
39       match tokens with
40       | SPACE(_)::rest -> brace_despace depth rest rtokens last
41       | NEWLINE::rest -> brace_despace depth rest rtokens last
42       | COLON::_ -> wsfail "Colon inside brace scoping."
43       | LBRACE::rest -> brace_despace (depth+1) rest (LBRACE::rtokens) last
44       | RBRACE::rest -> let rtokens = if depth = 1
45                             then SPACE(last)::NEWLINE::RBRACE::rtokens
46                             else RBRACE::rtokens in
47                             brace_despace (depth-1) rest rtokens last
48       | token::rest -> brace_despace depth rest (token::rtokens) last
49       | [] -> List.rev rtokens
50     else
51       match tokens with
52       | SPACE(n)::rest -> brace_despace depth rest (SPACE(n)::rtokens) n
53       | LBRACE::rest -> brace_despace (depth+1) rest (LBRACE::rtokens) last
54       | token::rest -> brace_despace depth rest (token::rtokens) last
55       | [] -> List.rev rtokens in
56   brace_despace 0 program [] 0
57
58 (**
59  Remove empty indentation — SPACE followed by COLON or NEWLINE
60  @param program A program as a list of tokens
61  @return A program without superfluous indentation
62 *)
63 let trim_lines program =
64   let rec lines_trim tokens rtokens =
65     match tokens with
66     | [] -> List.rev rtokens
67     | SPACE(_)::NEWLINE::rest -> lines_trim rest (NEWLINE::rtokens)
68     | SPACE(_)::COLON::rest -> lines_trim rest (COLON::rtokens)
69     | token::rest -> lines_trim rest (token::rtokens) in
70   lines_trim program []
71
72 (**
73  Remove consecutive newlines
74  @param program A program as a list of tokens
75  @return A program without consecutive newlines
76 *)
77 let squeeze_lines program =
78   let rec lines_squeeze tokens rtokens =
79     match tokens with

```



```

78         | [] -> List.rev rtokens
79         | NEWLINE::NEWLINE::rest -> lines_squeeze (NEWLINE::rest) rtokens
80         | COLON::NEWLINE::rest -> lines_squeeze (COLON::rest) rtokens (* scanner
handled this though *)
81         | token::rest -> lines_squeeze rest (token::rtokens) in
82     lines_squeeze program []
83
84 (**
85     Remove the initial space from a line but semantically note it
86     @return an ordered pair of the number of spaces at the beginning
87     of the line and the tokens in the line
88 *)
89 let spacing = function
90     | SPACE(n)::rest -> (n, rest)
91     | list             -> (0, list)
92
93 (**
94     Remove spaces, newlines, and colons but semantically note their presence.
95     @param program A full program (transformed by the above pipeline)
96     @return a list of triples, one for each line. Each triple's first item is
97     the number of spaces at the beginning of the line; the second item is the
98     tokens in the line; the third is whether the line ended in a colon.
99 *)
100 let tokens_to_lines program =
101     let rec lines_from_tokens rline rlines = function
102         | NEWLINE::rest ->
103             (match rline with
104              | [] -> lines_from_tokens [] rlines rest
105              | - -> let (spacer, line) = spacing (List.rev rline) in
106                     lines_from_tokens [] ((spacer, line, false)::rlines) rest
107         )
108         | COLON::rest ->
109             (match rline with
110              | [] -> lines_from_tokens [] rlines rest
111              | - -> let (spacer, line) = spacing (List.rev rline) in
112                     lines_from_tokens [] ((spacer, line, true)::rlines) rest)
113         | [] ->
114             (match rline with
115              | [] -> List.rev rlines
116              | - -> let (spacer, line) = spacing (List.rev rline) in
117                     lines_from_tokens [] ((spacer, line, false)::rlines) [])
118         | token::rest -> lines_from_tokens (token::rline) rlines rest in
119     lines_from_tokens [] [] program
120
121 (**
122     Merge line continuatons given output from tokens_to_lines.
123     Line n+1 continues n if n does not end in a colon and n+1 is more
124     indented than n (or if line n is a continuation and they are both
125     equally indented).
126     @param program_lines The individual lines of the program
127     @return The lines of the program with whitespace collapsed
128 *)
129 let merge_lines program_lines =
130     let rec lines_merge rlines = function
131         | ((n1, -, -) as line1)::((n2, -, -) as line2)::rest when n1 >= n2 -> lines_merge
132             (line1::rlines) (line2::rest)
133         | (n, line1, false)::(-, line2, colon)::rest -> lines_merge rlines ((n,
134             line1@line2, colon)::rest)
135         | ((-, -, true) as line)::rest -> lines_merge (line::rlines) rest
136         | line::[] -> lines_merge (line::rlines) []
137         | [] -> List.rev rlines in
138     lines_merge [] program_lines
139
140 (**
141     Check if a given line needs a semicolon at the end

```

```

139 *)
140 let rec needs_semi = function
141 | [] -> true (* General base case *)
142 | RBRACE::[] -> false (* The end of bodies do not require semicolons *)
143 | SEMI::[] -> false (* A properly terminated line does not require an
additional semicolon *)
144 | _::rest -> needs_semi rest (* Go through *)
145
146 (**
147 Build a block. Consecutive lines of the same indentation with only the last ending
148 in a colon are a 'block'. Blocks are just 'lines' merged together but joined with
149 a semi colon when necessary.
150 @param lines The full set of lines
151 @return A list of blocks
152 *)
153 let block_merge lines =
154 let add_semi = function
155 | (n, toks, true) -> (n, toks, true, false)
156 | (n, toks, false) -> (n, toks, false, needs_semi toks) in
157 let lines = List.map add_semi lines in
158 let rec merge_blocks rblocks = function
159 | (n1, line1, false, s1)::(n2, line2, colon, s2)::rest when n1 = n2 ->
160 let newline = line1 @ (if s1 then [SEMI] else []) @ line2 in
161 merge_blocks rblocks ((n1, newline, colon, s2)::rest)
162 | (n, line, colon, _)::rest -> merge_blocks ((n, line, colon)::rblocks) rest
163 | [] -> List.rev rblocks in
164 merge_blocks [] lines
165
166 (** Make sure every line is terminated with a semi-colon when necessary *)
167 let terminate_blocks blocks =
168 let rec block_terminate rblocks = function
169 | (n, toks, false)::rest ->
170 let terminated = if (needs_semi toks) then toks@[SEMI] else toks in
171 block_terminate ((n, terminated, false)::rblocks) rest
172 | other::rest ->
173 block_terminate (other::rblocks) rest
174 | [] -> List.rev rblocks in
175 block_terminate [] blocks
176
177 (** Pops the stack and adds rbraces when necessary *)
178 let rec arrange n stack rtokens =
179 match stack with
180 | top::rest when n <= top -> arrange n rest (RBRACE::rtokens)
181 | _ -> (stack, rtokens)
182
183 (**
184 Take results of pipeline and finally adds braces. If blocks are merged
185 then either consecutive lines differ in scope or there are colons.
186 so now everything should be easy peasy (lemon squeezy).
187 *)
188 let space_to_brace = function
189 | [] -> []
190 | linelist -> let rec despace_enbrace stack rtokens = function
191 | [] -> List.rev ((List.map (function _ -> RBRACE) stack) @ rtokens)
192 | (n, line, colon)::rest ->
193 let (stack, rtokens) = arrange n stack rtokens in
194 let (lbrace, stack) = if colon then ([LBRACE], n::stack) else ([], stack) in
195 despace_enbrace stack (lbrace@(List.rev line)@rtokens) rest
196 in despace_enbrace [] [] linelist
197
198 (** Drop the EOF from a stream of tokens, failing if not possible *)
199 let drop_eof program =
200 let rec eof_drop rtokens = function
201 | EOF::[] -> List.rev rtokens
202 | EOF::rest -> raise(Failure("Misplaced EOF"))

```

```

203 | [] -> raise(Failure("No EOF available.))
204 | tk::tks -> eof_drop (tk::rtokens) tks in
205 eof_drop [] program
206
207 (** Append an eof token to a program *)
208 let append_eof program =
209   let rec eof_add rtokens = function
210     | [] -> List.rev (EOF::rtokens)
211     | tk::tks -> eof_add (tk::rtokens) tks in
212   eof_add [] program
213
214 (** Run the entire pipeline *)
215 let convert program =
216   (* Get rid of the end of file *)
217   let noeof = drop_eof program in
218   (* Indent in response to blocks *)
219   let indented = indenting_space noeof in
220   (* Collapse whitespace around braces *)
221   let despaced = despace_brace indented in
222   (* Get rid of trailing whitespace *)
223   let trimmed = trim_lines despaced in
224   (* Remove consecutive newlines *)
225   let squeezed = squeeze_lines trimmed in
226   (* Turn tokens into semantics *)
227   let lines = tokens_to_lines squeezed in
228   (* Consolidate those semantics *)
229   let merged = merge_lines lines in
230   (* Turn the semantics into blocks *)
231   let blocks = block_merge merged in
232   (* Put in the semicolons *)
233   let terminated = terminate_blocks blocks in
234   (* Turn the blocks into braces *)
235   let converted = space_to_brace terminated in
236   (* Put the eof on *)
237   append_eof converted
238
239 (** A function to act like a lexfun *)
240 let lextoks toks =
241   let tokens = ref (convert toks) in
242   function _ ->
243     match !tokens with
244     | [] -> raise(Failure("Not even EOF given.))
245     | tk::tks -> tokens := tks; tk

```

Source 50: "WhiteSpace.ml"

```

1 open Cast
2 open StringModules
3
4 let c_indent = "  "
5
6 let dispatches = ref []
7 let dispatchon = ref []
8 let dispatcharr = ref []
9
10 let matches type1 type2 = String.trim (GenCast.get_tname type1) = String.trim type2
11
12 let lit_to_str lit = match lit with
13 | Ast.Int(i) -> "LIT.INT("^(string_of_int i)^")"
14 | Ast.Float(f) -> "LIT.FLOAT("^(string_of_float f)^")"
15 | Ast.String(s) -> "LIT.STRING(\"" ^ s ^ "\")" (* escapes were escaped during lexing *)
16 | Ast.Bool(b) -> if b then "LIT.BOOL(1)" else "LIT.BOOL(0)"

```

```

17
18 let stringify_unop op rop rtype =
19   let (is_int, is_flt, is_bool) = (matches "Integer", matches "Float", matches "Boolean") in
20   let is_type = (is_int rtype, is_flt rtype, is_bool rtype) in
21   let type_capital = match is_type with
22     | (true, -, -) -> "INTEGER"
23     | (-, true, -) -> "FLOAT"
24     | (-, -, true) -> "BOOLEAN"
25     | (-, -, -) -> raise (Failure "Incompatible type with unop") in
26   match op with
27   | Ast.Arithmetic(Ast.Neg) -> "NEG_" ^ type_capital ^ (" ^rop^ ")
28   | Ast.CombTest(Ast.Not) -> "NOT_" ^ type_capital ^ (" ^rop^ ")
29   | _ -> raise (Failure "Unknown operator")
30
31 let stringify_arith op suffix =
32   match op with
33   | Ast.Add -> "ADD_" ^ suffix
34   | Ast.Sub -> "SUB_" ^ suffix
35   | Ast.Prod -> "PROD_" ^ suffix
36   | Ast.Div -> "DIV_" ^ suffix
37   | Ast.Mod -> "MOD_" ^ suffix
38   | Ast.Neg -> raise (Failure "Unary operator")
39   | Ast.Pow -> "POW_" ^ suffix
40   (* | Ast.Pow -> Format.sprintf "pow(%s,%s)" lop rop *)
41
42 let stringify_numtest op suffix = match op with
43   | Ast.Eq -> "NTEST_EQ_" ^ suffix
44   | Ast.Neq -> "NTEST_NEQ_" ^ suffix
45   | Ast.Less -> "NTEST_LESS_" ^ suffix
46   | Ast.Grtr -> "NTEST_GRTR_" ^ suffix
47   | Ast.Leq -> "NTEST_LEQ_" ^ suffix
48   | Ast.Geq -> "NTEST_GEQ_" ^ suffix
49
50 let stringify_combtest op suffix = match op with
51   | Ast.And -> "CTEST_AND_" ^ suffix
52   | Ast.Or -> "CTEST_OR_" ^ suffix
53   | Ast.Nand -> "CTEST_NAND_" ^ suffix
54   | Ast.Nor -> "CTEST_NOR_" ^ suffix
55   | Ast.Xor -> "CTEST_XOR_" ^ suffix
56   | Ast.Not -> raise (Failure "Unary operator")
57
58 let stringify_binop op lop rop types =
59   let (is_int, is_flt, is_bool) = (matches "Integer", matches "Float", matches "Boolean") in
60   let is_type = (is_int (fst types), is_flt (fst types), is_bool (fst types), is_int (snd types), is_flt (snd types), is_bool (snd types)) in
61   let prefix = match is_type with
62     | (true, -, -, true, -, -) -> "INT_INT"
63     | (-, true, -, -, true, -) -> "FLOAT_FLOAT"
64     | (true, -, -, -, true, -) -> "INT_FLOAT"
65     | (-, true, -, true, -, -) -> "FLOAT_INT"
66     | (-, -, true, -, -, true) -> "BOOL_BOOL"
67     | (-, -, -, -, -, -) -> raise (Failure (Format.sprintf "Binary operator applied to %s, %s" (fst types) (snd types))) in
68   let suffix = prefix ^ (" ^lop^ ", " ^rop^ ") in
69   match op with
70   | Ast.Arithmetic(arith) -> stringify_arith arith suffix
71   | Ast.NumTest(numtest) -> stringify_numtest numtest suffix
72   | Ast.CombTest(combtest) -> stringify_combtest combtest suffix
73
74 let stringify_list stmtlist = String.concat "\n" stmtlist
75
76 let rec expr_to_cstr (exptype, expr_detail) = exprdetail_to_cstr expr_detail
77

```

```

78 and exprdetail_to_cstr castexpr_detail =
79   let generate_deref obj index =
80     let arrtype = fst obj in
81     Format.sprintf "((struct %s*)(%s))[INTEGER_OF((%s))]" arrtype (expr_to_cstr obj)
      (expr_to_cstr index) in
82
83   let generate_field obj field =
84     let exptype = fst obj in
85     Format.sprintf "(%s)->%s.%s" (expr_to_cstr obj) (GenCast.from_tname exptype)
      field in
86
87   let generate_invocation recvr fname args =
88     let this = Format.sprintf "((struct %s*)(%s))" (fst recvr) (expr_to_cstr recvr)
      in
89     let vals = List.map expr_to_cstr args in
90     Format.sprintf "%s(%s)" fname (String.concat ", " (this::vals)) in
91
92   let generate_vreference vname = function
93     | Sast.Local -> vname
94     | Sast.Instance(klass) -> Format.sprintf "(this->%s).%s" klass vname in
95
96   let generate_allocation klass fname args =
97     let vals = List.map expr_to_cstr args in
98     let alloc = Format.sprintf "MAKENEW(%s)" klass in
99     Format.sprintf "%s(%s)" fname (String.concat ", " (alloc::vals)) in
100
101   let generate_array_alloc _ fname args =
102     let vals = List.map expr_to_cstr args in
103     Format.sprintf "%s(%s)" fname (String.concat ", " vals) in
104
105   let generate_refine args ret = function
106     | Sast.Switch(_, _, dispatch) ->
107       let vals = List.map expr_to_cstr args in
108       Format.sprintf "%s(%s)" dispatch (String.concat ", " ("this"::vals))
109     | _ -> raise(Failure("Wrong switch applied to refine — compiler error.)) in
110
111   let generate_refinable = function
112     | Sast.Test(_, _, dispatchby) -> Format.sprintf "%s(this)" dispatchby
113     | _ -> raise(Failure("Wrong switch applied to refinable — compiler error.)) in
114
115   match castexpr_detail with
116   | This -> "this" (* There is no way this is right with
      implicit object passing *)
117   | Null -> "NULL"
118   | Id(vname, varkind) -> generate_vreference vname varkind
119   | NewObj(classname, fname, args) -> generate_allocation classname fname args
120   | NewArr(arrtype, fname, args) -> generate_array_alloc arrtype fname args
121   | Literal(lit) -> lit_to_str lit
122   | Assign((vtype, _) as memory, data) -> Format.sprintf "%s = ((struct %s*)(%s))" (
      expr_to_cstr memory) vtype (expr_to_cstr data)
123   | Deref(carray, index) -> generate_deref carray index
124   | Field(obj, fieldname) -> generate_field obj fieldname
125   | Invoc(recvr, fname, args) -> generate_invocation recvr fname args
126   | Unop(op, expr) -> stringify_unop op (expr_to_cstr expr) (fst
      expr)
127   | Binop(lop, op, rop) -> stringify_binop op (expr_to_cstr lop) (
      expr_to_cstr rop) ((fst lop), (fst rop))
128   | Refine(args, ret, switch) -> generate_refine args ret switch
129   | Refinable(switch) -> generate_refinable switch
130
131 and vdecl_to_cstr (vtype, vname) = Format.sprintf "struct %s%s" vtype vname
132
133
134 let rec collect_dispatches_exprs exprs = List.iter collect_dispatches_expr exprs
135 and collect_dispatches_stmts stmts = List.iter collect_dispatches_stmt stmts

```

```

136 and collect_dispatches_expr (_, detail) = match detail with
137 | This -> ()
138 | Null -> ()
139 | Id(_,_) -> ()
140 | NewObj(_,_, args) -> collect_dispatches_exprs args
141 | NewArr(arrtype, fname, args) -> collect_dispatch_arr arrtype fname args
142 | Literal(_) -> ()
143 | Assign(mem, data) -> collect_dispatches_exprs [mem; data]
144 | Deref(arr, idx) -> collect_dispatches_exprs [arr; idx]
145 | Field(obj, _) -> collect_dispatches_expr obj
146 | Invoc(recvr, _, args) -> collect_dispatches_exprs (recvr::args)
147 | Unop(_, expr) -> collect_dispatches_expr expr
148 | Binop(l, _, r) -> collect_dispatches_exprs [l; r]
149 | Refine(args, ret, switch) -> collect_dispatch args ret switch
150 | Refinable(switch) -> collect_dispatch_on switch
151 and collect_dispatches_stmt = function
152 | Decl(_, Some(expr), _) -> collect_dispatches_expr expr
153 | Decl(_, None, _) -> ()
154 | If(iflist, env) -> collect_dispatches_clauses iflist
155 | While(pred, body, _) -> collect_dispatches_expr pred; collect_dispatches_stmts body
156 | Expr(expr, _) -> collect_dispatches_expr expr
157 | Return(Some(expr), _) -> collect_dispatches_expr expr
158 | Super(_, _, args) -> collect_dispatches_exprs args
159 | Return(None, _) -> ()
160 and collect_dispatches_clauses pieces =
161 let (preds, bodies) = List.split pieces in
162 collect_dispatches_exprs (Util.filter_option preds);
163 collect_dispatches_stmts (List.flatten bodies)
164 and collect_dispatch_args ret = function
165 | Sast.Switch(klass, cases, dispatch) -> dispatches := (klass, ret, (List.map fst
166 args), dispatch, cases)::(!dispatches);
167 | Sast.Test(_, _, _) -> raise(Failure("Impossible (wrong switch — compiler error)"))
168 and collect_dispatch_on = function
169 | Sast.Test(klass, classes, dispatchby) -> dispatchon := (klass, classes, dispatchby)
170 ::(!dispatchon);
171 | Sast.Switch(_, _, _) -> raise(Failure("Impossible (wrong switch — compiler error)"))
172 and collect_dispatch_func func = collect_dispatches_stmts func.body
173 and collect_dispatch_arr arrtype fname args =
174 dispatcharr := (arrtype, fname, args)::(!dispatcharr)
175
176 (**
177 Takes an element from the dispatchon list and generates the test function for
178 refinable.
179 @param classes — list of classes in which the refinable method is defined for the
180 method
181 fuid — unique function name for the test function.
182 @return true or false
183 Checks if the object on which refinable was invoked has an associated refinable
184 method
185 dispatched via this function that's being generated in one of the classes.
186 **)
187
188 let generate_testsw (klass, classes, fuid) =
189 let test_klass = Format.sprintf "\tif ( IS_CLASS(this, \"%s\") ) return LIT_BOOL(1);"
190 (String.trim klass) in
191 let cases = String.concat "\n" (List.map test_classes) in
192 let body = Format.sprintf "%s\n\treturn LIT_BOOL(0);" cases in
193 Format.sprintf "struct t_Boolean *%s( struct %s*this )\n{\n%s\n}\n\n" fuid klass body
194
195 (**
196 Takes a dispatch element of the global dispatches list
197 And generates the dispatch function — dispatcher which dispatches
198 calls to refinable methods based on the RTTI of the this.

```

```

194     @param ret - return type of the function
195     args - arguments to the dispatcher and the dispatched method
196     dispatch uid - unique function name for the dispatcher
197     cases - list of classes and their corresponding uid of the invokable
198     refinable methods.
199
200 let generate_refinesw (klass, ret, args, dispatchuid, cases) =
201   let rettype = match ret with
202     | None -> "void "
203     | Some(atype) -> Format.sprintf "struct %s*" atype in
204   let this = (Format.sprintf "struct %s*" klass, "this") in
205   let formals = List.mapi (fun i t -> (Format.sprintf "struct %s*" t, Format.sprintf "
206     varg-%d" i)) args in
207   let signature = String.concat ", " (List.map (fun (t, v) -> t ^ v) (this::formals))
208   in
209   let actuals = List.map snd formals in
210   let withthis kname = String.concat ", " ((Format.sprintf "(struct %s*) this" kname)::
211     actuals) in
212   let invoc fuid kname = Format.sprintf "%s(%s)" fuid (withthis kname) in
213   let execute fuid kname = match ret with
214     | None -> Format.sprintf "%s; return;" (invoc fuid kname)
215     | Some(atype) -> Format.sprintf "return ((struct %s*)(%s));" (String.trim atype)
216   in (invoc fuid kname) in
217   let unroll_case (kname, fuid) =
218     Format.sprintf "\tif( IS_CLASS( this, \"%s\" ) )\n\t\t{ %s }\n" (String.trim kname)
219   ) (execute fuid kname) in
220   let generated = List.map unroll_case cases in
221   let fail = Format.sprintf "REFINE_FAIL(\"%s\")" (String.trim klass) in
222   Format.sprintf "%s%s(%s)\n{\n%s\n\t%s\n}\n\n" rettype dispatchuid signature (String.
223     concat "" generated) fail
224
225 let generate_arrayalloc (arrtype, fname, args) =
226   let params = List.mapi (fun i _ -> Format.sprintf "struct %s*v_dim%d" (GenCast.
227     get_tname "Integer") i) args in
228   match List.length params with
229   | 1 -> Format.sprintf "struct %s%s(%s) {\n\treturn ONEDIM_ALLOC(struct %s,
230     INTEGER_OF(v_dim0));\n}\n" arrtype fname (String.concat ", " params) arrtype
231   | _ -> raise(Failure("Only one dimensional arrays currently supported.))
232
233 (**
234   Take a list of cast_stmts and return a body of c statements
235   @param stmtlist A list of statements
236   @return A body of c statements
237 *)
238 let rec cast_to_c_stmt indent cast =
239   let indents = String.make indent '\t' in
240   let stmts = cast_to_c_stmtlist (indent+1) in
241
242   let cstmt = match cast with
243   | Decl((vtype, _) as vdecl, Some(expr), env) -> Format.sprintf "%s = ((struct %s
244     *)(%s));" (vdecl_to_cstr vdecl) vtype (expr_to_cstr expr)
245   | Decl(vdecl, None, env) -> Format.sprintf "%s;" (vdecl_to_cstr vdecl)
246   | If(iflist, env) -> cast_to_c_if_chain indent iflist
247   | While(pred, [], env) -> Format.sprintf "while ( BOOL_OF( %s ) ) { }" (
248     expr_to_cstr pred)
249   | While(pred, body, env) -> Format.sprintf "while ( BOOL_OF( %s ) ) {\n%s\n%s}" (
250     expr_to_cstr pred) (stmts body) indents
251   | Expr(expr, env) -> Format.sprintf "( %s );" (expr_to_cstr expr)
252   | Return(Some(expr), env) -> Format.sprintf "return ( %s );" (expr_to_cstr expr)
253   | Return(_, env) -> "return;"
254   | Super(klass, fuid, []) -> Format.sprintf "%s((struct %s*)(this));" fuid (
255     GenCast.get_tname klass)
256   | Super(klass, fuid, args) -> Format.sprintf "%s((struct %s*)(this), %s);" fuid (
257     GenCast.get_tname klass) (String.concat ", " (List.map expr_to_cstr args)) in

```

```

245     indents ^ cstmt
246
247 and cast_to_c_stmtlist indent stmts =
248   String.concat "\n" (List.map (cast_to_c_stmt indent) stmts)
249
250 and cast_to_c_if_pred = function
251   | None -> ""
252   | Some(ifpred) -> Format.sprintf "if ( BOOLOF( %s ) )" (expr_to_cstr ifpred)
253
254 and cast_to_c_if_chain indent pieces =
255   let indents = String.make indent '\t' in
256   let stmts = cast_to_c_stmtlist (indent + 1) in
257   let combine (pred, body) = Format.sprintf "%s {\n%s\n%s}" (cast_to_c_if_pred pred) (
258     stmts body) indents in
259   String.concat " else " (List.map combine pieces)
260
261 let cast_to_c_class_struct class_name ancestors =
262   let ancestor_var (vtype, vname) = Format.sprintf "struct %s*%s;" vtype vname in
263   let ancestor_vars vars = String.concat "\n\t\t" (List.map ancestor_var vars) in
264   let internal_struct (ancestor, vars) = match vars with
265     | [] -> Format.sprintf "struct { BYTE empty_vars; } %s;" ancestor
266     | _ -> Format.sprintf "struct {\n\t\t\t%s\n\t\t} %s;\n" (ancestor_vars vars) ancestor
267   in
268   let internals = String.concat "\n\n\t" (List.map internal_struct ancestors) in
269   let meta = "\tClassInfo *meta;" in
270   Format.sprintf "struct %s {\n\t%s\n\t\t%s\n};\n\n" (String.trim class_name) meta
271   internals
272
273 let cast_to_c_func cfunc =
274   let ret_type = match cfunc.returns with
275     | None -> "void "
276     | Some(atype) -> Format.sprintf "struct %s*" atype in
277   let body = match cfunc.body with
278     | [] -> " { }"
279     | body -> Format.sprintf "\n{\n\t%s\n}" (cast_to_c_stmtlist 1 body) in
280   let params = if cfunc.static = false then (GenCast.get_tname cfunc.inclass, "this")::
281     cfunc.formals
282     else cfunc.formals in
283   let signature = String.concat ", " (List.map (fun (t,v) -> "struct " ^ t ^ "*" ^ v)
284     params) in
285   if cfunc.builtin then Format.sprintf "/* Place-holder for %s%s(%s) */" ret_type cfunc
286     .name signature
287   else Format.sprintf "\n%s%s(%s)%s\n" ret_type cfunc.name signature body
288
289 let cast_to_c_proto cfunc =
290   let ret_type = match cfunc.returns with
291     | None -> "void "
292     | Some(atype) -> Format.sprintf "struct %s*" atype in
293   let first = if cfunc.static then [] else [(GenCast.get_tname cfunc.inclass, "this")]
294   in
295   let params = first@cfunc.formals in
296   let types = String.concat ", " (List.map (fun (t,v) -> "struct " ^ t ^ "*" ^ v) params)
297   in
298   let signature = Format.sprintf "%s%s(%s);" ret_type cfunc.name types in
299   if cfunc.builtin then Format.sprintf "" else signature
300
301 let cast_to_c_proto_dispatch_arr (arrtype, fname, args) =
302   let int = Format.sprintf "struct %s*" (GenCast.get_tname "Integer") in
303   let params = List.map (fun _ -> int) args in
304   Format.sprintf "struct %s%s(%s);" arrtype fname (String.concat ", " params)
305
306 let cast_to_c_proto_dispatch_on (klass, _, uid) =
307   Format.sprintf "struct t_Boolean *%s(struct %s *);" uid klass

```



```

302 let cast_to_c_proto_dispatch (klass, ret, args, uid, _) =
303   let types = List.map (fun t -> "struct " ^ t ^ "s") (klass::args) in
304   let proto_rtype = Format.sprintf "struct %s%s(%s);" rtype uid (String.concat ", "
types) in
305   match ret with
306   | None -> proto "void"
307   | Some(t) -> proto t
308
309 let cast_to_c_main mains =
310   let main_fmt = ""^^"\tif (!strcmp(gmain, \"%s\", %d)) { %s(&global_system, str_args)
; return 0; }" in
311   let for_main (klass, uid) = Format.sprintf main_fmt klass (String.length klass + 1)
uid in
312   let switch = String.concat "\n" (List.map for_main mains) in
313   let cases = Format.sprintf "\n%s\n" (String.concat ", " (List.map fst mains)) in
314   Format.sprintf "#define CASES %s\n\nint main(int argc, char **argv) {\n\tINIT_MAIN(
CASES)\n%s\n\tFAIL_MAIN(CASES)\n\treturn 1;\n}" cases switch
315
316 let commalines input n =
317   let newline string = String.length string >= n in
318   let rec line_builder line rlines = function
319   | [] -> List.map String.trim (List.rev (line::rlines))
320   | str::rest ->
321     let comma = match rest with [] -> false | _ -> true in
322     let str = if comma then str ^ ", " else str in
323     if newline line then line_builder str (line::rlines) rest
324     else line_builder (line ^ str) rlines rest in
325   match input with
326   | [] -> []
327   | [one] -> [one]
328   | str::rest -> line_builder (str ^ ", ") [] rest
329
330 let print_class_strings = function
331 | [] -> raise (Failure("Not even built in classes?"))
332 | classes -> commalines (List.map (fun k -> "\"" ^ k ^ "\"") classes) 75
333
334 let print_class_enums = function
335 | [] -> raise (Failure("Not even built in classes?"))
336 | first::rest ->
337   let first = first ^ " = 0" in
338   commalines (List.map String.uppercase (first::rest)) 75
339
340 let setup_meta klass =
341   Format.sprintf "ClassInfo M%s;" klass
342
343 let meta_init bindings =
344   let to_ptr klass = Format.sprintf "m_classes[%s]" (String.trim (String.uppercase (
GenCast.get_tname klass))) in
345   let init (klass, ancestors) =
346     let ancestors_strings = String.concat ", " (List.map to_ptr ancestors) in
347     Format.sprintf "class_info_init(&M%s, %d, %s);" klass (List.length ancestors)
ancestors_strings in
348   let bindings = List.filter (fun (k, _) -> not (StringSet.mem (GenCast.get_tname k)
GenCast.built_in_names)) bindings in
349   let inits = List.map init bindings in
350   let inits = List.map (Format.sprintf "\t%s") inits in
351   let built_in_init = "\tinit_built_in_infos();" in
352   Format.sprintf "void init_class_infos() {\n%s\n}\n" (String.concat "\n" (
built_in_init::inits))
353
354 let cast_to_c ((cdefs, funcs, mains, ancestry) : Cast.program) channel =
355   let out_string = Printf.fprintf channel "%s\n" string in
356   let noblanks = function
357   | "" -> ()
358   | string -> Printf.fprintf channel "%s\n" string in

```

```

359 let incl file = out (Format.sprintf "#include \"%s.h\"\n" file) in
360
361 let comment string =
362   let comments = Str.split (Str.regexp "\n") string in
363   let commented = List.map (Format.sprintf " * %s") comments in
364   out (Format.sprintf "\n\n/*\n%s\n */" (String.concat "\n" commented)) in
365
366 let func_compare f g =
367   let strcmp = Pervasives.compare f.name g.name in
368   if f.builtin = g.builtin then strcmp else if f.builtin then -1 else 1 in
369 let funcs = List.sort func_compare funcs in
370
371 comment "Passing over code to find dispatch data.";
372 List.iter collect_dispatch_func funcs;
373
374 comment "Gamma preamble — macros and such needed by various things";
375 incl "gamma-preamble";
376
377 comment "Ancestry meta-info to link to later.";
378 let classes = List.map (fun (kls, _) -> String.trim (GenCast.get_tname kls)) (
  StringMap.bindings ancestry) in
379 let class_strs = List.map (Format.sprintf "\t%s") (print_class_strings classes) in
380 out (Format.sprintf "char *m_classes[] = {\n%s\n};" (String.concat "\n" class_strs));
381
382 comment "Enums used to reference into ancestry meta-info strings.";
383 let class_enums = List.map (Format.sprintf "\t%s") (print_class_enums classes) in
384 out (Format.sprintf "enum m_class_idx {\n%s\n};" (String.concat "\n" class_enums));
385
386 comment "Header file containing meta information for built in classes.";
387 incl "gamma-builtin-meta";
388
389 comment "Meta structures for each class.";
390 let print_meta (klass, ancestors) =
391   if StringSet.mem (GenCast.get_tname klass) GenCast.built_in_names then ()
392   else out (setup_meta klass) in
393 List.iter print_meta (StringMap.bindings ancestry);
394 out "";
395 out (meta_init (StringMap.bindings ancestry));
396
397 comment "Header file containing structure information for built in classes.";
398 incl "gamma-builtin-struct";
399
400 comment "Structures for each of the objects.";
401 let print_class klass data =
402   if StringSet.mem klass GenCast.built_in_names then ()
403   else out (cast_to_c_class_struct klass data) in
404 StringMap.iter print_class cdefs;
405
406 comment "Header file containing information regarding built in functions.";
407 incl "gamma-builtin-functions";
408
409 comment "All of the function prototypes we need to do magic.";
410 List.iter (fun func -> noblanks (cast_to_c_proto func)) funcs;
411
412 comment "All the dispatching functions we need to continue the magic.";
413 List.iter (fun d -> out (cast_to_c_proto_dispatch_on d)) (!dispatchon);
414 List.iter (fun d -> out (cast_to_c_proto_dispatch d)) (!dispatches);
415
416 comment "Array allocators also do magic.";
417 List.iter (fun d -> out (cast_to_c_proto_dispatch_arr d)) (!dispatcharr);
418
419 comment "All of the functions we need to run the program.";
420 List.iter (fun func -> out (cast_to_c_func func)) funcs;
421
422 comment "Dispatch looks like this.";

```

```

423 List.iter (fun d -> out (generate_testsw d)) (!dispatchon);
424 List.iter (fun d -> out (generate_refinesw d)) (!dispatches);
425
426 comment "Array allocators.";
427 List.iter (fun d -> out (generate_arrayalloc d)) (!dispatcharr);
428
429 comment "The main.";
430 out (cast_to_c_main mains);

```

Source 51: "GenC.ml"

```

1  open Ast
2  open Variables
3  open StringModules
4
5  let rec get_vars_formals = function
6    | [] -> StringSet.empty
7    | [(-, var)] -> StringSet.singleton var
8    | (-, var)::tl -> StringSet.add var (get_vars_formals tl)
9
10 let _ =
11   let func = List.hd (Debug.get_example_longest_body "Multi" "Collection") in
12   let stmts = func.body in
13   let prebound = get_vars_formals func.formals in
14   let free_variables = free_vars prebound stmts in
15   StringSet.iter (Printf.printf "%s\n") free_variables

```

Source 52: "freevars.ml"

```

1  let debug_print tokens =
2    let ptoken header tokens =
3      Inspector.pprint_token_list header tokens;
4      print_newline () in
5    let plines header lines =
6      Inspector.pprint_token_lines header lines;
7      print_newline () in
8    begin
9      ptoken "Input:      " tokens;
10     let tokens = WhiteSpace.drop_eof tokens in
11     ptoken "No EOF      " tokens;
12     let tokens = WhiteSpace.indenting_space tokens in
13     ptoken "Indented:   " tokens;
14     let tokens = WhiteSpace.despace_brace tokens in
15     ptoken "In-Brace:  " tokens;
16     let tokens = WhiteSpace.trim_lines tokens in
17     ptoken "Trimmed:    " tokens;
18     let tokens = WhiteSpace.squeeze_lines tokens in
19     ptoken "Squeezed:   " tokens;
20     let lines = WhiteSpace.tokens_to_lines tokens in
21     plines "Lines:      " lines;
22     let lines = WhiteSpace.merge_lines lines in
23     plines "Merged:    " lines;
24     let lines = WhiteSpace.block_merge lines in
25     plines "Blocks:     " lines;
26     let tokens = WhiteSpace.space_to_brace lines in
27     ptoken "Converted:  " tokens;
28     let tokens = WhiteSpace.append_eof tokens in
29     ptoken "With EOF:   " tokens;
30   end
31
32 let _ =

```

```

33 let tokens = Inspector.from_channel stdin in
34 match Array.length Sys.argv with
35 | 1 -> Inspector.pprint_token_list "" (WhiteSpace.convert tokens)
36 | _ -> debug_print tokens

```

Source 53: "streams.ml"

```

1 val built_in_classes : Ast.class_def list
2 val is_built_in : string -> bool

```

Source 54: "BuiltIns.mli"

```

1 open Parser
2
3 let descan = Inspector.descan
4
5 let rec indenter depth indent =
6   for i = 1 to depth do print_string indent done
7
8   (* Unscan a sequence of tokens. Requires sanitized stream *)
9   let rec clean_unscan depth indent = function
10     (* ARRAY / LBRACKET RBRACKET ambiguity... *)
11     | LBRACKET::RBRACKET::rest ->
12       print_string ((descan LBRACKET) ^ " " ^ (descan RBRACKET));
13       clean_unscan depth indent rest
14     | LBRACE::rest ->
15       print_string (descan LBRACE);
16       print_newline ();
17       indenter (depth+1) indent;
18       clean_unscan (depth+1) indent rest
19     | SEMI::RBRACE::rest ->
20       print_string (descan SEMI);
21       clean_unscan depth indent (RBRACE::rest)
22     | RBRACE::RBRACE::rest ->
23       print_newline ();
24       indenter (max (depth-1) 0) indent;
25       print_string (descan RBRACE);
26       clean_unscan (max (depth-1) 0) indent (RBRACE::rest)
27     | RBRACE::rest ->
28       print_newline ();
29       indenter (depth-1) indent;
30       print_string (descan RBRACE);
31       print_newline ();
32       indenter (depth-1) indent;
33       clean_unscan (max (depth-1) 0) indent rest
34     | SEMI::rest ->
35       print_string (descan SEMI);
36       print_newline ();
37       indenter depth indent;
38       clean_unscan depth indent rest
39     | EOF::[] ->
40       print_newline ()
41     | EOF::_ ->
42       raise (Failure("Premature end of file."))
43     | token::rest ->
44       print_string (descan token);
45       print_string " ";
46       clean_unscan depth indent rest
47     | [] ->
48       print_newline ()
49

```

```

50 let _ =
51   let tokens = Inspector.from_channel stdin in
52   clean_unscan 0 " " (WhiteSpace.convert tokens)

```

Source 55: "canonical.ml"

```

1  open Ast
2  open StringModules
3
4  (** Module to contain global class hierarchy type declarations *)
5
6  (** A full class record table as a type *)
7  type class_data = {
8    known : StringSet.t; (** Set of known class names *)
9    classes : class_def lookup_map; (** class name -> class def map *)
10   parents : string lookup_map; (** class name -> parent name map *)
11   children : (string list) lookup_map; (** class name -> children list map *)
12   variables : (class_section * string) lookup_table; (** class name -> var name -> (
13     section, type) map *)
14   methods : (func_def list) lookup_table; (** class name -> method name -> func_def
15     list map *)
16   refines : (func_def list) lookup_table; (** class name -> host.refinement -> func_def
17     list map *)
18   mains : func_def lookup_map; (** class name -> main map *)
19   ancestors : (string list) lookup_map; (** class name -> ancestor list (given to
20     Object) *)
21   distance : int lookup_table; (** subtype -> supertype -> # hops map *)
22   refinable : (func_def list) lookup_table (** class -> host -> refinements (in
23     subclasses) *)
24 }
25
26 (**
27  All the different types of non-compiler errors that can occur (programmer errors)
28 *)
29 type class_data_error
30 = HierarchyIssue of string
31 | DuplicateClasses of string list
32 | DuplicateVariables of (string * string list) list
33 | DuplicateFields of (string * (string * string) list) list
34 | UnknownTypes of (string * (string * string) list) list
35 | ConflictingMethods of (string * (string * string list) list) list
36 | ConflictingInherited of (string * (string * string list) list) list
37 | PoorlyTypedSigs of (string * (string * string option * (string * string) list) list)
38   list
39 | Uninstantiable of string list
40 | ConflictingRefinements of (string * (string * string list) list) list
41 | MultipleMains of string list

```

Source 56: "GlobalData.mli"

```

1  {
2    open Parser
3
4    (** The general lexographic scanner for Gamma *)
5
6    (**
7     Build a string from a list of characters
8     from: http://caml.inria.fr/mantis/view.php?id=5367
9     @param l The list to be glued
10    @return A string of the characters in the list glued together
11    *)

```

```

12 let implode l =
13   let res = String.create (List.length l) in
14   let rec imp i = function
15     | [] -> res
16     | c :: l -> res.[i] <- c; imp (i + 1) l in
17   imp 0 l
18
19 (**
20   Explode a string into a list of characters
21   @param s The string to be exploded
22   @return A list of the characters in the string in order
23 *)
24 let explode s =
25   let rec exploder idx l =
26     if idx < 0
27     then l
28     else exploder (idx-1) (s.[idx] :: l) in
29   exploder (String.length s - 1) []
30
31 (**
32   A generic function to count the character-spaces of a character. (I.e. weight tabs
33   more heavily)
34 *)
35 let spacecounter = function
36   | '\t' -> 8
37   | _ -> 1
38
39 (**
40   Count the space width of a string using the spacecounter function
41   @param s The string to be evaluated
42   @return The effective width of the string when rendered
43 *)
44 let spacecount s =
45   let spaces = List.map spacecounter (explode s) in
46   List.fold_left (+) 0 spaces
47
48 (**/**)
49 let line_number = ref 1
50 (**/**)
51
52 (**
53   Count the lines in a series of vertical spacing characters.
54   Please note that as of now, it is not intelligent enough to understand
55   that \n\r should be counted as one. It seems like an oversized-amount
56   of work for something we will never effectively need.
57   @param v The vertical spacing series string
58 *)
59 let count_lines v = (line_number := !line_number + String.length v)
60
61 (**
62   Gracefully tell the programmer that they done goofed
63   @param msg The descriptive error message to convey to the programmer
64 *)
65 let lexfail msg =
66   raise (Failure("Line " ^ string_of_int !line_number ^ ": " ^ msg))
67 }
68
69 let digit = ['0'-'9']
70 let lower = ['a'-'z']
71 let upper = ['A'-'Z']
72 let alpha = lower | upper
73 let ualphanum = '_' | alpha | digit
74
75 (* horizontal spacing: space & tab *)
76 let hspace = [' ' '\t']

```

```

76
77 (* vertical spaces: newline (line feed), carriage return, vertical tab, form feed *)
78 let vspace = ['\n' '\r' '\011' '\012']
79
80
81 rule token = parse
82   (* Handling whitespace mode *)
83   | hspace+ as s      { SPACE(spacecount s) }
84   | ':' hspace* (vspace+ as v) { count_lines v; COLON }
85   | vspace+ as v      { count_lines v; NEWLINE }
86
87   (* Comments *)
88   | "/*"              { comment 0 lexbuf }
89
90   (* Boolean Tests & Values *)
91   | "refinable"        { REFINABLE }
92   | "and"              { AND }
93   | "or"              { OR }
94   | "xor"             { XOR }
95   | "nand"            { NAND }
96   | "nor"             { NOR }
97   | "not"            { NOT }
98   | "true"           { BLIT(true) }
99   | "false"          { BLIT(false) }
100  | "="              { EQ }
101  | "<"             { NEQ }
102  | "=/="           { NEQ }
103  | "<"            { LT }
104  | "<="          { LEQ }
105  | ">"            { GT }
106  | ">="          { GEQ }
107
108  (* Grouping [args, arrays, code, etc] *)
109  | "["              { ARRAY }
110  | '['             { LBRACKET }
111  | ']'            { RBRACKET }
112  | '('            { LPAREN }
113  | ')'            { RPAREN }
114  | '{'           { LBRACE }
115  | '}'           { RBRACE }
116
117  (* Punctuation for the syntax *)
118  | ';'           { SEMI }
119  | ','           { COMMA }
120
121  (* Arithmetic operations *)
122  | '+'          { PLUS }
123  | '-'          { MINUS }
124  | '*'          { TIMES }
125  | '/'          { DIVIDE }
126  | '%'          { MOD }
127  | '^'          { POWER }
128
129  (* Arithmetic assignment *)
130  | "+="         { PLUSA }
131  | "-="         { MINUSA }
132  | "*="         { TIMESA }
133  | "/="         { DIVIDEA }
134  | "%="         { MODA }
135  | "^="         { POWERA }
136
137  (* Control flow *)
138  | "if"         { IF }
139  | "else"       { ELSE }
140  | "elsif"      { ELSIF }

```

```

141 | "while"                { WHILE }
142 | "return"              { RETURN }
143
144 (* OOP Stuff *)
145 | "class"              { CLASS }
146 | "extends"           { EXTEND }
147 | "super"             { SUPER }
148 | "init"              { INIT }
149
150 (* Pre defined types / values *)
151 | "null"              { NULL }
152 | "void"             { VOID }
153 | "this"             { THIS }
154
155 (* Refinement / specialization related *)
156 | "refine"            { REFINE }
157 | "refinement"        { REFINES }
158 | "to"               { TO }
159
160 (* Access *)
161 | "private"           { PRIVATE }
162 | "public"            { PUBLIC }
163 | "protected"        { PROTECTED }
164
165 (* Miscellaneous *)
166 | '.'                { DOT }
167 | "main"             { MAIN }
168 | "new"              { NEW }
169 | "=="              { ASSIGN }
170
171 (* Variable and Type IDs *)
172 | '_'? lower ualphanum* as vid { ID(vid) }
173 | upper ualphanum* as tid { TYPE(tid) }
174
175 (* Literals *)
176 | digit+ as inum { ILIT(int_of_string inum) }
177 | digit+ '.' digit+ as fnum { FLIT(float_of_string fnum) }
178 | "" { stringlit [] lexbuf }
179
180 (* Some type of end, for sure *)
181 | eof { EOF }
182 | - as char { lexfail("Illegal character " ^ Char.escaped char) }
183
184 and comment level = parse
185 (* Comments can be nested *)
186 | "/*" { comment (level+1) lexbuf }
187 | "*/" { if level = 0 then token lexbuf else comment (level-1) lexbuf }
188 | eof { lexfail("File ended inside comment.") }
189 | vspace+ as v { count_lines v; comment level lexbuf }
190 | - { comment level lexbuf }
191
192 and stringlit chars = parse
193 (* Accept valid C string literals as that is what we will output directly *)
194 | '\\ ' { escapechar chars lexbuf }
195 | eof { lexfail("File ended inside string literal") }
196 | vspace as char { lexfail("Line ended inside string literal (" ^ Char.escaped char ^ "
used): " ^ implode(List.rev chars)) }
197 | "" { SLIT(implode(List.rev chars)) }
198 | - as char { stringlit (char::chars) lexbuf }
199
200 and escapechar chars = parse
201 (* Accept valid C escape sequences *)
202 | ['a' 'b' 'f' 'n' 'r' 't' 'v' '\\' ' ' '0'] as char {
203   stringlit (char :: '\\ ' :: chars) lexbuf
204 }

```



```

205 | eof      { lexfail("File ended while seeking escape character") }
206 | - as char { lexfail("Illegal escape character: \\\" ^ Char.escaped(char)) }

```

Source 57: "scanner.mll"

```

1  open Ast
2  open Sast
3  open Klass
4  open StringModules
5  open Util
6  open GlobalData
7
8  (** Module to take an AST and build the sAST out of it. *)
9
10 (**
11   Update an environment to have a variable
12   @param mode The mode the variable is in (instance, local)
13   @param vtype The type of the variable
14   @param vname The name of the variable
15   @return A function that will update an environment passed to it.
16   *)
17 let env_update mode (vtype, vname) env = match map_lookup vname env, mode with
18 | None, _ -> StringMap.add vname (vtype, mode) env
19 | Some((otype, Local)), Local -> raise(Failure("Local variable " ^ vname ^ " loaded
20 twice, once with type " ^ otype ^ " and then with type " ^ vtype ^ "."))
21 | _, Local -> StringMap.add vname (vtype, mode) env
22 | _, _ -> raise(Failure("Instance variable declared twice in ancestry chain — this
23 should have been detected earlier; compiler error.))
24 let env_updates mode = List.fold_left (fun env vdef -> env_update mode vdef env)
25 let add_ivars klass env level =
26   let sects = match level with
27   | Publics -> [Publics]
28   | Protects -> [Publics; Protects]
29   | Privates -> [Publics; Protects; Privates]
30   | _ -> raise(Failure("Inappropriate class section — access level.)) in
31   let filter (s, _) = List.mem s sects in
32   let vars = Klass.klass_to_variables klass in
33   let eligible = List.flatten (List.map snd (List.filter filter vars)) in
34   env_updates (Instance(klass.klass)) env eligible
35
36 (** Marker for being in the current class — ADT next time *)
37 let current_class = "_CurrentClassMarker_"
38
39 (** Marker for the null type — ADT next time *)
40 let null_class = "_Null_"
41
42 (** Empty environment *)
43 let empty_environment = StringMap.empty
44
45 (** Return whether an expression is a valid lvalue or not *)
46 let is_lvalue (expr : Ast.expr) = match expr with
47 | Ast.Id(_) -> true
48 | Ast.Field(_, _) -> true
49 | Ast.Deref(_, _) -> true
50 | _ -> false
51
52 (**
53   Map a literal value to its type
54   @param litparam a literal
55   @return A string representing the type.
56   *)
57 let getLiteralType litparam = match litparam with
58 | Ast.Int(i) -> "Integer"

```

```

57 | Ast.Float(f) -> "Float"
58 | Ast.String(s) -> "String"
59 | Ast.Bool(b) -> "Boolean"
60
61 (**
62  Map a return type string option to a return type string
63  @param ret_type The return type.
64  @return The return type — Void or its listed type.
65  *)
66 let getRetType ret_type = match ret_type with
67 | Some(retval) -> retval
68 | None -> "Void"
69
70 (**
71  Update a refinement switch based on updated data.
72  *)
73 let rec update_refinements_stmts klass_data kname mname = List.map (
74   update_refinements_stmt klass_data kname mname)
75 and update_refinements_exprs klass_data kname mname = List.map (update_refinements_expr
76   klass_data kname mname)
77 and update_refinements_expr klass_data kname mname (atype, expr) =
78   let doexp = update_refinements_expr klass_data kname mname in
79   let doexps = update_refinements_exprs klass_data kname mname in
80
81   let get_refine rname arglist desired uid =
82     let argtypes = List.map fst arglist in
83     let refines = Klass.refine_on klass_data kname mname rname argtypes desired in
84     let switch = List.map (fun (f : Ast.func_def) -> (f.inklass, f.uid)) refines in
85     (getRetType desired, Sast.Refine(rname, arglist, desired, Switch(kname, switch,
86   uid))) in
87
88   let get_refinable rname uid =
89     let refines = Klass.refinable_lookup klass_data kname mname rname in
90     let classes = List.map (fun (f : Ast.func_def) -> f.inklass) refines in
91     ("Boolean", Sast.Refinable(rname, Test(kname, classes, uid))) in
92
93   match expr with
94   | Sast.Refine(rname, args, desired, Switch(_, _, uid)) -> get_refine rname args
95     desired uid
96   | Sast.Refine(_, _, _, _) -> raise (Failure("Test in switch."))
97   | Sast.Refinable(rname, Test(_, _, uid)) -> get_refinable rname uid
98   | Sast.Refinable(_, _) -> raise (Failure("Switch in test."))
99
100  | Sast.Anonymous(_, _, _) -> raise (Failure("Anonymous detected during reswitching
101  ."))
102
103  | Sast.This -> (atype, Sast.This)
104  | Sast.Null -> (atype, Sast.Null)
105  | Sast.Id(id) -> (atype, Sast.Id(id))
106  | Sast.NewObj(klass, args, uid) -> (atype, Sast.NewObj(klass, doexps args, uid))
107  | Sast.Literal(lit) -> (atype, Sast.Literal(lit))
108  | Sast.Assign(l, r) -> (atype, Sast.Assign(doexp l, doexp r))
109  | Sast.Deref(l, r) -> (atype, Sast.Deref(doexp l, doexp r))
110  | Sast.Field(e, m) -> (atype, Sast.Field(doexp e, m))
111  | Sast.Invoc(r, m, args, uid) -> (atype, Sast.Invoc(doexp r, m, doexps args, uid)
112  )
113
114  | Sast.Unop(op, e) -> (atype, Sast.Unop(op, doexp e))
115  | Sast.Binop(l, op, r) -> (atype, Sast.Binop(doexp l, op, doexp r))
116
117 and update_refinements_stmt klass_data kname mname stmt =
118   let doexp = update_refinements_expr klass_data kname mname in
119   let doexps = update_refinements_exprs klass_data kname mname in
120   let dostmts = update_refinements_stmts klass_data kname mname in
121   let docls = update_refinements_clauses klass_data kname mname in
122
123   match stmt with

```

```

116 | Sast.Decl(-, None, -) as d -> d
117 | Sast.Decl(vdef, Some(e), env) -> Sast.Decl(vdef, Some(doexp e), env)
118 | Sast.If(pieces, env) -> Sast.If(docls pieces, env)
119 | Sast.While(pred, body, env) -> Sast.While(doexp pred, dostmts body, env)
120 | Sast.Expr(expr, env) -> Sast.Expr(doexp expr, env)
121 | Sast.Return(None, -) as r -> r
122 | Sast.Return(Some(e), env) -> Sast.Return(Some(doexp e), env)
123 | Sast.Super(args, uid, super, env) -> Sast.Super(doexps args, uid, super, env)
124 and update_refinements_clauses (klass_data : class_data) (kname : string) (mname : string
    ) (pieces : (Sast.expr option * Sast.sstmt list) list) : (Sast.expr option * Sast.
    sstmt list) list =
125   let dobody = update_refinements_stmts klass_data kname mname in
126   let dopred = update_refinements_expr klass_data kname mname in
127
128   let mapping = function
129     | (None, body) -> (None, dobody body)
130     | (Some(e), body) -> (Some(dopred e), dobody body) in
131   List.map mapping pieces
132
133 let update_refinements_func klass_data (func : Sast.func_def) =
134 { func with body = update_refinements_stmts klass_data func.inclass func.name func.
    body }
135
136 let update_refinements_member klass_data = function
137 | Sast.InitMem(i) -> Sast.InitMem(update_refinements_func klass_data i)
138 | Sast.MethodMem(m) -> Sast.MethodMem(update_refinements_func klass_data m)
139 | v -> v
140
141 let update_refinements_class klass_data (klass : Sast.class_def) =
142 let mems = List.map (update_refinements_member klass_data) in
143 let funs = List.map (update_refinements_func klass_data) in
144 let s = klass.sections in
145 let sects =
146 { publics = mems s.publics;
147   protects = mems s.protects;
148   privates = mems s.privates;
149   mains = funs s.mains;
150   refines = funs s.refines } in
151 { klass with sections = sects }
152
153 let update_refinements klass_data (classes : Sast.class_def list) =
154 List.map (update_refinements_class klass_data) classes
155
156 (**
157   Given a class_data record, a class name, an environment, and an Ast.expr expression,
158   return a Sast.expr expression.
159   @param klass_data A class_data record
160   @param kname The name of the current class
161   @param env The local environment (instance and local variables so far declared)
162   @param exp An expression to eval to a Sast.expr value
163   @return A Sast.expr expression, failing when there are issues.
164 *)
165 let rec eval klass_data kname mname isstatic env exp =
166 let eval' expr = eval klass_data kname mname isstatic env expr in
167 let eval_explist elist = List.map eval' elist in
168
169 let get_field expr mbr =
170 let (recvr_type, _) as recvr = eval' expr in
171 let this = (recvr_type = current_class) in
172 let recvr_type = if this then kname else recvr_type in
173 let field_type = match Klass.class_field_far_lookup klass_data recvr_type mbr
    this with
174 | Left((- , vtyp, -)) -> vtyp
175 | Right(true) -> raise(Failure("Field " ^ mbr ^ " is not accessible in " ^
    recvr_type ^ " from " ^ kname ^ "."))

```

```

176 | Right(false) -> raise(Failure("Unknown field " ^ mbr ^ " in the ancestry of
    | " ^ recvr_type ^ ".")) in
177 | (field_type, Sast.Field(recvr, mbr)) in
178
179 let cast_to klass (_, v) = (klass, v) in
180
181 let get_invoc expr methd elist =
182 | let (recvr_type, _) as recvr = eval' expr in
183 | let arglist = eval_exprlist elist in
184 | let this = (recvr_type = current_class) in
185 | let _ = if (this && isstatic)
186 | then raise(Failure(Format.sprintf "Cannot invoke %s on %s in %s for %s is
static." methd mname kname mname))
187 | else () in
188 | let recvr_type = if this then kname else recvr_type in
189 | let argtypes = List.map fst arglist in
190 | let mdef = match Klass.best_inherited_method klass_data recvr_type methd
argtypes this with
191 | None when this -> raise(Failure(Format.sprintf "Method %s not found
ancestrally in %s (this=%b)" methd recvr_type this))
192 | None -> raise(Failure("Method " ^ methd ^ " not found (publically) in the
ancestry of " ^ recvr_type ^ "."))
193 | Some(fdef) -> fdef in
194 | let mfid = if mdef.builtin then BuiltIn mdef.uid else FuncId mdef.uid in
195 | (getRetType mdef.returns, Sast.Invoc(cast_to (mdef.inclass) recvr, methd,
arglist, mfid)) in
196
197 let get_init class_name exprlist =
198 | let arglist = eval_exprlist exprlist in
199 | let argtypes = List.map fst arglist in
200 | let mdef = match best_method klass_data class_name "init" argtypes [Ast.Publics]
with
201 | None -> raise(Failure "Constructor not found")
202 | Some(fdef) -> fdef in
203 | let mfid = if mdef.builtin then BuiltIn mdef.uid else FuncId mdef.uid in
204 | (class_name, Sast.NewObj(class_name, arglist, mfid)) in
205
206 let get_assign e1 e2 =
207 | let (t1, t2) = (eval' e1, eval' e2) in
208 | let (type1, type2) = (fst t1, fst t2) in
209 | match is_subtype klass_data type2 type1, is_lvalue e1 with
210 | _, false -> raise(Failure "Assigning to non-lvalue")
211 | false, _ -> raise(Failure "Assigning to incompatible types")
212 | _ -> (type1, Sast.Assign(t1, t2)) in
213
214 let get_binop e1 op e2 =
215 | let isCompatible typ1 typ2 =
216 | if is_subtype klass_data typ1 typ2 then typ2
217 | else if is_subtype klass_data typ2 typ1 then typ1
218 | else raise (Failure (Format.sprintf "Binop takes incompatible types: %s %s"
typ1 typ2)) in
219 | let (t1, t2) = (eval' e1, eval' e2) in
220 | let gettype op (typ1, _) (typ2, _) = match op with
221 | Ast.Arithmetic(Neg) -> raise(Failure("Negation is not a binary operation!"))
222 | Ast.CombTest(Not) -> raise(Failure("Boolean negation is not a binary
operation!"))
223 | Ast.Arithmetic(_) -> isCompatible typ1 typ2
224 | Ast.NumTest(_)
225 | Ast.CombTest(_) -> ignore(isCompatible typ1 typ2); "Boolean" in
226 | (gettype op t1 t2, Sast.Binop(t1, op, t2)) in
227
228 let get_refine rname elist desired =
229 | let arglist = eval_exprlist elist in
230 | let argtypes = List.map fst arglist in

```

```

231     let refines = Klass.refine-on klass_data kname mname rname argtypes desired in
232     let switch = List.map (fun (f : Ast.func_def) -> (f.inklass, f.uid)) refines in
233     (getRetType desired, Sast.Refine(rname, arglist, desired, Switch(kname, switch,
234     UID.uid_counter ()))) in
235
236     let get_refinable rname =
237     let refines = Klass.refinable_lookup klass_data kname mname rname in
238     let classes = List.map (fun (f : Ast.func_def) -> f.inklass) refines in
239     ("Boolean", Sast.Refineable(rname, Test(kname, classes, UID.uid_counter ()))) in
240
241     let get_deref e1 e2 =
242     let expectArray typename = match Str.last_chars typename 2 with
243     | "[]" -> Str.first_chars typename (String.length typename - 2)
244     | _ -> raise (Failure "Not an array type") in
245     let (t1, t2) = (eval' e1, eval' e2) in
246     let getArrayType (typ1, _) (typ2, _) = match typ2 with
247     | "Integer" -> expectArray typ1
248     | _ -> raise (Failure "Dereferencing invalid") in
249     (getArrayType t1 t2, Sast.Deref(t1, t2)) in
250     let get_unop op expr = match op with
251     | Ast.Arithmetic(Neg) -> let (typ, _) as ealed = eval' expr in (typ, Sast.Unop(
252     op, ealed))
253     | Ast.CombTest(Not) -> ("Boolean", Sast.Unop(op, eval' expr))
254     | _ -> raise (Failure("Unknown binary operator " ^ Inspector.inspect_ast_op op ^ "
255     given.")) in
256
257     let lookup_type id = match map_lookup id env with
258     | None -> raise (Failure("Unknown id " ^ id ^ " in environment built around " ^
259     kname ^ ", " ^ mname ^ "."))
260     | Some((vtype, _)) -> vtype in
261
262     let get_new_arr atype args =
263     let arglist = eval_exprlist args in
264     if List.exists (fun (t, _) -> t <> "Integer") arglist
265     then raise (Failure "Size of an array dimensions does not correspond to an
266     integer.")
267     else (atype, Sast.NewObj(atype, arglist, ArrayAlloc(UID.uid_counter ()))) in
268
269     let get_new_obj atype args = try
270     let index = String.index atype '[' in
271     let dimensions = (String.length atype - index) / 2 in
272     match List.length args with
273     | n when n > dimensions -> raise (Failure("Cannot allocate array, too many
274     dimensions given."))
275     | n when n < dimensions -> raise (Failure("Cannot allocate array, too few
276     dimensions given."))
277     | 0 -> (null_class, Sast.Null)
278     | _ -> get_new_arr atype args
279     with Not_found -> get_init atype args in
280
281     match exp with
282     | Ast.This -> (current_class, Sast.This)
283     | Ast.Null -> (null_class, Sast.Null)
284     | Ast.Id(vname) -> (lookup_type vname, Sast.Id(vname))
285     | Ast.Literal(lit) -> (getLiteralType lit, Sast.Literal(lit))
286     | Ast.NewObj(s1, elist) -> get_new_obj s1 elist
287     | Ast.Field(expr, mbr) -> get_field expr mbr
288     | Ast.Invoc(expr, methd, elist) -> get_invoc expr methd elist
289     | Ast.Assign(e1, e2) -> get_assign e1 e2
290     | Ast.Binop(e1, op, e2) -> get_binop e1 op e2
291     | Ast.Refine(s1, elist, soption) -> get_refine s1 elist soption
292     | Ast.Deref(e1, e2) -> get_deref e1 e2
293     | Ast.Refinable(s1) -> get_refinable s1
294     | Ast.Unop(op, expr) -> get_unop op expr
295     | Ast.Anonymous(atype, args, body) -> (atype, Sast.Anonymous(atype, eval_exprlist

```

```

    args, body)) (* Delay evaluation *)
289
290 (**
291   Given a class_data record, the name of the current class, a list of AST statements,
292   and an initial environment, enumerate the statements and attach the environment at
293   each step to that statement, yielding Sast statements. Note that when there is an
294   issue the function will raise Failure.
295   @param klass_data A class_data record
296   @param kname The name of the class that is the current context.
297   @param stmts A list of Ast statements
298   @param initial_env An initial environment
299   @return A list of Sast statements
300 *)
301 let rec attach_bindings klass_data kname mname meth_ret isstatic stmts initial_env =
302   (* Calls that go easy on the eyes *)
303   let eval' = eval klass_data kname mname isstatic in
304   let attach' = attach_bindings klass_data kname mname meth_ret isstatic in
305   let eval_exprlist env elist = List.map (eval' env) elist in
306
307   let rec get_superinit kname arglist =
308     let parent = StringMap.find kname klass_data.parents in
309     let argtypes = List.map fst arglist in
310     match best_method klass_data parent "init" argtypes [Ast.Publics; Ast.Protects]
311   with
312     | None      -> raise (Failure "Cannot find super init")
313     | Some(fdef) -> fdef in
314
315   (* Helper function for building a predicate expression *)
316   let build_predicate pred_env exp = match eval' pred_env exp with
317     | ("Boolean", _) as ealed -> ealed
318     | _ -> raise (Failure "Predicates must be boolean") in
319
320   (* Helper function for building an optional expression *)
321   let opt_eval opt_expr opt_env = match opt_expr with
322     | None -> None
323     | Some(exp) -> Some(eval' opt_env exp) in
324
325   (* For each kind of statement, build the associated Sast statment *)
326   let build_ifstmt iflist if_env =
327     let build_block if_env (exp, slist) =
328       let exprtyp = match exp with
329         | None -> None
330         | Some exp -> Some(build_predicate if_env exp) in
331       (exprtyp, attach' slist if_env) in
332     Sast.If(List.map (build_block if_env) iflist, if_env) in
333
334   let build_whilestmt expr slist while_env =
335     let exprtyp = build_predicate while_env expr in
336     let stmts = attach' slist while_env in
337     Sast.While(exprtyp, stmts, while_env) in
338
339   let build_declstmt ((vtype, vname) as vdef) opt_expr decl_env =
340     if not (Klass.is_type klass_data vtype) then raise (Failure (Format.sprintf "%s in
341     %s.%s has unknown type %s." vname kname mname vtype))
342     else match opt_eval opt_expr decl_env with
343       | Some((atype, _)) as ealed -> if not (Klass.is_subtype klass_data atype
344       vtype)
345         then raise (Failure (Format.sprintf "%s in %s.%s is type %s but is assigned
346         a value of type %s." vname kname mname vtype atype))
347         else Sast.Decl(vdef, ealed, decl_env)
348       | None -> Sast.Decl(vdef, None, decl_env) in
349
350   let check_ret_type ret_type = match ret_type, meth_ret with
351     | None, Some(-) -> raise (Failure ("Void return from non-void function " ^ mname ^
352     " in class " ^ kname ^ "."))

```

```

348 | Some(_), None -> raise(Failure("Non-void return from void function " ^ mname ^
    " in class " ^ kname ^ "."))
349 | Some(r), Some(t) -> if not (Klass.is_subtype klass_data r t) then raise(Failure
    (Format.sprintf "Method %s in %s returns %s despite being declared returning %s"
    mname kname r t))
350 | _, _ -> () in
351
352 let build_returnstmt opt_expr ret_env =
353   let ret_val = opt_eval opt_expr ret_env in
354   let ret_type = match ret_val with Some(t, _) -> Some(t) | _ -> None in
355   check_ret_type ret_type;
356   Sast.Return(ret_val, ret_env) in
357 let build_exprstmt expr expr_env = Sast.Expr(eval' expr_env expr, expr_env) in
358 let build_superstmt expr_list super_env =
359   let arglist = eval_exprlist super_env expr_list in
360   let init = get_superinit kname arglist in
361   match map_lookup kname klass_data.parents with
362   | None -> raise(Failure("Error — getting parent for object without parent: "
    ^ kname))
363   | Some(parent) -> Sast.Super(arglist, init.uid, parent, super_env) in
364
365 (* Ast statement -> (Sast.Statement, Environment Update Option) *)
366 let updater in_env = function
367   | Ast.While(expr, slist) -> (build_whilestmt expr slist in_env, None)
368   | Ast.If(iflist) -> (build_ifstmt iflist in_env, None)
369   | Ast.Decl(vdef, opt_expr) -> (build_declstmt vdef opt_expr in_env, Some(vdef))
370   | Ast.Expr(expr) -> (build_exprstmt expr in_env, None)
371   | Ast.Return(opt_expr) -> (build_returnstmt opt_expr in_env, None)
372   | Ast.Super(exprs) -> (build_superstmt exprs in_env, None) in
373
374 (* Function to fold a statement into a growing reverse list of Sast statements *)
375 let build_env (output, acc_env) stmt =
376   let (node, update) = updater acc_env stmt in
377   let updated_env = match update with
378   | None -> acc_env
379   | Some(vdef) -> env_update Local vdef acc_env in
380   (node::output, updated_env) in
381
382 List.rev (fst(List.fold_left build_env ([], initial_env) stmts))
383
384 (**
385  Given a list of statements, return whether every execution path therein returns
386  @param stmts A bunch of Ast.stmts
387  @return true or false based on whether everything returns a value.
388 *)
389 let rec does_return_stmts (stmts : Ast.stmt list) = match stmts with
390 | [] -> false
391 | Return(None)::_ -> false
392 | Return(_)::_ -> true
393 | If(pieces)::rest -> does_return_clauses pieces || does_return_stmts rest
394 | _::rest -> does_return_stmts rest
395
396 (**
397  Given a collection of if clauses, return whether they represent a return from the
398  function.
399  @param pieces If clauses (option expr, stmt list)
400  @return whether or not it can be determined that a return is guaranteed here.
401 *)
402 and does_return_clauses pieces =
403   let (preds, bodies) = List.split pieces in
404   List.mem None preds && List.for_all does_return_stmts bodies
405
406 (**
407  Change inits so that they return this
408 *)
409 let init_returns (func : Sast.func_def) =

```

```

408   let body = if func.builtin then [] else func.body @ [Sast.Return(None,
empty_environment)] in
409   let this_val = (current_class, Sast.This) in
410   let return_this (stmt : Sast.sstmt) = match stmt with
411     | Return(None, env) -> Return(Some(this_val), env)
412     | _ -> stmt in
413   { func with
414     returns = Some(func.inclass);
415     body = List.map return_this body }
416
417   let rec update_current_ref_stmts (kname : string) (stmts : Sast.sstmt list) : Sast.sstmt
list = List.map (update_current_ref_stmt kname) stmts
418   and update_current_ref_exprs (kname : string) (exprs : Sast.expr list) = List.map (
update_current_ref_expr kname) exprs
419   and update_current_ref_stmt (kname : string) (stmt : Sast.sstmt) = match stmt with
420     | Sast.Decl(vdef, None, env) -> Sast.Decl(vdef, None, env)
421     | Sast.Decl(vdef, Some(expr), env) -> Sast.Decl(vdef, Some(update_current_ref_expr
kname expr), env)
422     | Sast.Expr(expr, env) -> Sast.Expr(update_current_ref_expr kname expr, env)
423     | Sast.If(pieces, env) -> Sast.If(update_current_ref_clauses kname pieces, env)
424     | Sast.While(expr, body, env) -> Sast.While(update_current_ref_expr kname expr,
update_current_ref_stmts kname body, env)
425     | Sast.Return(None, env) -> Sast.Return(None, env)
426     | Sast.Return(Some(expr), env) -> Sast.Return(Some(update_current_ref_expr kname expr
), env)
427     | Sast.Super(args, uid, parent, env) -> Sast.Super(update_current_ref_exprs kname
args, uid, parent, env)
428   and update_current_ref_expr (kname : string) ((atype, detail) : string * Sast.expr_detail
) : string * Sast.expr_detail =
429     let cleaned = match detail with
430       | Sast.This -> Sast.This
431       | Sast.Null -> Sast.Null
432       | Sast.Id(i) -> Sast.Id(i)
433       | Sast.NewObj(klass, args, uid) -> Sast.NewObj(klass, update_current_ref_exprs
kname args, uid)
434       | Sast.Anonymous(klass, args, refs) -> Sast.Anonymous(klass, args, refs)
435       | Sast.Literal(lit) -> Sast.Literal(lit)
436       | Sast.Assign(mem, data) -> Sast.Assign(update_current_ref_expr kname mem,
update_current_ref_expr kname data)
437       | Sast.Deref(arr, idx) -> Sast.Deref(update_current_ref_expr kname arr,
update_current_ref_expr kname idx)
438       | Sast.Field(expr, member) -> Sast.Field(update_current_ref_expr kname expr,
member)
439       | Sast.Invoc(expr, meth, args, id) -> Sast.Invoc(update_current_ref_expr kname
expr, meth, update_current_ref_exprs kname args, id)
440       | Sast.Unop(op, expr) -> Sast.Unop(op, update_current_ref_expr kname expr)
441       | Sast.Binop(l, op, r) -> Sast.Binop(update_current_ref_expr kname l, op,
update_current_ref_expr kname r)
442       | Sast.Refine(refine, args, ret, switch) -> Sast.Refine(refine,
update_current_ref_exprs kname args, ret, switch)
443       | Sast.Refinable(refine, switch) -> Sast.Refinable(refine, switch) in
444     let realtype : string = if current_class = atype then kname else atype in
445     (realtype, cleaned)
446   and update_current_ref_clauses (kname : string) pieces =
447     let (preds, bodies) = List.split pieces in
448     let preds = List.map (function None -> None | Some(expr) -> Some(
update_current_ref_expr kname expr)) preds in
449     let bodies = List.map (update_current_ref_stmts kname) bodies in
450     List.map2 (fun a b -> (a, b)) preds bodies
451
452   (**
453    Given a class_data record, an Ast.func_def, and an initial environment,
454    convert the func_def to a Sast.func_def. Can raise failure when there
455    are issues with the statements / expressions in the function.
456    @param klass_data A class_data record

```



```

457 @param func An Ast.func_def to transform
458 @param initial_env The initial environment
459 @return A Sast.func_def value
460 *)
461 let ast_func_to_sast_func klass_data (func : Ast.func_def) initial_env isinit =
462   let with_params = List.fold_left (fun env vdef -> env.update Local vdef env)
   initial_env func.formals in
463   let checked : Sast.sstmt list = attach_bindings klass_data func.inklass func.name
   func.returns func.static func.body with_params in
464   let cleaned = update_current_ref_stmts func.inklass checked in
465   let sast_func : Sast.func_def =
466     {
467       returns = func.returns;
468       host = func.host;
469       name = func.name;
470       formals = func.formals;
471       static = func.static;
472       body = cleaned;
473       section = func.section;
474       inklass = func.inklass;
475       uid = func.uid;
476       builtin = func.builtin } in
477   let isvoid = match func.returns with None -> true | _ -> false in
478   if not func.builtin && not isvoid && not (does_return_stmts func.body)
479   then raise (Failure (Format.sprintf "The function %s in %s does not return on all
   execution paths" (full_signature_string func) func.inklass))
480   else if isinit then init_returns sast_func else sast_func
481
482 (**
483  Given a class_data record, an Ast.member_def, and an initial environment,
484  convert the member into an Sast.member_def. May raise failure when there
485  are issues in the statements / expressions in the member.
486  @param klass_data A class_data record.
487  @param mem An Ast.member_def value
488  @param initial_env An environment of variables
489  @return A Sast.member_def
490 *)
491 let ast_mem_to_sast_mem klass_data (mem : Ast.member_def) initial_env =
492   let change_isinit func = ast_func_to_sast_func klass_data func initial_env isinit in
493   let transformed : Sast.member_def = match mem with
494     | Ast.VarMem(v) -> Sast.VarMem(v)
495     | Ast.MethodMem(m) -> Sast.MethodMem(change false m)
496     | Ast.InitMem(m) -> Sast.InitMem(change true m) in
497   transformed
498
499 let init_calls_super (aklass : Sast.class_def) =
500   let validate_init func_def = match func_def.builtin, func_def.body with
501     | true, _ -> true
502     | _, (Super(-, -, -, -)) :: _ -> true
503     | _, _ -> false in
504   let grab_init = function
505     | InitMem(m) -> Some(m)
506     | _ -> None in
507   let get_inits mems = Util.filter_option (List.map grab_init mems) in
508   let s = aklass.sections in
509   let inits = List.flatten (List.map get_inits [s.publics; s.protects; s.privates]) in
510   List.for_all validate_init inits
511
512 let check_main (func : Ast.func_def) = match func.formals with
513   | [("System", _); ("String[]", _)] -> func
514   | _ -> raise (Failure (Format.sprintf "Main functions can only have two arguments: A
   system (first) and an array of strings (second). — error in %s" func.inklass))
515
516 (**
517  Given a class_data object and an Ast.class_def, return a Sast.class_def
   object. May fail when there are issues in the statements / expressions.

```

```

518   @param klass_data A class_data record value
519   @param ast_class A class to transform
520   @return The transformed class.
521 *)
522 let ast_to_sast_class klass_data (ast_class : Ast.class_def) =
523   let s : Ast.class_sections_def = ast_class.sections in
524   let rec update_env env sect (klass : Ast.class_def) =
525     let env = add_ivars klass env sect in
526     match klass.class with
527     | "Object" -> env
528     | _ -> let parent = Klass.class_to_parent klass in
529           let pclass = StringMap.find parent klass_data.classes in
530           update_env env Protects pclass in
531   let env = update_env empty_environment Privates ast_class in
532
533   let mems = List.map (fun m -> ast_mem_to_sast_mem klass_data m env) in
534   let funs = List.map (fun f -> ast_func_to_sast_func klass_data f env false) in
535
536   let sections : Sast.class_sections_def =
537     {
538       publics = mems s.publics;
539       protects = mems s.protects;
540       privates = mems s.privates;
541       refines = funs s.refines;
542       mains = funs (List.map check_main s.mains) } in
543
544   let sast_class : Sast.class_def =
545     {
546       klass = ast_class.class;
547       parent = ast_class.parent;
548       sections = sections } in
549
550   if init_calls_super sast_class then sast_class
551   else raise (Failure (Format.sprintf "%s's inits don't always call super as their first
552 statement (maybe empty body, maybe something else)." sast_class.class))
553
554 (**
555   @param ast An ast program
556   @return A sast program
557 *)
558 let ast_to_sast klass_data =
559   let classes = StringMap.bindings klass_data.classes in
560   let to_sast (_, klass) = ast_to_sast_class klass_data klass in
561   List.map to_sast classes

```

Source 58: "BuildSast.ml"

```

1  (**
2   The abstract syntax tree for Gamma
3  *)
4
5  (**
6   The four literal classes of Gamma:
7   - Int - Integer
8   - Float - Floating-point number
9   - String - A sequence of characters
10  - Bool - a boolean value of either true or false
11  *)
12  type lit =
13    Int of int
14  | Float of float
15  | String of string
16  | Bool of bool
17
18  (** The binary arithmetic operators *)

```

```

19 type arith = Add | Sub | Prod | Div | Mod | Neg | Pow
20
21 (** The binary comparison operators *)
22 type numtest = Eq | Neq | Less | Grtr | Leq | Geq
23
24 (** The binary boolean operators *)
25 type combtest = And | Or | Nand | Nor | Xor | Not
26
27 (** All three sets of binary operators *)
28 type op = Arithmetic of arith | NumTest of numtest | CombTest of combtest
29
30 (** The various types of expressions we can have. *)
31 type expr =
32   | This
33   | Null
34   | Id of string
35   | NewObj of string * expr list
36   | Anonymous of string * expr list * func_def list
37   | Literal of lit
38   | Assign of expr * expr (* memory := data — whether memory is good is a semantic
39     issue *)
40   | Deref of expr * expr (* road[pavement] *)
41   | Field of expr * string (* road.pavement *)
42   | Invoc of expr * string * expr list (* receiver.method(args) *)
43   | Unop of op * expr (* !x *)
44   | Binop of expr * op * expr (* x + y *)
45   | Refine of string * expr list * string option
46   | Refinable of string (* refinable *)
47 (** The basic variable definition, a type and an id*)
48 and var_def = string * string (* Oh typing, you pain in the ass, add a int for array *)
49 (** The basic statements: Variable declarations, control statements, assignments, return
50 statements, and super class expressions *)
51 and stmt =
52   | Decl of var_def * expr option
53   | If of (expr option * stmt list) list
54   | While of expr * stmt list
55   | Expr of expr
56   | Return of expr option
57   | Super of expr list
58
59 (** Three access levels, the refinements, and the main function *)
60 and class_section = Publics | Protects | Privates | Refines | Mains
61
62 (** We have four different kinds of callable code blocks: main, init, refine, method. *)
63 and func_def = {
64   returns : string option; (* A return type (method/refine) *)
65   host : string option; (* A host class (refine) *)
66   name : string; (* The function name (all) *)
67   static : bool; (* If the function is static (main) *)
68   formals : var_def list; (* A list of all formal parameters of the function (all) *)
69   body : stmt list; (* A list of statements that form the function body (all) *)
70   section : class_section; (* A semantic tag of the class section in which the function
71     lives (all) *)
72   inclass : string; (* A semantic tag of the class in which the function lives (
73     all) *)
74   uid : string; (* A string for referencing this — should be maintained in
75     transformations to later ASTs *)
76   builtin : bool; (* Whether or not the function is built in (uid should have
77     _ in it then) *)
78 }
79
80 (** A member is either a variable or some sort of function *)
81 type member_def = VarMem of var_def | MethodMem of func_def | InitMem of func_def
82
83 (** Things that can go in a class *)

```

```

78 type class_sections_def = {
79   privates : member_def list;
80   protects : member_def list;
81   publics  : member_def list;
82   refines   : func_def list;
83   mains     : func_def list;
84 }
85
86 (* Just pop init and main in there? *)
87 (** The basic class definition *)
88 type class_def = {
89   klass      : string; (** A name string *)
90   parent     : string option; (** The parent class name *)
91   sections   : class_sections_def; (** The five sections *)
92 }
93
94 (** A program, right and proper *)
95 type program = class_def list

```

Source 59: "Ast.mli"

```

1 let _ =
2   let tokens = Inspector.from_channel stdin in
3   let classes = Parser.cdecls (WhiteSpace.lextoks tokens) (Lexing.from_string "") in
4   let pp_classes = List.map Pretty.pp_class_def classes in
5   print_string (String.concat "\n\n" pp_classes); print_newline ()

```

Source 60: "prettify.ml"

```

1 val deanonymize : GlobalData.class_data -> Sast.class_def list -> (GlobalData.class_data
  * Sast.class_def list, GlobalData.class_data_error) Util.either

```

Source 61: "Unanonymous.mli"

```

1
2 /* GLOBAL DATA */
3 struct t_System global_system;
4 int object_counter;
5 int global_argc;
6
7 /* Prototypes */
8 struct t_Object *allocate_for(size_t, ClassInfo *);
9 void *array_allocator(size_t, int);
10 struct t_Integer *integer_value(int);
11 struct t_Float *float_value(double);
12 struct t_Boolean *bool_value(unsigned char);
13 struct t_String *string_value(char *);
14 struct t_Boolean *boolean_init(struct t_Boolean *);
15 struct t_Integer *integer_init(struct t_Integer *);
16 struct t_Float *float_init(struct t_Float *);
17 struct t_Object *object_init(struct t_Object *);
18 struct t_String *string_init(struct t_String *);
19 struct t_Printer *printer_init(struct t_Printer *, struct t_Boolean *);
20 struct t_Scanner *scanner_init(struct t_Scanner *);
21 struct t_Integer *float_to_i(struct t_Float *);
22 struct t_Float *integer_to_f(struct t_Integer *);
23 struct t_Float *scanner_scan_float(struct t_Scanner *);
24 struct t_Integer *scanner_scan_integer(struct t_Scanner *);

```

```

25 struct t_String *scanner_scan_string(struct t_Scanner *);
26 void printer_print_float(struct t_Printer *, struct t_Float *);
27 void printer_print_integer(struct t_Printer *, struct t_Integer *);
28 void printer_print_string(struct t_Printer *, struct t_String *);
29 struct t_String **get_gamma_args(char **argv, int argc);
30
31
32 char *stack_overflow_getline(FILE *);
33
34 /* Functions! */
35
36 /* Magic allocator. DO NOT INVOKE THIS, USE MAKENEW(TYPE)
37  * where type is not prefixed (i.e. MAKENEW(Integer) not
38  * MAKENEW(t_Integer))
39  */
40 struct t_Object *allocate_for(size_t s, ClassInfo *meta) {
41     struct t_Object *this = (struct t_Object *) (malloc(s));
42     if (!this) {
43         fprintf(stderr, "Could not even allocate memory. Exiting.\n");
44         exit(1);
45     }
46     this->meta = meta;
47     return this;
48 }
49
50 void *array_allocator(size_t size, int n) {
51     void *mem = malloc(size * n);
52     if (!mem) {
53         fprintf(stderr, "Failure allocating for array. Exiting.\n");
54         exit(1);
55     }
56     memset(mem, 0, size * n);
57     return mem;
58 }
59
60 /* Make basic objects with the given values. */
61 struct t_Integer *integer_value(int in_i) {
62     struct t_Integer *i = MAKENEW(Integer);
63     i = integer_init(i);
64     i->Integer.value = in_i;
65     return i;
66 }
67
68 struct t_Float *float_value(double in_f) {
69     struct t_Float *f = MAKENEW(Float);
70     f = float_init(f);
71     f->Float.value = in_f;
72     return f;
73 }
74
75 struct t_Boolean *bool_value(unsigned char in_b) {
76     struct t_Boolean *b = MAKENEW(Boolean);
77     b = boolean_init(b);
78     b->Boolean.value = in_b;
79     return b;
80 }
81
82 struct t_String *string_value(char *s_in) {
83     size_t length = 0;
84     char *dup = NULL;
85     length = strlen(s_in) + 1;
86
87     struct t_String *s = MAKENEW(String);
88     s = string_init(s);
89     dup = malloc(sizeof(char) * length);

```

```

90     if (!dup) {
91         fprintf(stderr, "Out of memory in string_value.\n");
92         exit(1);
93     }
94     s->String.value = strcpy(dup, s_in);
95     return s;
96 }
97
98 struct t_Boolean *boolean_init(struct t_Boolean *this){
99     object_init((struct t_Object *) (this));
100     this->Boolean.value = 0;
101     return this;
102 }
103
104 struct t_Integer *integer_init(struct t_Integer *this){
105     object_init((struct t_Object *) (this));
106     this->Integer.value = 0;
107     return this;
108 }
109
110 struct t_Float *float_init(struct t_Float *this){
111     object_init((struct t_Object *) (this));
112     this->Float.value = 0.0;
113     return this;
114 }
115
116 struct t_Object *object_init(struct t_Object *this){
117     this->Object.v_system = &global_system;
118     return this;
119 }
120
121 struct t_String *string_init(struct t_String *this)
122 {
123     object_init((struct t_Object *) (this));
124     this->String.value = NULL;
125     return this;
126 }
127
128 struct t_System *system_init(struct t_System *this)
129 {
130     this->System.v_err = MAKENEW(Printer);
131     this->System.v_in = MAKENEW(Scanner);
132     this->System.v_out = MAKENEW(Printer);
133     this->System.v_argc = MAKENEW(Integer);
134
135     this->System.v_err->Printer.target = stderr;
136     this->System.v_in->Scanner.source = stdin;
137     this->System.v_out->Printer.target = stdout;
138     this->System.v_argc->Integer.value = global_argc;
139     this->Object.v_system =
140         this->System.v_err->Object.v_system =
141         this->System.v_in->Object.v_system =
142         this->System.v_out->Object.v_system =
143         this->System.v_argc->Object.v_system = this;
144     return this;
145 };
146
147 struct t_Printer *printer_init(struct t_Printer *this, struct t_Boolean *v_stdout)
148 {
149     object_init((struct t_Object *) (this));
150     this->Printer.target = v_stdout->Boolean.value ? stdout : stderr;
151     return this;
152 }
153
154 struct t_Scanner *scanner_init(struct t_Scanner *this)

```

```

155 {
156     object_init((struct t_Object *) (this));
157     this->Scanner.source = stdin;
158 }
159
160 struct t_Integer *float_to_i(struct t_Float *this){
161     return integer_value((int)(this->Float.value));
162 }
163
164 struct t_Float *integer_to_f(struct t_Integer *this){
165     return float_value((double)(this->Integer.value));
166 }
167
168 void toendl(FILE *in) {
169     int c = 0;
170     while (1) {
171         c = fgetc(in);
172         if (c == '\n' || c == '\r' || c == EOF) break;
173     }
174 }
175
176 struct t_Float *scanner_scan_float(struct t_Scanner *this)
177 {
178     double dval;
179     fscanf(this->Scanner.source, "%lf", &dval);
180     toendl(this->Scanner.source);
181
182     return float_value(dval);
183 }
184
185 struct t_Integer *scanner_scan_integer(struct t_Scanner *this)
186 {
187     int ival;
188     fscanf(this->Scanner.source, "%d", &ival);
189     toendl(this->Scanner.source);
190     return integer_value(ival);
191 }
192
193 struct t_String *scanner_scan_string(struct t_Scanner *this)
194 {
195     char *inpstr = NULL;
196     struct t_String *astring = NULL;
197
198     inpstr = stack_overflow_getline(this->Scanner.source);
199     astring = string_value(inpstr);
200
201     free(inpstr);
202     return astring;
203 }
204
205 void printer_print_float(struct t_Printer *this, struct t_Float *v_arg)
206 {
207     fprintf(this->Printer.target, "%lf", v_arg->Float.value);
208 }
209
210 void printer_print_integer(struct t_Printer *this, struct t_Integer *v_arg)
211 {
212     fprintf(this->Printer.target, "%d", v_arg->Integer.value);
213 }
214
215 void printer_print_string(struct t_Printer *this, struct t_String *v_arg)
216 {
217     fprintf(this->Printer.target, "%s", v_arg->String.value);
218 }
219

```

```

220 void system_exit(struct t_System *this, struct t_Integer *v_code) {
221     exit(INTEGER_OF(v_code));
222 }
223
224
225 struct t_String **get_gamma_args(char **argv, int argc) {
226     struct t_String **args = NULL;
227     int i = 0;
228
229     if (!argc) return NULL;
230     args = ONEDIM_ALLOC(struct t_String *, argc);
231     for (i = 0; i < argc; ++i)
232         args[i] = string_value(argv[i]);
233     args[i] = NULL;
234
235     return args;
236 }
237
238
239
240 char *stack_overflow_getline(FILE *in) {
241     char * line = malloc(100), * linep = line;
242     size_t lenmax = 100, len = lenmax;
243     int c;
244
245     if(line == NULL)
246         return NULL;
247
248     for(;;) {
249         c = fgetc(in);
250         if(c == EOF)
251             break;
252
253         if(--len == 0) {
254             len = lenmax;
255             char * linen = realloc(linep, lenmax * 2);
256
257             if(linen == NULL) {
258                 free(linep);
259                 return NULL;
260             }
261             line = linen + (line - linep);
262             linep = linen;
263         }
264
265         if((*line++ = c) == '\n')
266             break;
267     }
268     *line = '\0';
269     return linep;
270 }

```

Source 62: "headers/gamma-builtin-functions.h"

```

1  #include <stdarg.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  typedef struct {
6      int generation;
7      char* class;
8      char** ancestors;
9  } ClassInfo;

```



```

10
11
12 ClassInfo M_Boolean;
13 ClassInfo M_Float;
14 ClassInfo M_Integer;
15 ClassInfo M_Object;
16 ClassInfo M_Printer;
17 ClassInfo M_Scanner;
18 ClassInfo M_String;
19 ClassInfo M_System;
20
21
22 /*
23      Initializes the given ClassInfo
24 */
25 void class_info_init(ClassInfo* meta, int num_args, ...) {
26
27     int i;
28     va_list objtypes;
29     va_start(objtypes, num_args);
30
31     meta->ancestors = malloc(sizeof(char *) * num_args);
32
33     if (meta->ancestors == NULL) {
34         printf("\nMemory error - class_info_init failed\n");
35         exit(0);
36     }
37     for(i = 0; i < num_args; i++) {
38         meta->ancestors[i] = va_arg(objtypes, char * );
39     }
40     meta->generation = num_args - 1;
41     meta->class = meta->ancestors[meta->generation];
42     va_end(objtypes);
43 }
44
45
46 void init_built_in_infos() {
47     class_info_init(&M_Boolean, 2, m_classes[T_OBJECT], m_classes[T_BOOLEAN]);
48     class_info_init(&M_Float, 2, m_classes[T_OBJECT], m_classes[T_FLOAT]);
49     class_info_init(&M_Integer, 2, m_classes[T_OBJECT], m_classes[T_INTEGER]);
50     class_info_init(&M_Object, 1, m_classes[T_OBJECT]);
51     class_info_init(&M_Printer, 2, m_classes[T_OBJECT], m_classes[T_PRINTER]);
52     class_info_init(&M_Scanner, 2, m_classes[T_OBJECT], m_classes[T_SCANNER]);
53     class_info_init(&M_String, 2, m_classes[T_OBJECT], m_classes[T_STRING]);
54     class_info_init(&M_System, 2, m_classes[T_OBJECT], m_classes[T_SYSTEM]);
55 }

```

Source 63: "headers/gamma-builtin-meta.h"

```

1
2
3 /*
4  * Structures for each of the objects.
5  */
6 struct t_Boolean;
7 struct t_Float;
8 struct t_Integer;
9 struct t_Object;
10 struct t_Printer;
11 struct t_Scanner;
12 struct t_String;
13 struct t_System;
14

```

```

15
16 struct t_Boolean {
17     ClassInfo *meta;
18
19     struct {
20         struct t_System *v_system;
21     } Object;
22
23
24     struct { unsigned char value; } Boolean;
25 };
26
27
28 struct t_Float {
29     ClassInfo *meta;
30
31     struct {
32         struct t_System *v_system;
33     } Object;
34
35
36     struct { double value; } Float;
37 };
38
39
40 struct t_Integer {
41     ClassInfo *meta;
42
43     struct {
44         struct t_System *v_system;
45     } Object;
46
47
48     struct { int value; } Integer;
49 };
50
51
52 struct t_Object {
53     ClassInfo *meta;
54
55     struct {
56         struct t_System *v_system;
57     } Object;
58 };
59
60
61 struct t_Printer {
62     ClassInfo *meta;
63
64     struct {
65         struct t_System *v_system;
66     } Object;
67
68
69     struct { FILE *target; } Printer;
70 };
71
72
73 struct t_Scanner {
74     ClassInfo *meta;
75
76     struct {
77         struct t_System *v_system;
78     } Object;
79

```

```

80     struct { FILE *source; } Scanner;
81 };
82
83
84
85 struct t_String {
86     ClassInfo *meta;
87
88     struct {
89         struct t_System *v_system;
90     } Object;
91
92
93     struct { char *value; } String;
94 };
95
96
97 struct t_System {
98     ClassInfo *meta;
99
100     struct {
101         struct t_System *v_system;
102     } Object;
103
104
105     struct {
106         struct t_Printer *v_err;
107         struct t_Scanner *v_in;
108         struct t_Printer *v_out;
109         struct t_Integer *v_argc;
110     } System;
111 };

```

Source 64: "headers/gamma-builtin-struct.h"

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <math.h>
5
6  #define BYTE unsigned char
7
8  #define PROMOTE_INTEGER(ival)    integer_value((ival))
9  #define PROMOTE_FLOAT(fval)     float_value((fval))
10 #define PROMOTE_STRING(sval)    string_value((sval))
11 #define PROMOTE_BOOL(bval)      bool_value((bval))
12
13 #define LIT_INT(lit_int)        PROMOTE_INTEGER(lit_int)
14 #define LIT_FLOAT(litflt)       PROMOTE_FLOAT(litflt)
15 #define LIT_STRING(lit_str)     PROMOTE_STRING(lit_str)
16 #define LIT_BOOL(lit_bool)      PROMOTE_BOOL(lit_bool)
17
18 #define ADD_INT_INT(l, r)        PROMOTE_INTEGER(INTEGER_OF(l) + INTEGER_OF(r))
19 #define ADD_FLOAT_FLOAT(l, r)    PROMOTE_FLOAT(FLOAT_OF(l) + FLOAT_OF(r))
20 #define SUB_INT_INT(l, r)        PROMOTE_INTEGER(INTEGER_OF(l) - INTEGER_OF(r))
21 #define SUB_FLOAT_FLOAT(l, r)    PROMOTE_FLOAT(FLOAT_OF(l) - FLOAT_OF(r))
22 #define PROD_INT_INT(l, r)       PROMOTE_INTEGER(INTEGER_OF(l) * INTEGER_OF(r))
23 #define PROD_FLOAT_FLOAT(l, r)   PROMOTE_FLOAT(FLOAT_OF(l) * FLOAT_OF(r))
24 #define DIV_INT_INT(l, r)        PROMOTE_INTEGER(INTEGER_OF(l) / INTEGER_OF(r))
25 #define DIV_FLOAT_FLOAT(l, r)    PROMOTE_FLOAT(FLOAT_OF(l) / FLOAT_OF(r))
26 #define MOD_INT_INT(l, r)        PROMOTE_INTEGER(INTEGER_OF(l) % INTEGER_OF(r))
27 #define POW_INT_INT(l, r)        PROMOTE_INTEGER((int)pow(INTEGER_OF(l), INTEGER_OF(r)))

```

```

28 #define POW_FLOAT_FLOAT(l, r)  PROMOTE_FLOAT( pow(FLOAT_OF(l), FLOAT_OF(r)) )
29
30 #define MAKE_NEW2(type, meta) ((struct type *) (allocate_for(sizeof(struct type), &meta)))
31 #define MAKE_NEW(t_name) MAKE_NEW2(t_##t_name, M_##t_name)
32
33 #define CAST(type, v) ( (struct t_##type *) (v) )
34 #define VAL_OF(type, v) ( CAST(type, v) -> type.value )
35 #define BOOL_OF(b)      VAL_OF(Boolean, b)
36 #define FLOAT_OF(f)     VAL_OF(Float, f)
37 #define INTEGER_OF(i)   VAL_OF(Integer, i)
38 #define STRING_OF(s)    VAL_OF(String, s)
39
40 #define NEG_INTEGER(i)      PROMOTE_INTEGER(-INTEGER_OF(i))
41 #define NEG_FLOAT(f)       PROMOTE_FLOAT(-FLOAT_OF(f))
42 #define NOT_BOOLEAN(b)     PROMOTE_BOOL(!BOOL_OF(b))
43
44 #define BINOP(type, op, l, r) ( VAL_OF(type, l) op VAL_OF(type, r) )
45 #define PBINOP(type, op, l, r) PROMOTE_BOOL(BINOP(type, op, l, r))
46 #define IBINOP(op, l, r)      PBINOP(Integer, op, l, r)
47 #define FBINOP(op, l, r)      PBINOP(Float, op, l, r)
48 #define BBINOP(op, l, r)      PBINOP(Boolean, op, l, r)
49
50 #define NTEST_EQ_INT_INT(l, r)  IBINOP(==, l, r)
51 #define NTEST_NEQ_INT_INT(l, r) IBINOP(!=, l, r)
52 #define NTEST_LESS_INT_INT(l, r) IBINOP(<, l, r)
53 #define NTEST_GRTR_INT_INT(l, r) IBINOP(>, l, r)
54 #define NTEST_LEQ_INT_INT(l, r) IBINOP(<=, l, r)
55 #define NTEST_GEQ_INT_INT(l, r) IBINOP(>=, l, r)
56
57 #define NTEST_EQ_FLOAT_FLOAT(l, r)  FBINOP(==, l, r)
58 #define NTEST_NEQ_FLOAT_FLOAT(l, r) FBINOP(!=, l, r)
59 #define NTEST_LESS_FLOAT_FLOAT(l, r) FBINOP(<, l, r)
60 #define NTEST_GRTR_FLOAT_FLOAT(l, r) FBINOP(>, l, r)
61 #define NTEST_LEQ_FLOAT_FLOAT(l, r)  FBINOP(<=, l, r)
62 #define NTEST_GEQ_FLOAT_FLOAT(l, r)  FBINOP(>=, l, r)
63
64 #define CTEST_AND_BOOL_BOOL(l, r)  BBINOP(&&, l, r)
65 #define CTEST_OR_BOOL_BOOL(l, r)   BBINOP(||, l, r)
66 #define CTEST_NAND_BOOL_BOOL(l, r) PROMOTE_BOOL(( ! (BOOL_OF(l) && BOOL_OF(r)) ))
67 #define CTEST_NOR_BOOL_BOOL(l, r)  PROMOTE_BOOL(( ! (BOOL_OF(l) || BOOL_OF(r)) ))
68 #define CTEST_XOR_BOOL_BOOL(l, r)  PROMOTE_BOOL((!BOOL_OF(l) != !BOOL_OF(r)))
69
70 #define IS_CLASS(obj, kname) ( strcmp((obj)->meta->ancestors[obj->meta->generation], (
    kname)) == 0 )
71
72 #define ONE_DIM_ALLOC(type, len) ((type *) array_allocator(sizeof(type), (len)))
73
74 #define INIT_MAIN(options) \
75 struct t_String **str_args = NULL; \
76 char *gmain = NULL; \
77 --argc; ++argv; \
78 if (!argc) { \
79     fprintf(stderr, "Please select a main to use.  Available options: " options "\n"); \
80     exit(1); \
81 } \
82 gmain = *argv; ++argv; --argc; \
83 init_class_infos(); \
84 global_argc = argc; \
85 system_init(&global_system); \
86 str_args = get_gamma_args(argv, argc);
87
88
89 #define FAIL_MAIN(options) \
90 fprintf(stderr, "None of the available options were selected. Options were: " options "\n
    "); \

```

```

91 exit(1);
92
93 #define REFINE_FAIL(parent) \
94     fprintf(stderr, "Refinement fail: " parent "\n"); \
95     exit(1);

```

Source 65: "headers/gamma-preamble.h"

```

1
2 (** Types for the semantic abstract syntax tree *)
3
4 (** A switch for refinement or refinable checks *)
5 type refine_switch =
6   | Switch of string * (string * string) list * string (* host class, class/best-uid
7     list, switch uid *)
8   | Test of string * string list * string (* host class, class list, uid of switch *)
9
10 (** The type of a variable in the environment *)
11 type varkind = Instance of string | Local
12
13 (** The environment at any given statement. *)
14 type environment = (string * varkind) Map.Make(String).t
15
16 (** The ID can be built in (and so won't get mangled) or an array allocator. *)
17 type funcid = BuiltIn of string | FuncId of string | ArrayAlloc of string
18
19 (** An expression value — like in AST *)
20 type expr_detail =
21   | This
22   | Null
23   | Id of string
24   | NewObj of string * expr list * funcid
25   | Anonymous of string * expr list * Ast.func_def list (* Evaluation is delayed *)
26   | Literal of Ast.lit
27   | Assign of expr * expr (* memory := data — whether memory is good is a semantic
28     issue *)
29   | Deref of expr * expr (* road[pavement] *)
30   | Field of expr * string (* road.pavement *)
31   | Invoc of expr * string * expr list * funcid (* receiver.method(args) *
32     bestmethod_uid *)
33   | Unop of Ast.op * expr (* !x *)
34   | Binop of expr * Ast.op * expr (* x + y *)
35   | Refine of string * expr list * string option * refine_switch (* refinement, arg
36     list, opt ret type, switch *)
37   | Refinable of string * refine_switch (* desired refinement, list of classes
38     supporting refinement *)
39
40 (** An expression with a type tag *)
41 and expr = string * expr_detail
42
43 (** A statement tagged with an environment *)
44 and sstmt =
45   | Decl of Ast.var_def * expr option * environment
46   | If of (expr option * sstmt list) list * environment
47   | While of expr * sstmt list * environment
48   | Expr of expr * environment
49   | Return of expr option * environment
50   | Super of expr list * string * string * environment (**arglist, uid of super init,
51     superclass, env**)
52
53 (** A function definition *)
54 and func_def = {
55   returns : string option;

```

```

50     host      : string option;
51     name      : string;
52     static    : bool;
53     formals   : Ast.var_def list;
54     body      : sstmt list;
55     section   : Ast.class_section; (* Makes things easier later *)
56     inclass   : string;
57     uid       : string;
58     builtin   : bool;
59 }
60
61 (* A member is either a variable or some sort of function *)
62 type member_def = VarMem of Ast.var_def | MethodMem of func_def | InitMem of func_def
63
64 (* Things that can go in a class *)
65 type class_sections_def = {
66     privates : member_def list;
67     protects : member_def list;
68     publics  : member_def list;
69     refines  : func_def list;
70     mains    : func_def list;
71 }
72
73 (* Just pop init and main in there? *)
74 type class_def = {
75     klass      : string;
76     parent     : string option;
77     sections   : class_sections_def;
78 }
79
80 type program = class_def list

```

Source 66: "Sast.mli"

```

1  open StringModules
2
3  (* The detail of an expression *)
4  type cexpr_detail =
5      | This
6      | Null
7      | Id of string * Sast.varkind (* name, local/instance *)
8      | NewObj of string * string * cexpr list (* ctype * fname * args *)
9      | NewArr of string * string * cexpr list (* type (with []'s) * fname * args (sizes) *)
10     | Literal of Ast.lit
11     | Assign of cexpr * cexpr (* memory := data — whether memory is good is a semantic
12     | Deref of cexpr * cexpr (* road[pavement] *)
13     | Field of cexpr * string (* road.pavement *)
14     | Invoc of cexpr * string * cexpr list (* Invoc(receiver, functionname, args) *)
15     | Unop of Ast.op * cexpr (* !x *)
16     | Binop of cexpr * Ast.op * cexpr (* x + y *)
17     | Refine of cexpr list * string option * Sast.refine_switch (* arg list, opt ret type
18     | Refinable of Sast.refine_switch (* list of classes supporting refinement *)
19
20 (* The expression and its type *)
21 and cexpr = string * cexpr_detail
22
23 (* A statement which has cexpr detail *)
24 and cstmt =
25     | Decl of Ast.var_def * cexpr option * Sast.environment
26     | If of (cexpr option * cstmt list) list * Sast.environment

```

```

27 | While of cexpr * cstmt list * Sast.environment
28 | Expr of cexpr * Sast.environment
29 | Super of string * string * cexpr list (* class, fuid, args *)
30 | Return of cexpr option * Sast.environment
31
32 (* A c func is a simplified function (no host, etc) *)
33 and cfunc = {
34   returns : string option;
35   name : string; (* Combine uid and name into this *)
36   formals : Ast.var_def list;
37   body : cstmt list;
38   builtin : bool;
39   inklass : string; (* needed for THIS *)
40   static : bool;
41 }
42
43 (* The bare minimum for a struct representation *)
44 type class_struct = (string * Ast.var_def list) list (* All the data for this object from
45   the root (first item) down, paired with class name *)
46
47 (* A main is a class name and a function name for that main *)
48 type main_func = (string * string)
49
50 (* We actually need all the ancestry information, cause we're gonna do it the right way [
51   lists should go from object down] *)
52 type ancestry_info = (string list) lookup_map
53
54 (* A program is a map from all classes to their struct's, a list of all functions, and a
55   list of mainfuncs, and ancestor information *)
56 type program = class_struct lookup_map * cfunc list * main_func list * ancestry_info

```

Source 67: "Cast.mli"

```

1  #!/bin/bash
2
3  function errwith {
4    echo "$1" >&2
5    exit 1
6  }
7
8  function run_file {
9    test "$#" -lt 1 && errwith "Please give a file to test"
10   file=$1
11
12   test -e "$file" || errwith "File $file does not exist."
13   test -f "$file" || errwith "File $file is not a file."
14
15   echo "=====
16   echo "=====
17   echo "$file"
18   cat "$file"
19   echo "=====
20   echo "=====
21   ./bin/ray "$file" > ctest/test.c && ( cd ctest && ./compile && ./a.out Test )
22 }
23
24 for afile in "${@"}"; do
25   run_file "$afile"
26 done

```

Source 68: "run-compiler-test.sh"

```

1 open Ast
2
3 (** Various utility functions *)
4
5 (* Types *)
6 (**
7   Paramaterized variable typing for building binary ASTs
8   @see <http://caml.inria.fr/pub/docs/oreilly-book/html/book-ora016.html#toc19> For
9   more details on paramterized typing
10  *)
11 type ('a, 'b) either = Left of 'a | Right of 'b
12
13 (** Split a list of 'a 'b either values into a pair of 'a list and 'b list *)
14 let either_split eithers =
15   let rec split_eithers (left, right) = function
16     | [] -> (List.rev left, List.rev right)
17     | (Left(a))::rest -> split_eithers (a::left, right) rest
18     | (Right(b))::rest -> split_eithers (left, b::right) rest in
19   split_eithers ([], []) eithers
20
21 (** Reduce a list of options to the values in the Some constructors *)
22 let filter_option list =
23   let rec do_filter rlist = function
24     | [] -> List.rev rlist
25     | None::tl -> do_filter rlist tl
26     | (Some(v))::tl -> do_filter (v::rlist) tl in
27   do_filter [] list
28
29 let option_as_list = function
30   | Some(v) -> [v]
31   | _ -> []
32
33 let decide_option x = function
34   | true -> Some(x)
35   | _ -> None
36
37 (** Lexically compare two lists of comparable items *)
38 let rec lexical_compare list1 list2 = match list1, list2 with
39   | [], [] -> 0
40   | [], _ -> -1
41   | _, [] -> 1
42   | (x::xs), (y::ys) -> if x < y then -1 else if x > y then 1 else lexical_compare xs
43   ys
44
45 (**
46   Loop through a list and find all the items that are minimum with respect to the total
47   ordering cmp. (If an item is found to be a minimum, any item that is found to
48   be equal to the item is in the returned list.) Note can return any size list.
49   @param cmp A comparator function
50   @param alist A list of items
51   @return A list of one or more items deemed to be the minimum by cmp.
52  *)
53 let find_all_min cmp alist =
54   let rec min_find found items = match found, items with
55     | _, [] -> List.rev found (* Return in the same order at least *)
56     | [], i::is -> min_find [i] is
57     | (f::fs), (i::is) -> let result = cmp i f in
58       if result = 0 then min_find (i::found) is
59       else if result < 0 then min_find [i] is
60       else min_find found is in
61   min_find [] alist
62
63 (**
64   Either monad stuffage
65  *)

```



```

63     @param value A monad
64     @param func A function to run on a monad
65     @return The result of func if we're on the left side, or the error if we're on the
        right
66 *)
67 let (|->) value func =
68     match value with
69     | Left(v) -> func(v)
70     | Right(problem) -> Right(problem)
71
72 (** Sequence a bunch of monadic actions together, piping results together *)
73 let rec seq init actions = match init, actions with
74 | Right(issue), _ -> Right(issue)
75 | Left(data), [] -> Left(data)
76 | Left(data), act::ions -> seq (act data) ions
77
78 (**
79     Return the length of a block — i.e. the total number of statements (recursively) in
        it
80     @param stmt_list A list of stmt type objects
81     @return An int encoding the length of a block
82 *)
83 let get_statement_count stmt_list =
84     let rec do_count stmts blocks counts = match stmts, blocks with
85     | [], [] -> counts
86     | [], _ -> do_count blocks [] counts
87     | (stmt::rest), _ -> match stmt with
88     | Decl(_) -> do_count rest blocks (counts + 1)
89     | Expr(_) -> do_count rest blocks (counts + 1)
90     | Return(_) -> do_count rest blocks (counts + 1)
91     | Super(_) -> do_count rest blocks (counts + 1)
92     | While(_, block) -> do_count rest (block @ blocks) (counts + 1)
93     | If(parts) ->
94         let ifblocks = List.map snd parts in
95         let ifstmts = List.flatten ifblocks in
96         do_count rest (ifstmts @ blocks) (counts + 1) in
97     do_count stmt_list [] 0

```

Source 69: "Util.ml"

```

1  open Parser
2  open Ast
3
4  (** Provides functionality for examining values used in the compilation pipeline. *)
5
6  (* TOKEN stuff *)
7  (** Convert a given token to a string representation for output *)
8  let token_to_string = function
9  | SPACE(n) -> "SPACE(" ^ string_of_int n ^ ")"
10 | COLON -> "COLON"
11 | NEWLINE -> "NEWLINE"
12 | THIS -> "THIS"
13 | ARRAY -> "ARRAY"
14 | REFINABLE -> "REFINABLE"
15 | AND -> "AND"
16 | OR -> "OR"
17 | XOR -> "XOR"
18 | NAND -> "NAND"
19 | NOR -> "NOR"
20 | NOT -> "NOT"
21 | EQ -> "EQ"
22 | NEQ -> "NEQ"
23 | LT -> "LT"

```

```

24 | LEQ -> "LEQ"
25 | GT -> "GT"
26 | GEQ -> "GEQ"
27 | LBRACKET -> "LBRACKET"
28 | RBRACKET -> "RBRACKET"
29 | LPAREN -> "LPAREN"
30 | RPAREN -> "RPAREN"
31 | LBRACE -> "LBRACE"
32 | RBRACE -> "RBRACE"
33 | SEMI -> "SEMI"
34 | COMMA -> "COMMA"
35 | PLUS -> "PLUS"
36 | MINUS -> "MINUS"
37 | TIMES -> "TIMES"
38 | DIVIDE -> "DIVIDE"
39 | MOD -> "MOD"
40 | POWER -> "POWER"
41 | PLUSA -> "PLUSA"
42 | MINUSA -> "MINUSA"
43 | TIMESA -> "TIMESA"
44 | DIVIDEA -> "DIVIDEA"
45 | MODA -> "MODA"
46 | POWERA -> "POWERA"
47 | IF -> "IF"
48 | ELSE -> "ELSE"
49 | ELSIF -> "ELSIF"
50 | WHILE -> "WHILE"
51 | RETURN -> "RETURN"
52 | CLASS -> "CLASS"
53 | EXTEND -> "EXTEND"
54 | SUPER -> "SUPER"
55 | INIT -> "INIT"
56 | NULL -> "NULL"
57 | VOID -> "VOID"
58 | REFINES -> "REFINE"
59 | REFINES -> "REFINES"
60 | TO -> "TO"
61 | PRIVATE -> "PRIVATE"
62 | PUBLIC -> "PUBLIC"
63 | PROTECTED -> "PROTECTED"
64 | DOT -> "DOT"
65 | MAIN -> "MAIN"
66 | NEW -> "NEW"
67 | ASSIGN -> "ASSIGN"
68 | ID(vid) -> Printf.sprintf "ID(%s)" vid
69 | TYPE(tid) -> Printf.sprintf "TYPE(%s)" tid
70 | BLIT(bool) -> Printf.sprintf "BLIT(%B)" bool
71 | ILIT(inum) -> Printf.sprintf "ILIT(%d)" inum
72 | FLIT(fnum) -> Printf.sprintf "FLIT(%f)" fnum
73 | SLIT(str) -> Printf.sprintf "SLIT(\"%s\")" (str)
74 | EOF -> "EOF"
75
76 | (** Convert token to its (assumed) lexographical source *)
77 | let descan = function
78 | | COLON -> ":"
79 | | NEWLINE -> "\n"
80 | | SPACE(n) -> String.make n ' '
81 | | REFINABLE -> "refinable"
82 | | AND -> "and"
83 | | OR -> "or"
84 | | XOR -> "xor"
85 | | NAND -> "nand"
86 | | NOR -> "nor"
87 | | NOT -> "not"
88 | | EQ -> "="

```

```

89 | NEQ -> "=/"
90 | LT -> "<"
91 | LEQ -> "<="
92 | GT -> ">"
93 | GEQ -> ">="
94 | ARRAY -> "[]"
95 | LBRACKET -> "["
96 | RBRACKET -> "]"
97 | LPAREN -> "("
98 | RPAREN -> ")"
99 | LBRACE -> "{"
100 | RBRACE -> "}"
101 | SEMI -> ";"
102 | COMMA -> ","
103 | PLUS -> "+"
104 | MINUS -> "-"
105 | TIMES -> "*"
106 | DIVIDE -> "/"
107 | MOD -> "%"
108 | POWER -> "^"
109 | PLUSA -> "+="
110 | MINUSA -> "-="
111 | TIMESA -> "*="
112 | DIVIDEA -> "/="
113 | MODA -> "%="
114 | POWERA -> "^="
115 | IF -> "if"
116 | ELSE -> "else"
117 | ELSIF -> "elseif"
118 | WHILE -> "while"
119 | RETURN -> "return"
120 | CLASS -> "class"
121 | EXTEND -> "extends"
122 | SUPER -> "super"
123 | INIT -> "init"
124 | NULL -> "null"
125 | VOID -> "void"
126 | THIS -> "this"
127 | REFINE -> "refine"
128 | REFINES -> "refinement"
129 | TO -> "to"
130 | PRIVATE -> "private"
131 | PUBLIC -> "public"
132 | PROTECTED -> "protected"
133 | DOT -> "."
134 | MAIN -> "main"
135 | NEW -> "new"
136 | ASSIGN -> ":="
137 | ID(var) -> var
138 | TYPE(typ) -> typ
139 | BLIT(b) -> if b then "true" else "false"
140 | ILIT(i) -> string_of_int(i)
141 | FLIT(f) -> string_of_float(f)
142 | SLIT(s) -> Format.sprintf "%s" s
143 | EOF -> "eof"
144
145 (**
146   Given a lexing function and a lexing buffer, consume tokens until
147   the end of file is reached. Return the generated tokens.
148   @param lexfun A function that takes a lexbuf and returns a token
149   @param lexbuf A lexicographical buffer from Lexing
150   @return A list of scanned tokens
151 *)
152 let token_list (lexfun : Lexing.lexbuf -> token) (lexbuf : Lexing.lexbuf) =
153   let rec list_tokens rtokens =

```

```

154         match (lexfun lexbuf) with
155         | EOF -> List.rev (EOF::rtokens)
156         | tk -> list_tokens (tk::rtokens) in
157     list_tokens []
158
159     (**
160     Scan a list of tokens from an input file.
161     @param source A channel to get tokens from
162     @return A list of tokens taken from a source
163     *)
164     let from_channel source = token_list Scanner.token (Lexing.from_channel source)
165
166     (**
167     Print a list of tokens to stdout.
168     @param tokens A list of tokens
169     @return Only returns a unit
170     *)
171     let print_token_list tokens = print_string (String.concat " " (List.map token_to_string tokens))
172
173     (**
174     Used to print out de-whitespacing lines which consist of a number (indentation), a
175     list
176     of tokens (the line), and whether there is a colon at the end of the line.
177     @return Only returns a unit
178     *)
179     let print_token_line = function
180     | (space, toks, colon) ->
181         print_string "(" ^ string_of_int space ^ "," ^ string_of_bool colon ^ ") ";
182         print_token_list toks
183
184     (**
185     Print out a list of tokens with a specific header and some extra margins
186     @param header A nonsemantic string to preface our list
187     @param toks A list of tokens
188     @return Only returns a unit
189     *)
190     let pprint_token_list header toks = print_string header ; print_token_list toks ;
191         print_newline ()
192
193     (**
194     Print out de-whitespacing lines (see print_token_line) for various lines, but with a
195     header.
196     @param header A nonsemantic string to preface our list
197     @param lines A list of line representations (number of spaces, if it ends in a colon,
198     a list of tokens)
199     @return Only returns a unit
200     *)
201     let pprint_token_lines header lines =
202     let spaces = String.make (String.length header) ' ' in
203     let rec lines_printer prefix = function
204     | line::rest ->
205         print_string prefix;
206         print_token_line line;
207         print_newline ();
208         lines_printer spaces rest
209     | [] -> () in
210     lines_printer header lines
211
212     (** The majority of the following functions are relatively direct AST to string
213     operations *)
214
215     (* Useful for both sAST and AST *)
216     let _id x = x
217     let inspect_str_list stringer a_list = Printf.sprintf "[%s]" (String.concat ", " (List.

```

```

213     map stringer a_list))
214 let inspect_opt stringer = function
215 | None -> "None"
216 | Some(v) -> Printf.sprintf "Some(%s)" (stringer v)
217
218 (* AST Parser Stuff *)
219 let inspect_ast_lit (lit : Ast.lit) = match lit with
220 | Int(i) -> Printf.sprintf "Int(%d)" i
221 | Float(f) -> Printf.sprintf "Float(%f)" f
222 | String(s) -> Printf.sprintf "String(\"%s\")" s
223 | Bool(b) -> Printf.sprintf "Bool(%B)" b
224
225 let inspect_ast_arith (op : Ast.arith) = match op with
226 | Add -> "Add"
227 | Sub -> "Sub"
228 | Prod -> "Prod"
229 | Div -> "Div"
230 | Mod -> "Mod"
231 | Neg -> "Neg"
232 | Pow -> "Pow"
233
234 let inspect_ast_numtest (op : Ast.numtest) = match op with
235 | Eq -> "Eq"
236 | Neq -> "Neq"
237 | Less -> "Less"
238 | Grtr -> "Grtr"
239 | Leq -> "Leq"
240 | Geq -> "Geq"
241
242 let inspect_ast_combtest (op : Ast.combtest) = match op with
243 | And -> "And"
244 | Or -> "Or"
245 | Nand -> "Nand"
246 | Nor -> "Nor"
247 | Xor -> "Xor"
248 | Not -> "Not"
249
250 let inspect_ast_op (op : Ast.op) = match op with
251 | Arithmetic(an_op) -> Printf.sprintf "Arithmetic(%s)" (inspect_ast_arith an_op)
252 | NumTest(an_op) -> Printf.sprintf "NumTest(%s)" (inspect_ast_numtest an_op)
253 | CombTest(an_op) -> Printf.sprintf "CombTest(%s)" (inspect_ast_combtest an_op)
254
255 let rec inspect_ast_expr (expr : Ast.expr) = match expr with
256 | Id(id) -> Printf.sprintf "Id(%s)" id
257 | This -> "This"
258 | Null -> "Null"
259 | NewObj(the_type, args) -> Printf.sprintf("NewObj(%s, %s)") the_type (
inspect_str_list inspect_ast_expr args)
260 | Anonymous(the_type, args, body) -> Printf.sprintf("Anonymous(%s, %s, %s)") the_type
(inspect_str_list inspect_ast_expr args) (inspect_str_list inspect_ast_func_def body)
261 | Literal(l) -> Printf.sprintf "Literal(%s)" (inspect_ast_lit l)
262 | Invoc(receiver, meth, args) -> Printf.sprintf "Invocation(%s, %s, %s)" (
inspect_ast_expr receiver) meth (inspect_str_list inspect_ast_expr args)
263 | Field(receiver, field) -> Printf.sprintf "Field(%s, %s)" (inspect_ast_expr receiver)
field
264 | Deref(var, index) -> Printf.sprintf "Deref(%s, %s)" (inspect_ast_expr var) (
inspect_ast_expr var)
265 | Unop(an_op, exp) -> Printf.sprintf "Unop(%s, %s)" (inspect_ast_op an_op) (
inspect_ast_expr exp)
266 | Binop(left, an_op, right) -> Printf.sprintf "Binop(%s, %s, %s)" (inspect_ast_op
an_op) (inspect_ast_expr left) (inspect_ast_expr right)
267 | Refine(fname, args, totype) -> Printf.sprintf "Refine(%s,%s,%s)" fname (
inspect_str_list inspect_ast_expr args) (inspect_opt_id totype)
268 | Assign(the_var, the_expr) -> Printf.sprintf "Assign(%s, %s)" (inspect_ast_expr

```

```

268 the_var) (inspect_ast_expr the_expr)
269 | Refinable(the_var) -> Printf.sprintf "Refinable(%s)" the_var
269 and inspect_ast_var_def (var : Ast.var_def) = match var with
270 | (the_type, the_var) -> Printf.sprintf "(%s, %s)" the_type the_var
271 and inspect_ast_stmt (stmt : Ast.stmt) = match stmt with
272 | Decl(the_def, the_expr) -> Printf.sprintf "Decl(%s, %s)" (inspect_ast_var_def
the_def) (inspect_opt inspect_ast_expr the_expr)
273 | If(clauses) -> Printf.sprintf "If(%s)" (inspect_str_list inspect_ast_clause clauses
)
274 | While(pred, body) -> Printf.sprintf "While(%s, %s)" (inspect_ast_expr pred) (
inspect_str_list inspect_ast_stmt body)
275 | Expr(the_expr) -> Printf.sprintf "Expr(%s)" (inspect_ast_expr the_expr)
276 | Return(the_expr) -> Printf.sprintf "Return(%s)" (inspect_opt inspect_ast_expr
the_expr)
277 | Super(args) -> Printf.sprintf "Super(%s)" (inspect_str_list inspect_ast_expr args)
278 and inspect_ast_clause ((opt_expr, body) : Ast.expr option * Ast.stmt list) =
279 Printf.sprintf "(%s, %s)" (inspect_opt inspect_ast_expr opt_expr) (inspect_str_list
inspect_ast_stmt body)
280 and inspect_ast_class_section (sect : Ast.class_section) = match sect with
281 | Publics -> "Publics"
282 | Protects -> "Protects"
283 | Privates -> "Privates"
284 | Refines -> "Refines"
285 | Mains -> "Mains"
286 and inspect_ast_func_def (func : Ast.func_def) =
287 Printf.sprintf "{ returns = %s, host = %s, name = %s, static = %B, formals = %s, body
= %s, section = %s, inclass = %s, uid = %s }"
288 (inspect_opt _id func.returns)
289 (inspect_opt _id func.host)
290 func.name
291 func.static
292 (inspect_str_list inspect_ast_var_def func.formals)
293 (inspect_str_list inspect_ast_stmt func.body)
294 (inspect_ast_class_section func.section)
295 func.inclass
296 func.uid
297
298 let inspect_ast_member_def (mem : Ast.member_def) = match mem with
299 | VarMem(vmem) -> Printf.sprintf "VarMem(%s)" (inspect_ast_var_def vmem)
300 | MethodMem(mmem) -> Printf.sprintf "MethodMem(%s)" (inspect_ast_func_def mmem)
301 | InitMem(imem) -> Printf.sprintf "InitMem(%s)" (inspect_ast_func_def imem)
302
303 let inspect_ast_class_sections (sections : Ast.class_sections_def) =
304 Printf.sprintf "{ privates = %s, protects = %s, publics = %s, refines = %s, mains = %
s }"
305 (inspect_str_list inspect_ast_member_def sections.privates)
306 (inspect_str_list inspect_ast_member_def sections.protects)
307 (inspect_str_list inspect_ast_member_def sections.publics)
308 (inspect_str_list inspect_ast_func_def sections.refines)
309 (inspect_str_list inspect_ast_func_def sections.mains)
310
311 let inspect_ast_class_def (the_class : Ast.class_def) =
312 Printf.sprintf "{ klass = %s, parent = %s, sections = %s }"
313 the_class.klass
314 (inspect_opt _id the_class.parent)
315 (inspect_ast_class_sections the_class.sections)

```

Source 70: "Inspector.ml"

```

1 open Util
2
3 module StringSet = Set.Make(String)
4 module StringMap = Map.Make(String)

```

```

5
6 (** A place for StringSet and StringMap to live. *)
7
8 (**
9  Convenience type to make reading table types easier. A lookup_table
10 is a primary key -> second key -> value map (i.e. the values of the
11 first StringMap are themselves StringMap maps...
12 *)
13 type 'a lookup_table = 'a StringMap.t StringMap.t
14
15 (**
16 Convenience type to make reading string maps easier. A lookup_map
17 is just a StringMap map.
18 *)
19 type 'a lookup_map = 'a StringMap.t
20
21
22 (** Print the contents of a lookup_map *)
23 let print_lookup_map map stringer =
24   let print_item (secondary, item) =
25     print_string (stringer secondary item) in
26   List.iter print_item (StringMap.bindings map)
27
28 (** Print the contents of a lookup_table *)
29 let print_lookup_table table stringer =
30   let print_lookup_map (primary, table) =
31     print_lookup_map table (stringer primary) in
32   List.iter print_lookup_map (StringMap.bindings table)
33
34
35 (**
36 To put it into symbols, we have builder : (StringMap, errorList) -> item -> (
37 StringMap', errorList')
38 @param builder A function that accepts a StringMap/(error list) pair and a new item
39 and returns a new pair with either an updated map or updated error list
40 @param alist The list of data to build the map out of.
41 *)
42 let build_map_track_errors builder alist =
43   match List.fold_left builder (StringMap.empty, []) alist with
44   | (value, []) -> Left(value)
45   | (_, errors) -> Right(errors)
46
47 (**
48 Look a value up in a map
49 @param key The key to look up
50 @param map The map to search in
51 @return Some(value) or None
52 *)
53 let map_lookup key map = if StringMap.mem key map
54   then Some(StringMap.find key map)
55   else None
56
57 (**
58 Look a list up in a map
59 @param key The key to look up
60 @param map The map to search in
61 @return a list or None
62 *)
63 let map_lookup_list key map = if StringMap.mem key map
64   then StringMap.find key map
65   else []
66
67 (** Updating a string map that has list of possible values *)
68 let add_map_list key value map =
69   let old = map_lookup_list key map in

```

```

69   StringMap.add key (value::old) map
70
71   (** Updating a string map that has a list of possible values with a bunch of new values
72   *)
73   let concat_map_list key values map =
74     let old = map_lookup_list key map in
75     StringMap.add key (values@old) map
76
77   (** Update a map but keep track of collisions *)
78   let add_map_unique key value (map, collisions) =
79     if StringMap.mem key map
80       then (map, key::collisions)
81       else (StringMap.add key value map, collisions)

```

Source 71: "StringModules.ml"

```

1  val token_to_string : Parser.token -> string
2  val descant : Parser.token -> string
3  val token_list : (Lexing.lexbuf -> Parser.token) -> Lexing.lexbuf -> Parser.token list
4  val from_channel : Pervasives.in_channel -> Parser.token list
5  val pprint_token_list : string -> Parser.token list -> unit
6  val pprint_token_lines : string -> (int * Parser.token list * bool) list -> unit
7  val inspect_ast_lit : Ast.lit -> string
8  val inspect_ast_arith : Ast.arith -> string
9  val inspect_ast_numtest : Ast.numtest -> string
10 val inspect_ast_combtest : Ast.combtest -> string
11 val inspect_ast_op : Ast.op -> string
12 val inspect_ast_expr : Ast.expr -> string
13 val inspect_ast_var_def : Ast.var_def -> string
14 val inspect_ast_stmt : Ast.stmt -> string
15 val inspect_ast_clause : Ast.expr option * Ast.stmt list -> string
16 val inspect_ast_class_section : Ast.class_section -> string
17 val inspect_ast_func_def : Ast.func_def -> string
18 val inspect_ast_member_def : Ast.member_def -> string
19 val inspect_ast_class_sections : Ast.class_sections_def -> string
20 val inspect_ast_class_def : Ast.class_def -> string

```

Source 72: "Inspector.mli"

```

1  let _ =
2    let tokens = Inspector.from_channel stdin in
3    let classes = Parser.cdecls (WhiteSpace.lex toks tokens) (Lexing.from_string "") in
4    let inspect_classes = List.map Inspector.inspect_ast_class_def classes in
5    print_string (String.concat "\n\n" inspect_classes); print_newline ()

```

Source 73: "inspect.ml"

```

1  open Parser
2  open Ast
3
4  (**
5   A collection of pretty printing functions.
6   I don't believe it actually needs the Parser dependency.
7   Should probably absorb a fair margin from other files like Inspector.ml
8   *)
9
10 let indent level = String.make (level*2) ' '
11 let _id x = x

```



```

12
13 let pp_lit = function
14 | Int(i)    -> Printf.sprintf "Int(%d)" i
15 | Float(f)  -> Printf.sprintf "Float(%f)" f
16 | String(s) -> Printf.sprintf "String(%s)" s
17 | Bool(b)   -> Printf.sprintf "Bool(%B)" b
18
19 let pp_arith = function
20 | Add -> "Add"
21 | Sub -> "Sub"
22 | Prod -> "Prod"
23 | Div -> "Div"
24 | Mod -> "Mod"
25 | Neg -> "Neg"
26 | Pow -> "Pow"
27
28 let pp_numtest = function
29 | Eq  -> "Eq"
30 | Neq -> "Neq"
31 | Less -> "Less"
32 | Grtr -> "Grtr"
33 | Leq  -> "Leq"
34 | Geq  -> "Geq"
35
36 let pp_combtest = function
37 | And  -> "And"
38 | Or   -> "Or"
39 | Nand -> "Nand"
40 | Nor  -> "Nor"
41 | Xor  -> "Xor"
42 | Not  -> "Not"
43
44 let pp_op = function
45 | Arithmetic(an_op) -> Printf.sprintf "Arithmetic(%s)" (pp_arith an_op)
46 | NumTest(an_op)    -> Printf.sprintf "NumTest(%s)" (pp_numtest an_op)
47 | CombTest(an_op)   -> Printf.sprintf "CombTest(%s)" (pp_combtest an_op)
48
49 let pp_str_list stringer a_list depth = Printf.sprintf "[ %s ]" (String.concat ", " (List
    .map stringer a_list))
50 let pp_opt stringer = function
51 | None -> "None"
52 | Some(v) -> Printf.sprintf "Some(%s)" (stringer v)
53
54 let rec pp_expr depth = function
55 | Id(id) -> Printf.sprintf "Id(%s)" id
56 | This -> "This"
57 | Null -> "Null"
58 | NewObj(the_type, args) -> Printf.sprintf("\n%sNewObj(%s, %s)" (indent depth)
    the_type (pp_str_list (pp_expr depth) args depth)
59 | Anonymous(the_type, args, body) -> Printf.sprintf("\n%sAnonymous(%s, %s, %s)" (
    indent depth) the_type (pp_str_list (pp_expr depth) args depth) (pp_str_list (
    pp_func_def depth) body depth)
60 | Literal(l) -> Printf.sprintf "\n%sLiteral(%s)" (indent depth) (pp_lit l)
61 | Invoc(receiver, meth, args) -> Printf.sprintf "\n%sInvocation(%s, %s, %s)" (indent
    depth) ((pp_expr (depth+1)) receiver) meth (pp_str_list (pp_expr (depth+1)) args
    depth)
62 | Field(receiver, field) -> Printf.sprintf "\n%sField(%s, %s)" (indent depth) ((
    pp_expr depth) receiver) field
63 | Deref(var, index) -> Printf.sprintf "\n%sDeref(%s, %s)" (indent depth) ((pp_expr
    depth) var) ((pp_expr depth) var)
64 | Unop(an_op, exp) -> Printf.sprintf "\n%sUnop(%s, %s)" (indent depth) (pp_op an_op)
    ((pp_expr depth) exp)
65 | Binop(left, an_op, right) -> Printf.sprintf "\n%sBinop(%s, %s, %s)" (indent depth)
    (pp_op an_op) ((pp_expr depth) left) ((pp_expr depth) right)
66 | Refine(fname, args, totype) -> Printf.sprintf "Refine(%s, %s, %s)" fname (

```

```

67     pp_str_list (pp_expr (depth+1)) args (depth+1)) (pp_opt _id totype)
    | Assign(the_var, the_expr) -> Printf.sprintf "\n%sAssign(%s, %s)" (indent depth) ((
68     pp_expr (depth+1)) the_var) ((pp_expr (depth+1)) the_expr)
    | Refinable(the_var) -> Printf.sprintf "\n%sRefinable(%s)" (indent depth) the_var
69 and pp_var_def depth (the_type, the_var) = Printf.sprintf "\n%s(%s, %s)" (indent depth)
    the_type the_var
70 and pp_stmt depth = function
71 | Decl(the_def, the_expr) -> Printf.sprintf "\n%sDecl(%s, %s)" (indent depth) ((
    pp_var_def (depth+1)) the_def) (pp_opt (pp_expr depth) the_expr)
72 | If(clauses) -> Printf.sprintf "\n%sIf(%s)" (indent depth) (pp_str_list (
    inspect_clause depth) clauses depth)
73 | While(pred, body) -> Printf.sprintf "\n%sWhile(%s, %s)" (indent depth) ((pp_expr
    depth) pred) (pp_str_list (pp_stmt (depth+1)) body depth)
74 | Expr(the_expr) -> Printf.sprintf "\n%sExpr(%s)" (indent depth) ((pp_expr (depth+1))
    the_expr)
75 | Return(the_expr) -> Printf.sprintf "\n%sReturn(%s)" (indent depth) (pp_opt (pp_expr
    depth) the_expr)
76 | Super(args) -> Printf.sprintf "\n%sSuper(%s)" (indent depth) (pp_str_list (pp_expr
    depth) args depth)
77 and inspect_clause depth (opt_expr, body) = Printf.sprintf "(%s, %s)" (pp_opt (pp_expr
    depth) opt_expr) (pp_str_list (pp_stmt (depth+1)) body depth)
78 and class_section = function
79 | Publics -> "Publics"
80 | Protects -> "Protects"
81 | Privates -> "Privates"
82 | Refines -> "Refines"
83 | Mains -> "Mains"
84 and pp_func_def depth func = Printf.sprintf "\n%s{\n%sreturns = %s,\n%shost = %s,\n%sname
    = %s,\n%ssstatic = %B,\n%sformals = %s,\n%sbbody = %s,\n%sssection = %s,\n%sinklass = %
    s,\n%ssuid = %s\n%s}"
85     (indent (depth-1))
86     (indent depth)
87     (pp_opt _id func.returns)
88     (indent depth)
89     (pp_opt _id func.host)
90     (indent depth)
91     func.name
92     (indent depth)
93     func.static
94     (indent depth)
95     (pp_str_list (pp_var_def (depth+1)) func.formals depth)
96     (indent depth)
97     (pp_str_list (pp_stmt (depth+1)) func.body depth)
98     (indent depth)
99     (class_section func.section)
100     (indent depth)
101     func.inclass
102     (indent depth)
103     func.uid
104     (indent (depth-1))
105
106 let pp_member_def depth = function
107 | VarMem(vmem) -> Printf.sprintf "\n%sVarMem(%s)" (indent depth) (pp_var_def (depth
    +1) vmem)
108 | MethodMem(mmem) -> Printf.sprintf "\n%sMethodMem(%s)" (indent depth) (pp_func_def (
    depth+1) mmem)
109 | InitMem(imem) -> (*let fmt = "@[<v " ^^ (string_of_int depth) ^^ ">@,InitMem(%s)@
    ]" in*)
110     Format.sprintf "\n%sInitMem(%s)@"
111     (indent depth) (pp_func_def (depth+1) imem)
112     (*Format.sprintf fmt
113     (pp_func_def (depth+1) imem)*)
114
115 let pp_class_sections sections depth =
116     Format.sprintf "@[<v 3>@,{@[<v 2>@,privates = %s,@,protects = %s,@,publics = %s,@,

```

```

117     refines = %s,@,mains = %s@]@,}@]"
118     (pp_str_list (pp_member_def (depth+1)) sections.privates depth)
119     (pp_str_list (pp_member_def (depth+1)) sections.protects depth)
120     (pp_str_list (pp_member_def (depth+1)) sections.publics depth)
121     (pp_str_list (pp_func_def (depth+1)) sections.refines depth)
122     (pp_str_list (pp_func_def (depth+1)) sections.mains depth)
123
124 let pp_class_def the_klass =
125   Format.sprintf "@[<v>@,{@[<v 2>@,klass = %s,@,parent = %s,@,sections = %s@]@,}@]"
126   the_klass.klass
127   (pp_opt _id the_klass.parent)
128   (pp_class_sections the_klass.sections 3)

```

Source 74: "Pretty.ml"

```

1  (** A global UID generator *)
2
3  (** The number of digits in a UID [error after rollover] *)
4  let uid_digits = 8
5
6  (**
7   A function to return the a fresh UID. Note that UIDs are copies,
8   so they need not be copied on their own
9  *)
10 let uid_counter =
11   let counter = String.make uid_digits '0' in
12   let inc () =
13     let i = ref (uid_digits - 1) in
14     while (!i >= 0) && (String.get counter (!i) = 'z') do
15       String.set counter (!i) '0' ;
16       i := !i - 1
17     done ;
18     String.set counter (!i) (match String.get counter (!i) with
19      | '9' -> 'A'
20      | 'Z' -> 'a'
21      | c -> char_of_int (int_of_char c + 1));
22     String.copy counter in
23   inc

```

Source 75: "UID.ml"

```

1
2  if [ "${#@}" -eq 0 ] ; then
3    # Read from stdin when there are no arguments (runtool)
4    cat
5    exit 0
6  fi
7
8  dir="$1"
9  file="$2"
10 shift 2
11
12 type="Brace"
13 if [ "${#@}" -ne 0 ] ; then
14   case "$1" in
15     -b) type="Brace"
16         ;;
17     -s) type="Space"
18         ;;
19     -ml) type="Mixed1"
20         ;;

```

```

21 *) echo "Unknown meta-directory $1" >&2
22     exit 1
23     ;;
24 esac
25 fi
26
27 cat "test/tests/${type}/${dir}/${file}"

```

Source 76: "tools/show-example"

```

1
2 program="$( basename "$0" )"
3 if [ ${#@} -lt 3 ] ; then
4     echo "Usage: $program dir file tool [-s|-b|-m1]" >&2
5     exit 1
6 fi
7
8 dir="$1"
9 file="$2"
10 tool="$3"
11 shift 3
12
13 type="Brace"
14 if [ ${#@} -ne 0 ] ; then
15     case "$1" in
16         -b) type="Brace"
17             ;;
18         -s) type="Space"
19             ;;
20         -m1) type="Mixed1"
21             ;;
22         *) echo "Unknown meta-directory $1" >&2
23             exit 1
24             ;;
25     esac
26 fi
27
28 tool="$( basename "$tool" )"
29 if [ ! -e "tools/${tool}" ] ; then
30     echo "Cannot find tool '${tool}' to execute." >&2
31     exit 1
32 fi
33
34 test -e "tools/${tool}"
35 cat "test/tests/${type}/${dir}/${file}" | "tools/${tool}" "$@"

```

Source 77: "tools/runtool"

```

1 open Ast
2 open Sast
3 open Cast
4 open Klass
5 open StringModules
6 open GlobalData
7
8 let to_fname fuid fname = Format.sprintf "f-%s-%s" fuid fname
9 let to_aname fuid fname = Format.sprintf "a-%s-%s" fuid fname
10 let to_rname fuid fhost fname = Format.sprintf "f-%s-%s-%s" fuid fhost fname
11 let to_dispatch fuid fhost fname = Format.sprintf "d-%s-%s-%s" fuid fhost fname
12
13 let get_fname ( f : Sast.func_def ) = to_fname f.uid f.name

```

```

14 let get_rname (f : Sast.func_def) = match f.host with
15 | None -> raise(Failure("Generating refine name for non-refinement " ^ f.name ^ " in
    class " ^ f.inclass ^ "."))
16 | Some(host) -> to_rname f.uid host f.name
17 let get_vname vname = "v_" ^ vname
18 let get_pointer typ = ("t_" ^ (Str.global_replace (Str.regexp "\\[\\]") "*" typ));;
19
20 let get_tname tname =
21   let fixtypes str = try
22     let splitter n = (String.sub str 0 n, String.sub str n (String.length str - n))
23     in
24     let (before, after) = splitter (String.index str '*') in (String.trim before) ^ "
    " ^ (String.trim after)
25   with Not_found -> str ^ " " in
26   fixtypes (get_pointer tname)
27
28 let from_tname tname = String.sub tname 2 (String.length tname - 3)
29 let opt_tname = function
30 | None -> None
31 | Some(atype) -> Some(get_tname atype)
32 let get_vdef (vtype, vname) = (get_tname vtype, get_vname vname)
33
34 let cast_switch meth refine =
35   let update_klass klass = get_tname klass in
36   let update_dispatch (klass, uid) = (get_tname klass, to_rname uid meth refine) in
37   let update_test klass = get_tname klass in
38   function
39   | Switch(klass, cases, uid) -> Switch(update_klass klass, List.map
    update_dispatch cases, to_dispatch uid meth refine)
40   | Test(klass, classes, uid) -> Test(update_klass klass, List.map update_test
    classes, to_dispatch uid meth refine)
41
42 (*Convert the sast expr to cast expr*)
43 let rec sast_to_castexpr mname env (typetag, sastexpr) = (get_tname typetag,
    c_expr_detail mname sastexpr env)
44 and sast_to_castexprlist mname env explist = List.map (sast_to_castexpr mname env)
    explist
45
46 (* Convert the sast expr_detail to cast_expr detail; convert names / types / etc *)
47 and c_expr_detail mname sastexp env = match sastexp with
48 | Sast.This -> Cast.This
49 | Sast.Null -> Cast.Null
50 | Sast.Id(vname) -> Cast.Id(get_vname vname, snd (
    StringMap.find vname env))
51 | Sast.NewObj(klass, args, BuiltIn(fuid)) -> Cast.NewObj(klass, fuid,
    sast_to_castexprlist mname env args)
52 | Sast.NewObj(klass, args, FuncId(fuid)) -> Cast.NewObj(klass, to_fname fuid "
    init", sast_to_castexprlist mname env args)
53 | Sast.NewObj(klass, args, ArrayAlloc(fuid)) -> Cast.NewArr(get_tname klass,
    to_aname fuid "array_alloc", sast_to_castexprlist mname env args)
54 | Sast.Literal(lit) -> Cast.Literal(lit)
55 | Sast.Assign(e1, e2) -> Cast.Assign(sast_to_castexpr mname
    env e1, sast_to_castexpr mname env e2)
56 | Sast.Deref(e1, e2) -> Cast.Deref(sast_to_castexpr mname
    env e1, sast_to_castexpr mname env e2)
57 | Sast.Field(e1, field) -> Cast.Field(sast_to_castexpr mname
    env e1, get_vname field)
58 | Sast.Invoc(recv, fname, args, BuiltIn(fuid)) -> Cast.Invoc(sast_to_castexpr mname
    env recv, fuid, sast_to_castexprlist mname env args)
59 | Sast.Invoc(recv, fname, args, FuncId(fuid)) -> Cast.Invoc(sast_to_castexpr mname
    env recv, to_fname fuid fname, sast_to_castexprlist mname env args)
60 | Sast.Invoc(_, -, -, ArrayAlloc(_)) -> raise(Failure "Cannot allocate an
    array in an invocation, that is nonsensical.")
61 | Sast.Unop(op, expr) -> Cast.Unop(op, sast_to_castexpr

```

```

62   mname env expr)
63   | Sast.Binop(e1, op, e2)                -> Cast.Binop(sast_to_castexpr mname
env e1, op, sast_to_castexpr mname env e2)
64   | Sast.Refine(name, args, rtype, switch) -> Cast.Refine(sast_to_castexprlist
mname env args, opt_tname rtype, cast-switch mname name switch)
65   | Sast.Refinable(name, switch)          -> Cast.Refinable(cast-switch mname
name switch)
66   | Anonymous(-, -, -)                    -> raise(Failure("Anonymous objects
should have been deanonymized."))
67
68   (*Convert the statement list by invoking cstmt on each of the sast stmt*)
69   let rec cstmtlist mname slist = List.map (cstmt mname) slist
70
71   (* Prepend suffixes *)
72   and cdef vdef = get_vdef vdef
73
74   (*convert sast statement to c statements*)
75   and cstmt mname sstmt =
76   let getoptexpr env = function
77   | Some exp -> Some(sast_to_castexpr mname env exp)
78   | None     -> None in
79
80   let rec getiflist env = function
81   | []                -> []
82   | [(optexpr, slist)] -> [(getoptexpr env optexpr, cstmtlist mname slist)]
83   | (optexpr, slist)::tl -> (getoptexpr env optexpr, cstmtlist mname slist)::(
getiflist env tl) in
84
85   let getsuper args fuid parent env =
86   let init = if BuiltIns.is_built_in parent then fuid else to_fname fuid "init" in
87   let cargs = sast_to_castexprlist mname env args in
88   Cast.Super(parent, init, cargs) in
89
90   match sstmt with
91   | Sast.Decl(var-def, optexpr, env) -> Cast.Decl(cdef var-def, getoptexpr env
optexpr, env)
92   | Sast.If(iflist, env)             -> Cast.If(getiflist env iflist, env)
93   | Sast.While(expr, sstmtlist, env) -> Cast.While(sast_to_castexpr mname env
expr, cstmtlist mname sstmtlist, env)
94   | Sast.Expr(exp, env)              -> Cast.Expr(sast_to_castexpr mname env
exp, env)
95   | Sast.Return(optexpr, env)        -> Cast.Return(getoptexpr env optexpr,
env)
96   | Sast.Super(args, fuid, parent, env) -> getsuper args fuid parent env
97
98   (**
99   Trim up the sast func-def to the cast cfunc-def
100  @param func It's a sast func-def. Woo.
101  @return It's a cast cfunc-def. Woo.
102  *)
103  let sast_to_cast_func (func : Sast.func_def) : cfunc =
104  let name = match func.host, func.builtin with
105  | -, true -> func.uid
106  | None, _ -> get_fname func
107  | Some(host), _ -> get_rname func in
108  {
109    returns = opt_tname func.returns;
110    name = name;
111    formals = List.map get_vdef func.formals;
112    body = cstmtlist func.name func.body;
113    builtin = func.builtin;
114    inclass = func.inclass;
115    static = func.static;
116  }
117
118  let build_class_struct_map klass_data (sast-classes : Sast.class_def list) =

```

```

117 (* Extract the ancestry and variables from a class into a cdef *)
118 let klass_to_struct klass_name (aklass : Ast.class_def) =
119   let compare (_, n1) (_, n2) = Pervasives.compare n1 n2 in
120   let ivars = List.flatten (List.map snd (Klass.klass_to_variables aklass)) in
121   let renamed = List.map get_vdef ivars in
122   [(klass_name, List.sort compare renamed)] in
123
124 (* Map each individual class to a basic class_struct *)
125 let struct_map = StringMap.mapi klass_to_struct klass_data.classes in
126
127 (* Now, assuming we get parents before children, update the maps appropriately *)
128 let folder map = function
129   | "Object" -> StringMap.add (get_tname "Object") (StringMap.find "Object"
130 struct_map) map
131   | aklass ->
132     let parent = StringMap.find aklass klass_data.parents in
133     let ancestors = StringMap.find (get_tname parent) map in
134     let this = StringMap.find aklass struct_map in
135     StringMap.add (get_tname aklass) (this @ ancestors) map in
136
137 (* Update the map so that each child has information from parents *)
138 let struct_map = List.fold_left folder StringMap.empty (Klass.get_class_names
139 klass_data) in
140
141 (* Reverse the values so that they start from the root *)
142 StringMap.map List.rev struct_map
143
144 let sast_functions (klasses : Sast.class_def list) =
145   (* Map a Sast class to its functions *)
146   let get_functions (klass : Sast.class_def) =
147     let s = klass.sections in
148     let funcs = function
149       | Sast.MethodMem(m) -> Some(m)
150       | Sast.InitMem(i) -> Some(i)
151       | _ -> None in
152     let get_funcs mems = Util.filter_option (List.map funcs mems) in
153     List.flatten [ get_funcs s.publics ; get_funcs s.protects ; get_funcs s.privates
154 ; s.refines ; s.mains ] in
155
156   let all_functions = List.flatten (List.map get_functions classes) in
157   let all_mains = List.flatten (List.map (fun k -> k.sections.mains) classes) in
158
159   (all_functions, all_mains)
160
161 let leaf_ancestors klass_data =
162   let leaves = get_leaves klass_data in
163   let mangled l = List.map get_tname (map_lookup_list l klass_data.ancestors) in
164   let ancestors l = (l, List.rev (mangled l)) in
165   List.map ancestors leaves
166
167 let sast_to_cast klass_data (klasses : Sast.class_def list) : Cast.program =
168   let (funcs, mains) = sast_functions classes in
169   let main_case (f : Sast.func_def) = (f.inklass, get_fname f) in
170   let cfuncs = List.map sast_to_cast_func funcs in
171   let main_switch = List.map main_case mains in
172   let struct_map = build_class_struct_map klass_data classes in
173   let ancestor_data = klass_data.ancestors in
174
175   (struct_map, cfuncs, main_switch, StringMap.map List.rev ancestor_data)
176
177 let built_in_names =
178   let klass_names = List.map (fun (f : Ast.class_def) -> get_tname f.klass) BuiltIns.
179   built_in_classes in
180   List.fold_left (fun set i -> StringSet.add i set) StringSet.empty klass_names

```

Source 78: "GenCast.ml"

```

1 open Util
2
3 val klass_to_parent : Ast.class_def -> string
4 val section_string : Ast.class_section -> string
5 val klass_to_variables : Ast.class_def -> (Ast.class_section * Ast.var_def list) list
6 val klass_to_methods : Ast.class_def -> (Ast.class_section * Ast.func_def list) list
7 val klass_to_functions : Ast.class_def -> (Ast.class_section * Ast.func_def list) list
8 val conflicting_signatures : Ast.func_def -> Ast.func_def -> bool
9 val signature_string : Ast.func_def -> string
10 val full_signature_string : Ast.func_def -> string
11 val class_var_lookup : GlobalData.class_data -> string -> string -> (Ast.class_section *
    string) option
12 val class_field_lookup : GlobalData.class_data -> string -> string -> (string * string *
    Ast.class_section) option
13 val class_field_far_lookup : GlobalData.class_data -> string -> string -> bool -> ((
    string * string * Ast.class_section), bool) either
14 val class_method_lookup : GlobalData.class_data -> string -> string -> Ast.func_def list
15 val class_ancestor_method_lookup : GlobalData.class_data -> string -> string -> bool ->
    Ast.func_def list
16 val refine_lookup : GlobalData.class_data -> string -> string -> string -> Ast.func_def
    list
17 val refinable_lookup : GlobalData.class_data -> string -> string -> string -> Ast.
    func_def list
18 val get_distance : GlobalData.class_data -> string -> string -> int option
19 val is_type : GlobalData.class_data -> string -> bool
20 val is_subtype : GlobalData.class_data -> string -> string -> bool
21 val is_proper_subtype : GlobalData.class_data -> string -> string -> bool
22 val compatible_formals : GlobalData.class_data -> string list -> string list -> bool
23 val compatible_function : GlobalData.class_data -> string list -> Ast.func_def -> bool
24 val compatible_return : GlobalData.class_data -> string option -> Ast.func_def -> bool
25 val compatible_signature : GlobalData.class_data -> string option -> string list -> Ast.
    func_def -> bool
26 val best_matching_signature : GlobalData.class_data -> string list -> Ast.func_def list
    -> Ast.func_def list
27 val best_method : GlobalData.class_data -> string -> string -> string list -> Ast.
    class_section list -> Ast.func_def option
28 val best_inherited_method : GlobalData.class_data -> string -> string -> string list ->
    bool -> Ast.func_def option
29 val refine_on : GlobalData.class_data -> string -> string -> string -> string list ->
    string option -> Ast.func_def list
30 val get_class_names : GlobalData.class_data -> string list
31 val get_leaves : GlobalData.class_data -> string list

```

Source 79: "Klass.mli"

```

1 open Ast
2 open Str
3
4 (** Built in classes *)
5
6 let built_in cname : Ast.func_def = match Str.split (regexp "-") cname with
7   | [] -> raise (Failure "Bad cname — empty.")
8   | [klass] -> raise (Failure ("Bad cname — just class: " ^ klass))
9   | klass::func ->
10     let methname = match func with
11     | [] -> raise (Failure ("Impossible!"))
12     | func::rest -> func ^ (String.concat "" (List.map String.capitalize rest))

```



```

13   in
14     { returns = None;
15       host = None;
16       name = methname;
17       static = false;
18       formals = [];
19       body = [];
20       section = Publics;
21       inclass = String.capitalize klass;
22       uid = cname;
23       builtin = true }
24 let breturns cname atype = { (builtin cname) with returns = Some(atype) }
25 let btakes cname formals = { (builtin cname) with formals = formals }
26
27 let sections : Ast.class_sections_def =
28   { publics = [];
29     protects = [];
30     privates = [];
31     refines = [];
32     mains = [] }
33
34 let func f = if f.name = "init" then InitMem(f) else MethodMem(f)
35 let var v = VarMem(v)
36 let variables = List.map var
37 let functions = List.map func
38 let members f v = (functions f) @ (variables v)
39
40 let class_object : Ast.class_def =
41   let name = "Object" in
42
43     let init_obj : Ast.func_def = { (builtin "object_init") with section = Protects } in
44     let system = ("System", "system") in
45
46     let sections : Ast.class_sections_def =
47       { sections with
48         publics = [];
49         protects = [func init_obj; var system] } in
50
51     { klass = name; parent = None; sections = sections }
52
53 let class_scanner : Ast.class_def =
54   let name = "Scanner" in
55
56     let scan_line : Ast.func_def = breturns "scanner_scan_string" "String" in
57     let scan_int : Ast.func_def = breturns "scanner_scan_integer" "Integer" in
58     let scan_float : Ast.func_def = breturns "scanner_scan_float" "Float" in
59     let scan_init : Ast.func_def = builtin "scanner_init" in
60
61     let sections : Ast.class_sections_def =
62       { sections with
63         publics = functions [scan_line; scan_int; scan_float; scan_init] } in
64
65     { klass = name; parent = None; sections = sections }
66
67 let class_printer : Ast.class_def =
68   let name = "Printer" in
69
70     let print_string : Ast.func_def = btakes "printer_print_string" [("String", "arg")] in
71     let print_int : Ast.func_def = btakes "printer_print_integer" [("Integer", "arg")] in
72     let print_float : Ast.func_def = btakes "printer_print_float" [("Float", "arg")] in
73     let print_init : Ast.func_def = btakes "printer_init" [("Boolean", "stdout")] in
74
75     let sections : Ast.class_sections_def =
76       { sections with

```

```

76         publics = functions [print_string; print_int; print_float; print_init] } in
77
78     { klass = name; parent = None; sections = sections }
79
80 let class_string : Ast.class_def =
81     let name = "String" in
82
83     let string_init : Ast.func_def = built_in "string_init" in
84
85     let sections : Ast.class_sections_def =
86         { sections with
87             protects = [func string_init] } in
88
89     { klass = name; parent = None; sections = sections }
90
91
92 let class_boolean : Ast.class_def =
93     let name = "Boolean" in
94
95     let boolean_init : Ast.func_def = built_in "boolean_init" in
96
97     let sections : Ast.class_sections_def =
98         { sections with
99             protects = [func boolean_init] } in
100
101     { klass = name; parent = None; sections = sections }
102
103 let class_integer : Ast.class_def =
104     let name = "Integer" in
105
106     let integer_init : Ast.func_def = built_in "integer_init" in
107     let integer_float : Ast.func_def = breturns "integer_to_f" "Float" in
108
109     let sections : Ast.class_sections_def =
110         { sections with
111             publics = [func integer_float];
112             protects = [func integer_init] } in
113
114     { klass = name; parent = None; sections = sections }
115
116 let class_float : Ast.class_def =
117     let name = "Float" in
118
119     let float_init : Ast.func_def = built_in "float_init" in
120     let float_integer : Ast.func_def = breturns "float_to_i" "Integer" in
121
122     let sections : Ast.class_sections_def =
123         { sections with
124             publics = [func float_integer];
125             protects = [func float_init] } in
126
127     { klass = name; parent = None; sections = sections }
128
129 let class_system : Ast.class_def =
130     let name = "System" in
131
132     let system_init : Ast.func_def = built_in "system_init" in
133     let system_exit : Ast.func_def = btakes "system_exit" [("Integer", "code")] in
134
135     let system_out = ("Printer", "out") in
136     let system_err = ("Printer", "err") in
137     let system_in = ("Scanner", "in") in
138     let system_argc = ("Integer", "argc") in
139
140     let sections : Ast.class_sections_def =

```

```

141     { sections with
142       publics = members [system_init; system_exit] [system_out; system_err; system_in
; system_argc]; } in
143
144     { klass = name; parent = None; sections = sections }
145
146 (** The list of built in classes and their methods *)
147 let built_in_classes =
148   [ class_object; class_string; class_boolean; class_integer; class_float; class_printer;
    class_scanner; class_system ]
149
150 (** Return whether a class is built in or not *)
151 let is_built_in name =
152   List.exists (fun klass -> klass.class = name) built_in_classes

```

Source 80: "BuiltIns.ml"

```

1  open Ast
2  open Util
3  open StringModules
4
5  (** Module for getting sets of variables *)
6
7  (** Get the formal variables of a function *)
8  let formal_vars func =
9    let add_param set (_, v) = StringSet.add v set in
10   List.fold_left add_param StringSet.empty func.formals
11
12  (** Get the free variables of a list of statements *)
13  let free_vars bound stmts =
14    let rec get_free_vars free = function
15      | [] -> free
16      | (bound, Left(stmts))::todo -> get_free_stmts free bound todo stmts
17      | (bound, Right(exprs))::todo -> get_free_exprs free bound todo exprs
18    and get_free_stmts free bound todo = function
19      | [] -> get_free_vars free todo
20      | stmt::rest ->
21        let (expr_block_list, stmt_block_list, decl) = match stmt with
22          | Decl((_, var), e) -> ([option_as_list e], [], Some(var))
23          | Expr(e)           -> ([e], [], None)
24          | Return(e)         -> ([option_as_list e], [], None)
25          | Super(es)         -> ([es], [], None)
26          | While(e, body)    -> ([e], [body], None)
27          | If(parts)         -> let (es, ts) = List.split parts in
                                ([filter_option es], ts
28                                , None) in
29        let expressions = List.map (function exprs -> (bound, Right(exprs)))
    expr_block_list in
30        let statements  = List.map (function stmts -> (bound, Left(stmts)))
    stmt_block_list in
31        let bound = match decl with
32          | Some(var) -> StringSet.add var bound
33          | _ -> bound in
34        get_free_stmts free bound (expressions @ statements @ todo) rest
35    and get_free_exprs free bound todo = function
36      | [] -> get_free_vars free todo
37      | expr::rest ->
38        let func_to_task bound func =
39          (StringSet.union (formal_vars func) bound, Left(func.body)) in
40
41        let (exprs, tasks, id) = match expr with
42          | NewObj(_, args) -> (args, [], None)
43          | Assign(l, r)    -> ([l; r], [], None)

```

```

44 | Deref(v, i)          -> ([v; i], [], None)
45 | Field(e, _)         -> ([e], [], None)
46 | Invoc(e, _, args)   -> (e::args, [], None)
47 | Unop(_, e)          -> ([e], [], None)
48 | Binop(l, _, r)       -> ([l; r], [], None)
49 | Refine(_, args, _)   -> (args, [], None)
50 | This                -> ([], [], None)
51 | Null                -> ([], [], None)
52 | Refinable(_)         -> ([], [], None)
53 | Literal(_)           -> ([], [], None)
54 | Id(id)               -> ([], [], decide_option id (not (StringSet.
mem id bound)))
55 | Anonymous(_, args, funcs) -> (args, List.map (func_to_task bound) funcs
, None) in
56
57     let rest = exprs @ rest in
58     let todo = tasks @ todo in
59     let free = match id with
60     | Some(id) -> StringSet.add id free
61     | None -> free in
62     get_free_exprs free bound todo rest in
63
64     get_free_vars StringSet.empty [(bound, Left(stmts))]
65
66 (** Get the free variables in a function. *)
67 let free_vars_func bound func =
68   let params = formal_vars func in
69   free_vars (StringSet.union bound params) func.body
70
71 (** Get the free variables in a whole list of functions. *)
72 let free_vars_funcs bound funcs =
73   let sets = List.map (free_vars_func bound) funcs in
74   List.fold_left StringSet.union StringSet.empty sets

```

Source 81: "Variables.ml"

```

1 gcc -g -I ../headers -lm -o a.out test.c

```

Source 82: "ctest/compile"

```

1 open Util
2
3 let show_classes builder classes = match builder classes with
4 | Left(data) -> KlassData.print_class_data data; exit(0)
5 | Right(issue) -> Printf.fprintf stderr "%s\n" (KlassData.errstr issue); exit(1)
6
7 let from_input builder =
8   let tokens = Inspector.from_channel stdin in
9   let classes = Parser.cdecls (WhiteSpace.lextoks tokens) (Lexing.from_string "") in
10  show_classes builder classes
11 let from_basic builder = show_classes builder []
12
13 let basic_info_test () = from_basic KlassData.build_class_data_test
14 let basic_info () = from_basic KlassData.build_class_data
15
16 let test_info () = from_input KlassData.build_class_data_test
17 let normal_info () = from_input KlassData.build_class_data
18
19 let exec name func = Printf.printf "Executing mode %s\n" name; flush stdout; func ()
20
21 let _ = try

```

```

22     Printexc.record_backtrace true;
23     match Array.to_list Sys.argv with
24     | []      -> raise (Failure("Not even program name given as argument."))
25     | [-]    -> exec "Normal Info" normal_info
26     | _::arg::_ -> match arg with
27     | "_"      -> exec "Basic Info" basic_info
28     | "___"    -> exec "Basic Test" basic_info_test
29     | _        -> exec "Test Info" test_info
30 with _ ->
31     Printexc.print_backtrace stderr

```

Source 83: "classinfo.ml"

```

1  #!/bin/bash
2
3  testdir="$( dirname "$0" )"
4  testprogram=".testdrive"
5
6  "$testdir/$testprogram" "$0" "inspect" "expect-parser" "$@"

```

Source 84: "test/parser"

```

1  test types:
2  * Brace   — these should be with {, }, and ;
3  * Mixed1  — these should be mixed (closer to Space for now)
4  * Space   — these should be with :
5
6  in each type there are test folders:
7  * Empty   — structurally empty tests
8  * Trivial — just above empty, should do something... trivial
9  * Simple  — some basic programs, more than just trivial
10
11 each test type requires the same tests. at the end, the outputs are compared

```

Source 85: "test/README"

```

1  #!/bin/bash
2
3  program="$( basename "$1" )"
4  scriptdir="$( dirname "$1" )"
5  exe=".tools/$2"
6  old="$3"
7  shift 3
8
9  # Arguments
10 justrun=
11 save=
12 verbose=
13 pattern=*
14 folderpattern=*
15
16 # Calculated values change in each iteration
17 current=
18 results=
19
20 # Don't change per iteration
21 tmpfile="test/check"
22 tmperr="test/err"

```

```

23 testdir="test/tests"
24 maxlength=0
25 oneline=0
26 files=()
27 folders=()
28 temp=()
29 errored=0
30 dropadj=1
31
32 # Formatting values
33 bold='tput bold '
34 normal='tput sgr0 '
35 uline='tput smul '
36 green='tput setaf 2 '
37 red='tput setaf 1 '
38 blue='tput setaf 4 '
39 backblue='tput setab 4 '
40
41 function errWith {
42     echo "$1" >&2
43     exit 1
44 }
45
46 function execerror {
47     echo "${bold}${uline}${red}ERROR${normal} $1"
48     errored=1
49 }
50
51 function dots {
52     local len='echo "$current" | wc -c '
53     for i in `seq $len $maxlength` ; do
54         echo -n '.'
55     done
56     echo -n ' '
57 }
58
59 function contains {
60     local elem
61     for elem in "${@:2}" ; do
62         test "$elem" = "$1" && return 0
63     done
64     return 1
65 }
66
67 function dropdirprefix {
68     echo "$1" | cut -c $(( ${#2} + $dropadj ))-
69 }
70
71 function setdropadj {
72     local result=$( dropdirprefix "/dev/null" "/dev/" )
73     local null="null"
74     dropadj=$(( dropadj + (${#null} - ${#result}) )
75 }
76
77 function show_standard {
78     echo "${red}Standard — START${normal}"
79     cat "$results"
80     echo "${red}Standard — END${normal}"
81 }
82
83 function testit {
84     local testing="${bold}Testing:${normal} ${uline}${current}${normal}"
85     test "$oneline" -eq 0 && echo "$testing"
86     test "$oneline" -ne 0 && echo -n "$testing"
87     test "$oneline" -ne 0 && dots

```

```

88 test -n "$verbose" && cat "$1"
89 if [ -n "$justrun" ] ; then
90     cat "$1" | "$exe"
91     return 0
92 fi
93 cat "$1" | "$exe" 1> "$tmpfile" 2> "$tmperr"
94 if [ $? -ne 0 ] ; then
95     execerror "Error testing $program with $current"
96     cat "$tmperr"
97 elif [ -n "$save" ] ; then
98     echo "${bold}Saving${normal} $current"
99     mkdir -p $( dirname "$results" )
100    mv "$tmpfile" "$results"
101 elif [ ! -e "$results" ] ; then
102     execerror "Cannot check results — standard does not exist"
103 else
104     if [ -n "$verbose" ] ; then
105         echo -n "${bold}Output:${normal} "
106         cat "$tmpfile"
107     fi
108     test "$soneline" -eq 0 && echo -n "${bold}Results:${normal} "
109     diff -q "$tmpfile" "$results" &> /dev/null
110     if [ $? -eq 0 ] ; then
111         echo "${bold}${green}PASS${normal}"
112     else
113         echo "${bold}${red}MISMATCH${normal}"
114         test -n "$verbose" && show_standard
115     fi
116 fi
117
118 test -e "$tmpfile" && rm "$tmpfile" # Sometimes happens
119 test -e "$tmperr" && rm "$tmperr" # Always happens
120
121 test "$soneline" -eq 0 && echo ""
122 }
123
124 function listandexit {
125     for afile in $( find "$testdir" -type f -name "$pattern" ) ; do
126         current=$( dropdirprefix "$afile" "$testdir" )
127         echo "$current"
128     done
129     exit 0
130 }
131
132 function usage {
133     cat <<USAGE
134     $program -[chlpvs]
135     -f pattern
136         Filter meta-folders by pattern
137
138     -h
139         Display this help
140
141     -l
142         Display the name of all tests; note that pattern can be used
143
144     -p pattern
145         Filter tests to be used based on pattern (as in find -name)
146
147     -R
148         merely run the driving exe and output the result to stdout (no checking anything)
149
150     -s
151         save results
152

```

```

153     -v
154     verbose output
155 USAGE
156     exit 0
157 }
158
159 setdropadj
160
161 while getopts "f:hlRsvp:" OPTION ; do
162     case "$OPTION" in
163         f) folderpattern=$OPTARG ;;
164         h) usage ;;
165         R) justrun=1 ;;
166         s) save=1 ;;
167         v) verbose=1 ;;
168         p) pattern=$OPTARG ;;
169         l) list=1;;
170         ?) errWith "Unknown option; aborting" ;;
171     esac
172 done
173 shift $((OPTIND - 1))
174
175 test -n "$list" && listandexit
176
177 test -e "$exe" || errWith "Testing $program but $exe unavailable"
178 test -f "$exe" || errWith "Testing $program but $exe is not a file"
179 test -x "$exe" || errWith "Testing $program but $exe unexecutable"
180
181 test -z "$verbose" && oneline=1
182
183 for adir in $( find "$testdir" -mindepth 1 -maxdepth 1 -type d -name "$folderpattern" ) ;
184 do
185     adir=$( dropdirprefix "$adir" "$testdir/" )
186     folders+=( "$adir" )
187 done
188 test "${#folders[@]}" -eq 0 && errWith "No folders in test directory. Good-bye."
189
190 for afolder in "${folders[@]}" ; do
191     test -d "$testdir/$afolder" || errWith "$afolder is not a directory ($testdir)"
192 done
193
194 for afile in $( find "$testdir/${folders[0]}" -type f -name "$pattern" ) ; do
195     test "README" = $( basename "$afile" ) || files+=( $( dropdirprefix "$afile" "$testdir/"
196     "${folders[0]}/" ) )
197 done
198
199 for afolder in "${folders[@]}" ; do
200     temp=()
201     for afile in $( find "$testdir/$afolder" -type f -name "$pattern" ) ; do
202         test "README" = $( basename "$afile" ) || temp+=( $( dropdirprefix "$afile" "$testdir/"
203         "$afolder/" ) )
204     done
205
206     for afile in "${files[@]}" ; do
207         contains "$afile" "${temp[@]}" || errWith "$afolder does not contain $afile but ${
208         folders[0]} does"
209     done
210
211     for bfile in "${temp[@]}" ; do
212         contains "$bfile" "${files[@]}" || errWith "$afolder contains $bfile but ${
213         folders[0]} does not"
214     done
215 done
216
217 test "${#files[@]}" -eq 0 && errWith "No files match the given pattern. Good-bye."
218
219 # All the test directories have the same structure.

```



```

213 for current in "${files[@]}" ; do
214     len='echo "$current" | wc -c'
215     test $len -gt $maxlength && maxlength="$len"
216 done
217 maxlength=$(( maxlength + 5 ))
218
219 for afolder in "${folders[@]}" ; do
220     echo "${bold}${blue}Testing:${normal} $afolder"
221     for current in "${files[@]}" ; do
222         results="test/$old/$afolder/$current"
223         testit "$testdir/$afolder/$current"
224     done
225 done
226
227 test $errored -eq 1 && exit 1
228 test -n "$justrun" && exit 0
229
230 # Ensure that all the results are the same.
231 for current in "${files[@]}" ; do
232     master="test/$old/${folders[0]}/$current"
233     matched=1
234
235     for afolder in "${folders[@]}" ; do
236         target="test/$old/$afolder/$current"
237         diff -q "$master" "$target" &> /dev/null
238         if [ $? -ne 0 ] ; then
239             echo "$current ${bold}${red}DIFFERS${normal} between ${folders[0]} (reference) and
240                 $afolder"
241             matched=0
242         fi
243     done
244     test $matched -eq 1 && echo "$current ${bold}${green}MATCHES${normal} across all
245     folders"
246 done

```

Source 86: "test/.testdrive"

```

1 #!/bin/bash
2
3 testdir="$(dirname "$0")"
4 testprogram=".testdrive"
5
6 "$testdir/$testprogram" "$0" "pretty" "expect-ast-pretty" "$@"

```

Source 87: "test/ast-pretty"

```

1 #!/bin/bash
2
3 testdir="$(dirname "$0")"
4 testprogram=".testdrive"
5
6 "$testdir/$testprogram" "$0" "streams" "expect-scanner" "$@"

```

Source 88: "test/scanner"

```

1 class List {
2 }

```

Source 89: "test/tests/Brace/Empty/Class"

```
1 class List {  
2     public {  
3         init() {  
4             }  
5         void noop() {  
6             }  
7     }  
8 }
```

Source 90: "test/tests/Brace/Empty/InitMethod"

```
1 class List {  
2     refinement {  
3     }  
4 }
```

Source 91: "test/tests/Brace/Empty/Refinements"

```
1 class List {  
2     public {  
3         void noop() {  
4             }  
5     }  
6 }
```

Source 92: "test/tests/Brace/Empty/Method"

```
1 class List {  
2     private {  
3     }  
4 }
```

Source 93: "test/tests/Brace/Empty/Private"

```
1 class List {  
2     public {  
3         void noop() {  
4             while(true) {  
5             }  
6         }  
7     }  
8 }
```

Source 94: "test/tests/Brace/Empty/WhileMethod"

```
1 class List {  
2     public {  
3         init() {
```

```

4   }
5   }
6   }

```

Source 95: "test/tests/Brace/Empty/Init"

```

1   class List {
2       public {
3       }
4   }

```

Source 96: "test/tests/Brace/Empty/Public"

```

1   class List {
2       protected {
3       }
4   }

```

Source 97: "test/tests/Brace/Empty/Protected"

```

1   class List {
2       public {
3           void noop() {
4               if(true) {
5               }
6           }
7       }
8   }

```

Source 98: "test/tests/Brace/Empty/IfMethod"

```

1   class Collection {
2       protected {
3           init() {
4           }
5       }
6
7       public {
8           Boolean mutable() {
9               return refine answer() to Boolean;
10          }
11
12          void add(Object item) {
13              refine do(item) to void;
14          }
15
16          void addAll(Collection other) {
17              if(refinable(do)) {
18                  refine combine(other) to void;
19              } else {
20                  Iterator items := other.iterator();
21                  while(not items.done()) {
22                      add(items.next());
23                  }
24              }
25          }
26      }
27  }

```

```

26
27     void clear() {
28         refine do() to void;
29     }
30
31     Boolean contains(Object item) {
32         if(refinable(check)) {
33             return refine check(item) to Boolean;
34         }
35
36         Iterator items := this.iterator();
37         while(not items.done()) {
38             if(items.next() = item) {
39                 return true;
40             }
41         }
42         return false;
43     }
44
45     Boolean containsAll(Collection other) {
46         if(refinable(check)) {
47             return refine check(other) to Boolean;
48         }
49
50         Iterator items := other.iterator();
51         while(not items.done()) {
52             if(not this.contains(items.next())) {
53                 return false;
54             }
55         }
56         return true;
57     }
58 }
59 }

```

Source 99: "test/tests/Brace/Multi/Collection"

```

1  class List extends Node {
2      public {
3          init() {
4              Int c;
5              c := 1234;
6          }
7      }
8  }

```

Source 100: "test/tests/Brace/Trivial/InitStatement"

```

1  class List extends Node {
2      main {
3          List l = new List();
4          Int mac = l.macguffin();
5      }
6      private {
7          init() {
8              }
9      }
10     public {
11         Int macguffin() {
12             return 4;
13         }
14     }
15 }

```

```
14 }  
15 }
```

Source 101: "test/tests/Brace/Trivial/MainWithBuilding"

```
1 class Rectangle extends Shape {  
2   public {  
3     init(Int width, Int height) {  
4       this.width := width;  
5       this.height := height;  
6     }  
7     Int area() {  
8       return width * height;  
9     }  
10    Int perimeter() {  
11      return 2 * (width + height);  
12    }  
13  }  
14  protected {  
15    Int width;  
16    Int height;  
17  }  
18 }
```

Source 102: "test/tests/Brace/Simple/Rectangle"

```
1 class List:
```

Source 103: "test/tests/Mixed1/Empty/Class"

```
1 class List:  
2   public:  
3     init():  
4     void noop() {  
5     }
```

Source 104: "test/tests/Mixed1/Empty/InitMethod"

```
1 class List:  
2   refinement {  
3   }
```

Source 105: "test/tests/Mixed1/Empty/Refinements"

```
1 class List:  
2   public:  
3     void noop() {  
4     }
```

Source 106: "test/tests/Mixed1/Empty/Method"

```
1 class List:
```

```

2 private {
3 }

```

Source 107: "test/tests/Mixed1/Empty/Private"

```

1 class List:
2   public:
3     void noop():
4       while(true){
5
6     }

```

Source 108: "test/tests/Mixed1/Empty/WhileMethod"

```

1 class List:
2   public:
3     init() {
4     }

```

Source 109: "test/tests/Mixed1/Empty/Init"

```

1 class List:
2   public {
3   }

```

Source 110: "test/tests/Mixed1/Empty/Public"

```

1 class List:
2   protected {
3   }

```

Source 111: "test/tests/Mixed1/Empty/Protected"

```

1 class List:
2   public:
3     void noop(){
4       if(true){}
5     }

```

Source 112: "test/tests/Mixed1/Empty/IfMethod"

```

1 class Collection:
2   protected:
3     init() {
4     }
5
6   public:
7     Boolean mutable() {
8       return refine answer() to Boolean;
9     }
10
11     void add(Object item):

```

```

12     refine do(item) to void
13
14 void addAll(Collection other):
15     if(refinable(do)) {
16         refine combine(other) to void;
17     } else:
18         Iterator items := other.iterator()
19         while(not items.done()) {
20             add(items.next());
21         }
22
23 void clear():
24     refine do() to void
25
26 Boolean contains(Object item):
27     if(refinable(check)):
28         return refine check(item) to Boolean
29
30     Iterator items := this.iterator()
31     while(not items.done()):
32         if(items.next() = item) {
33             return true;
34         }
35     return false
36
37 Boolean containsAll(Collection other):
38     if(refinable(check)) {
39         return refine check(other) to Boolean;
40     }
41
42     Iterator items := other.iterator()
43     while(not items.done()):
44         if(not this.contains(items.next())):
45             return false
46     return true

```

Source 113: "test/tests/Mixed1/Multi/Collection"

```

1 class List extends Node:
2     public:
3         init() {
4             Int c;
5             c := 1234;
6         }

```

Source 114: "test/tests/Mixed1/Trivial/InitStatement"

```

1 class Rectangle extends Shape:
2     public:
3         init(Int width, Int height) {
4             this.width := width;
5             this.height := height;
6         }
7
8         Int area():
9             return width * height
10
11         Int perimeter():
12             return 2 * (width + height)
13
14     protected {

```

```

15     Int width;
16     Int height;
17 }

```

Source 115: "test/tests/Mixed1/Simple/Rectangle"

```

1 class List:

```

Source 116: "test/tests/Space/Empty/Class"

```

1 class List:
2     public:
3         init():
4         void noop():

```

Source 117: "test/tests/Space/Empty/InitMethod"

```

1 class List:
2     refinement:

```

Source 118: "test/tests/Space/Empty/Refinements"

```

1 class List:
2     public:
3         void noop():

```

Source 119: "test/tests/Space/Empty/Method"

```

1 class List:
2     private:

```

Source 120: "test/tests/Space/Empty/Private"

```

1 class List:
2     public:
3         void noop():
4             while(true):

```

Source 121: "test/tests/Space/Empty/WhileMethod"

```

1 class List:
2     public:
3         init():

```

Source 122: "test/tests/Space/Empty/Init"

```

1 class List:

```



```
2 public:
```

Source 123: "test/tests/Space/Empty/Public"

```
1 class List:
2   protected:
```

Source 124: "test/tests/Space/Empty/Protected"

```
1 class List:
2   public:
3     void noop():
4       if(true):
```

Source 125: "test/tests/Space/Empty/IfMethod"

```
1 class Collection:
2   protected:
3     /* Only subclasses can be created */
4     init():
5
6   public:
7     Boolean mutable():
8       return refine answer() to Boolean
9
10    void add(Object item):
11      refine do(item) to void
12
13    void addAll(Collection other):
14      if (refinable(do)):
15        refine combine(other) to void
16      else:
17        Iterator items := other.iterator()
18        while (not items.done()):
19          add(items.next())
20
21    void clear():
22      refine do() to void
23
24    Boolean contains(Object item):
25      if (refinable(check)):
26        return refine check(item) to Boolean
27
28      Iterator items := this.iterator()
29      while (not items.done()):
30        if (items.next() = item):
31          return true
32      return false
33
34    Boolean containsAll(Collection other):
35      if (refinable(check)):
36        return refine check(other) to Boolean
37
38      Iterator items := other.iterator()
39      while (not items.done()):
40        if (not this.contains(items.next())):
41          return false
42      return true
```

Source 126: "test/tests/Space/Multi/Collection"

```

1 class List extends Node:
2   public:
3     init():
4       Int c;
5       c := 1234;

```

Source 127: "test/tests/Space/Trivial/InitStatement"

```

1 class Rectangle extends Shape:
2   public:
3     init(Int width, Int height):
4       this.width := width
5       this.height := height
6
7     Int area():
8       return width * height
9
10    Int perimeter():
11      return 2 * (width + height)
12
13  protected:
14    Int width
15    Int height

```

Source 128: "test/tests/Space/Simple/Rectangle"

```

1 open StringModules
2 open Sast
3 open Ast
4 open Util
5
6 (** Take a collection of Sast class_defs and deanonymize them. *)
7
8
9 (** The data needed to deanonymize a list of classes and store the results. *)
10 type anon_state = {
11   labeler : int lookup_map ;      (** Label deanonymized classes *)
12   deanon : Ast.class_def list ;   (** List of Ast.class_def classes that are
13     deanonymized. *)
14   clean : Sast.class_def list ;   (** List of clean Sast.class_def classes *)
15   data : GlobalData.class_data ;  (** A class_data record used for typing *)
16   current : string ;              (** The class that is currently being examined *)
17 }
18
19 (**
20  Given the initial anon_state, an environment, and an expr_detail, remove all
21  anonymous object instantiations from the expr and replace them with the
22  instantiation of a newly constructed class. This returns a changed expr_detail
23  value and an updated state — i.e. maybe a new ast class is added to it.
24  @param init_state anon_state value
25  @param env an environment (like those attached to statements in sAST)
26  @param expr_deets an expr_detail to transform
27  @return (new expr_detail, updated state)
28 *)
29 let rec deanon_expr_detail init_state env expr_deets =

```

```

29 let get_label state klass =
30   let (n, labeler) = match map_lookup klass state.labeler with
31     | None -> (0, StringMap.add klass 0 state.labeler)
32     | Some(n) -> (n+1, StringMap.add klass (n+1) state.labeler) in
33   (Format.sprintf "anon-%s-%d" klass n, { state with labeler = labeler }) in
34
35 let get_var_type state env var_name =
36   match map_lookup var_name env with
37   | Some(vinfo) -> Some(fst vinfo)
38   | None -> match Klass.class_field_lookup state.data state.current var_name
39 with
40   | Some((- , vtype, -)) -> Some(vtype)
41   | _ -> None in
42
43 let deanon_init args formals klass : Ast.func_def =
44   let givens = List.map (fun (t, _) -> (t, "Anon_v_" ^ UID.uid_counter ())) args in
45   let all_formals = givens @ formals in
46   let super = Ast.Super(List.map (fun (_, v) -> Ast.Id(v)) givens) in
47   let assigner (_, vname) = Ast.Expr(Ast.Assign(Ast.Field(Ast.This, vname), Ast.Id(
48     vname))) in
49   {
50     returns = None;
51     host = None;
52     name = "init";
53     static = false;
54     formals = all_formals;
55     body = super::(List.map assigner formals);
56     section = Publics;
57     inklass = klass;
58     uid = UID.uid_counter ();
59     builtin = false } in
60
61 let deanon_class args freedefs klass parent refines =
62   let init = deanon_init args freedefs klass in
63   let vars = List.map (fun vdef -> Ast.VarMem(vdef)) freedefs in
64   let sections =
65     {
66       privates = vars;
67       protects = [];
68       publics = [InitMem(init)];
69       refines = List.map (fun r -> { r with inklass=klass }) refines;
70       mains = []; } in
71   let theklass =
72     {
73       klass = klass;
74       parent = Some(parent);
75       sections = sections } in
76   (init.uid, theklass) in
77
78 let deanon_freedefs state env funcs =
79   let freeset = Variables.free_vars_funcs StringSet.empty funcs in
80   let freevars = List.sort compare (StringSet.elements freeset) in
81
82   let none_snd = function
83     | (None, v) -> Some(v)
84     | _ -> None in
85   let some_fst = function
86     | (Some(t), v) -> Some((t, v))
87     | _ -> None in
88   let add_type v = (get_var_type state env v, v) in
89
90   let typed = List.map add_type freevars in
91   let unknowns = List.map none_snd typed in
92   let knowns = List.map some_fst typed in
93
94   match Util.filter_option unknowns with
95   | [] -> Util.filter_option knowns
96   | vs -> raise(Failure("Unknown variables " ^ String.concat ", " vs ^ " within

```

```

anonymous object definition.")) in
92
93 match expr.deets with
94 | Sast.Anonymous(klass, args, refines) ->
95   let (newklass, state) = get_label init_state klass in
96   let freedefs = deanon_freedefs state env refines in
97   let (init_id, ast_class) = deanon_klass args freedefs newklass klass refines
98   in
99     let freeargs = List.map (fun (t, v) -> (t, Sast.Id(v))) freedefs in
100    let instance = Sast.NewObj(newklass, args @ freeargs, Sast.FuncId init_id) in
101    let state = { state with deanon = ast_class :: state.deanon } in
102    (instance, state)
103 | Sast.This -> (Sast.This, init_state)
104 | Sast.Null -> (Sast.Null, init_state)
105 | Sast.Id(id) -> (Sast.Id(id), init_state)
106 | Sast.NewObj(klass, args, funcid) ->
107   let (args, state) = deanon_exprs init_state env args in
108   (Sast.NewObj(klass, args, funcid), state)
109 | Sast.Literal(lit) -> (Sast.Literal(lit), init_state)
110 | Sast.Assign(mem, data) ->
111   let (mem, state) = deanon_expr init_state env mem in
112   let (data, state) = deanon_expr state env data in
113   (Sast.Assign(mem, data), state)
114 | Sast.Deref(arr, idx) ->
115   let (arr, state) = deanon_expr init_state env arr in
116   let (idx, state) = deanon_expr state env idx in
117   (Sast.Deref(arr, idx), state)
118 | Sast.Field(expr, mbr) ->
119   let (expr, state) = deanon_expr init_state env expr in
120   (Sast.Field(expr, mbr), state)
121 | Sast.Invoc(recvr, klass, args, funcid) ->
122   let (recvr, state) = deanon_expr init_state env recvr in
123   let (args, state) = deanon_exprs state env args in
124   (Sast.Invoc(recvr, klass, args, funcid), state)
125 | Sast.Unop(op, expr) ->
126   let (expr, state) = deanon_expr init_state env expr in
127   (Sast.Unop(op, expr), state)
128 | Sast.Binop(l, op, r) ->
129   let (l, state) = deanon_expr init_state env l in
130   let (r, state) = deanon_expr state env r in
131   (Sast.Binop(l, op, r), state)
132 | Sast.Refine(refine, args, ret, switch) ->
133   let (args, state) = deanon_exprs init_state env args in
134   (Sast.Refine(refine, args, ret, switch), state)
135 | Sast.Refinable(refine, switch) ->
136   (Sast.Refinable(refine, switch), init_state)
137
138 (**
139  Update an type-tagged sAST expression to be deanonymized.
140  Returns the deanonymized expr and a possibly updated anon_state
141  @param init_state anon_state value
142  @param env an environment like those attached to stmts in the sAST
143  @param t the type of the expr_detail exp
144  @param exp an expression detail
145  @return ((t, exp'), state') where exp' is exp but deanonymized and
146  state' is an updated version of init_state
147 *)
148 and deanon_expr init_state env (t, exp) =
149   let (deets, state) = deanon_expr_detail init_state env exp in
150   ((t, deets), state)
151
152 (**
153  Deanonymize a list of expressions maintaining the state properly throughout.
154  Returns the list of expressions (deanonymized) and the updated state.
155  @param init_state an anon_state value

```

```

155 @param env an environment like those attached to statments (sAST)
156 @param list a list of expressions (sAST exprs)
157 @return (list', state') where list' is the deanonymized list and
158 state' is the updated state
159 *)
160 and deanon_exprs init_state env list =
161   let folder (rexprs, state) expr =
162     let (deets, state) = deanon_expr state env expr in
163     (deets::rexprs, state) in
164   let (rexprs, state) = List.fold_left folder ([], init_state) list in
165   (List.rev rexprs, state)
166
167 (**
168   Deanonymize a statement.
169   Returns the deanonymized statement and the updated state.
170   @param input_state an anon_state value
171   @param stmt a statement to deanonymize
172   @return (stmt', state') the statement and state, updated.
173   *)
174 and deanon_stmt input_state stmt =
175   let deanon_decl init_state env = function
176     | (vdef, Some(expr)) ->
177       let (deets, state) = deanon_expr init_state env expr in
178       (Sast.Decl(vdef, Some(deets), env), state)
179     | (vdef, _) -> (Sast.Decl(vdef, None, env), init_state) in
180
181   let deanon_exprstmt init_state env expr =
182     let (deets, state) = deanon_expr init_state env expr in
183     (Sast.Expr(deets, env), state) in
184
185   let deanon_return init_state env = function
186     | None -> (Sast.Return(None, env), init_state)
187     | Some(expr) ->
188       let (deets, state) = deanon_expr init_state env expr in
189       (Sast.Return(Some(deets), env), state) in
190
191   let deanon_super init_state env args built_in init_id =
192     let (deets, state) = deanon_exprs init_state env args in
193     (Sast.Super(deets, init_id, built_in, env), state) in
194
195   let deanon_while init_state env (expr, stmts) =
196     let (test, state) = deanon_expr init_state env expr in
197     let (body, state) = deanon_stmts state stmts in
198     (Sast.While(test, body, env), state) in
199
200   let deanon_if init_state env pieces =
201     let folder (rpieces, state) piece =
202       let (piece, state) = match piece with
203       | (None, stmts) ->
204         let (body, state) = deanon_stmts state stmts in
205         ((None, body), state)
206       | (Some(expr), stmts) ->
207         let (test, state) = deanon_expr state env expr in
208         let (body, state) = deanon_stmts state stmts in
209         ((Some(test), body), state) in
210     (piece::rpieces, state) in
211   let (rpieces, state) = List.fold_left folder ([], init_state) pieces in
212   (Sast.If(List.rev rpieces, env), state) in
213
214   match stmt with
215   | Sast.Decl(vdef, opt_expr, env) -> deanon_decl input_state env (vdef, opt_expr)
216   | Sast.If(pieces, env) -> deanon_if input_state env pieces
217   | Sast.While(test, body, env) -> deanon_while input_state env (test, body)
218   | Sast.Expr(expr, env) -> deanon_exprstmt input_state env expr
219   | Sast.Return(opt_expr, env) -> deanon_return input_state env opt_expr

```

```

220 | Sast.Super(args, init_id, built_in, env) -> deanon_super init_state env args
    built_in init_id
221
222 (**
223  Update an entire list of statements to be deanonymized.
224  Maintains the update to the state throughout the computation.
225  Returns a deanonymized list of statements and an updated state.
226  @param init_state an anon_state value
227  @param stmts a list of statements
228  @return (stmts', state') the updated statements and state
229  *)
230 and deanon_stmts init_state stmts =
231   let folder (rstmts, state) stmt =
232     let (stmt, state) = deanon_stmt state stmt in
233     (stmt::rstmts, state) in
234   let (rstmts, state) = List.fold_left folder ([], init_state) stmts in
235   (List.rev rstmts, state)
236
237 (**
238  Deanonymize the body of a function.
239  Return the updated function and updated state.
240  @param init_state an anon_state value
241  @param func a func_def (sAST)
242  @return (func', state') the updated function and state
243  *)
244 let deanon_func init_state (func : Sast.func_def) =
245   let (stmts, state) = deanon_stmts init_state func.body in
246   ({ func with body = stmts }, state)
247
248 (**
249  Deanonymize an entire list of functions, threading the state
250  throughout and maintaining the changes. Returns the list of
251  functions, updated, and the updated state.
252  @param init_state an anon_state value
253  @param funcs a list of functions
254  @return (funcs', state') the updated functions and state
255  *)
256 let deanon_funcs init_state funcs =
257   let folder (rfuncs, state) func =
258     let (func, state) = deanon_func state func in
259     (func::rfuncs, state) in
260   let (funcs, state) = List.fold_left folder ([], init_state) funcs in
261   (List.rev funcs, state)
262
263 (**
264  Deanonymize an Sast member_def
265  Returns the deanonymized member and a possibly updated state.
266  @param init_state an anon_state value
267  @param mem a member to deanonymize
268  @return (mem', state') the updated member and state
269  *)
270 let deanon_member init_state mem = match mem with
271 | Sast.MethodMem(f) ->
272   let (func, state) = deanon_func init_state f in
273   (Sast.MethodMem(func), state)
274 | Sast.InitMem(f) ->
275   let (func, state) = deanon_func init_state f in
276   (Sast.InitMem(func), state)
277 | mem -> (mem, init_state)
278
279 (**
280  Deanonymize a list of members. Return the deanonymized list
281  and a possibly updated state.
282  @param init_state an anon_state value
283  @param members a list of members to deanonymize

```

```

284     @return (mems', state') the updated members and state
285 *)
286 let deanon_memlist (init_state : anon_state) (members : Sast.member_def list) : (Sast.
    member_def list * anon_state) =
287     let folder (rmems, state) mem =
288         let (mem, state) = deanon_member state mem in
289         (mem::rmems, state) in
290     let (rmems, state) = List.fold_left folder ([], init_state) members in
291     (List.rev rmems, state)
292
293 (**
294     Deanonymize an entire class. Return the deanonymized class
295     and an updated state.
296     @param init_state an anon_state value
297     @param aklass an sAST class to deanonymize
298     @return (class', state') the updated class and state.
299 *)
300 let deanon_class init_state (aklass : Sast.class_def) =
301     let s = aklass.sections in
302     let state = { init_state with current = aklass.klass } in
303     let (publics, state) = deanon_memlist state s.publics in
304     let (protects, state) = deanon_memlist state s.protects in
305     let (privates, state) = deanon_memlist state s.privates in
306     let (refines, state) = deanon_funcs state s.refines in
307     let (mains, state) = deanon_funcs state s.mains in
308     let sections : Sast.class_sections_def =
309         {
310             publics = publics;
311             protects = protects;
312             privates = privates;
313             refines = refines;
314             mains = mains } in
315     let cleaned = { aklass with sections = sections } in
316     (state.deanon, { state with clean = cleaned::state.clean; current = ""; deanon = []
    })
317
318 (** A starting state for deanonymization. *)
319 let empty_deanon_state data =
320     {
321         labeler = StringMap.empty;
322         deanon = [];
323         clean = [];
324         data = data;
325         current = ""; }
326
327 (**
328     Given global class information and parsed and tagged classes,
329     deanonymize the classes. This will add more classes to the
330     global data, which will be updated accordingly.
331     @param klass_data global class_data info
332     @param sast_klasses tagged sAST class list
333     @return If everything goes okay with updating the global data
334     for each deanonymization, then left((state', data')) will be
335     returned where state' contains all (including newly created)
336     sAST classes in its clean list and data' has been updated to
337     reflect any new classes. If anything goes wrong, Right(issue)
338     is returned, where the issue is just as in building the global
339     class_data info to begin with, but now specific to what goes
340     on in deanonymization (i.e. restricted to those restricted
341     classes themselves).
342 *)
343 let deanonymize klass_data sast_klasses =
344     let is_empty = function
345         | [] -> true
346         | _ -> false in
347
348     let rec run_deanon init_state asts sasts = match asts, sasts with

```

```

347 (* Every sAST has been deanonymized, even the deanonymized ones converted into
sASTs
348 * Every Ast has been sAST'd too. So we are done.
349 *)
350 | [], [] ->
351   if is_empty init_state.deanon then Left((init_state.data, init_state.clean))
352   else raise(Failure("Deanonimization somehow did not recurse properly."))
353
354 | [], klass::rest ->
355   let (asts, state) = deanon_class init_state klass in
356   run_deanon state asts rest
357
358 | klass::rest, _ -> match KlassData.append_leaf init_state.data klass with
359   | Left(data) ->
360     let sast_klass = BuildSast.ast_to_sast_klass data klass in
361     let state = { init_state with data = data } in
362     run_deanon state rest (sast_klass::sasts)
363   | Right(issue) -> Right(issue) in
364
365 run_deanon (empty_deanon_state klass_data) [] sast_classes

```

Source 129: "Unanonymous.ml"

```

1  open StringModules
2  open Util
3
4  val fold_classes : GlobalData.class_data -> ('a -> Ast.class_def -> 'a) -> 'a -> 'a
5  val map_classes : GlobalData.class_data -> ('a StringMap.t -> Ast.class_def -> 'a
StringMap.t) -> 'a StringMap.t
6  val dfs_errors : GlobalData.class_data -> (string -> 'a -> 'b -> ('a * 'b)) -> 'a -> 'b
-> 'b
7
8  val build_class_data : Ast.class_def list -> (GlobalData.class_data, GlobalData.
class_data_error) either
9  val build_class_data_test : Ast.class_def list -> (GlobalData.class_data, GlobalData.
class_data_error) either
10
11 val append_leaf : GlobalData.class_data -> Ast.class_def -> (GlobalData.class_data,
GlobalData.class_data_error) either
12 val append_leaf_test : GlobalData.class_data -> Ast.class_def -> (GlobalData.class_data,
GlobalData.class_data_error) either
13
14 val print_class_data : GlobalData.class_data -> unit
15 val errstr : GlobalData.class_data_error -> string

```

Source 130: "KlassData.mli"

```

1  open Ast
2  open Util
3  open StringModules
4  open GlobalData
5  open Klass
6
7  (** Build a class_data object. *)
8
9  (** Construct an empty class_data object *)
10 let empty_data : class_data = {
11   known = StringSet.empty;
12   classes = StringMap.empty;
13   parents = StringMap.empty;
14   children = StringMap.empty;

```



```

15     variables = StringMap.empty;
16     methods = StringMap.empty;
17     refines = StringMap.empty;
18     mains = StringMap.empty;
19     ancestors = StringMap.empty;
20     distance = StringMap.empty;
21     refinable = StringMap.empty;
22 }
23
24 (**
25  Map function collisions to the type used for collection that information.
26  This lets us have a 'standard' form of method / refinement collisions and so
27  we can easily build up a list of them.
28  @param aklass the class we are currently examining (class name — string)
29  @param funcs a list of funcs colliding in aklass
30  @param reqhost are we requiring a host (compiler error if no host and true)
31  @return a tuple representing the collisions — (class name, collision tuples)
32  where collision tuples are ([host.]name, formals)
33 *)
34 let build_collisions aklass funcs reqhost =
35   let to_collision func =
36     let name = match func.host, reqhost with
37     | None, true -> raise (Invalid_argument "Cannot build refinement collisions —
38       refinement without host [compiler error].")
39     | None, _ -> func.name
40     | Some(host), _ -> host ^ "." ^ func.name in
41     (name, List.map fst func.formals) in
42   (aklass, List.map to_collision funcs)
43
44 (** Fold over the values in a class_data record's classes map. *)
45 let fold_classes data folder init =
46   let do_fold _ aklass result = folder result aklass in
47   StringMap.fold do_fold data.classes init
48
49 (**
50  Fold over the values in a class_data record's classes map, but
51  enforce building up a StringMap.
52 *)
53 let map_classes data folder = fold_classes data folder StringMap.empty
54
55 (**
56  Recursively explore the tree starting at the root, accumulating errors
57  in a list as we go. The explorer function should take the current class
58  the current state, the current errors and return a new state / errors
59  pair (updating state when possible if there are errors for further
60  accumulation). This is the state that will be passed to all children,
61  and the errors will accumulate across all children.
62  @param data A class_data record value
63  @param explore Something that goes from the current node to a new state/error pair
64  @init_state the initial state of the system
65  @init_error the initial errors of the system
66  @return The final accumulated errors
67 *)
68 let dfs_errors data explore init_state init_error =
69   let rec recurse aklass state errors =
70     let (state, errors) = explore aklass state errors in
71     let explore_kids errors child = recurse child state errors in
72     let children = map_lookup_list aklass data.children in
73     List.fold_left explore_kids errors children in
74   recurse "Object" init_state init_error
75
76 (**
77  Given a list of classes, build an initial class_data object with
78  the known and classes fields set appropriately. If there are any
79  duplicate class names a StringSet of the collisions will then be

```

```

79     returned in Right, otherwise the data will be returned in Left.
80     @param classes A list of classes
81     @return Left(data) which is a class_data record with the known
82     set filled with names or Right(collisions) which is a set of
83     collisions (StringSet.t)
84 *)
85 let initialize_class_data classes =
86   let build_known (set, collisions) aklass =
87     if StringSet.mem aklass.klass set
88       then (set, StringSet.add aklass.klass collisions)
89       else (StringSet.add aklass.klass set, collisions) in
90   let classes = BuiltIns.built_in_classes @ classes in
91   let build_classes map aklass = StringMap.add aklass.klass aklass map in
92   let (known, collisions) = List.fold_left build_known (StringSet.empty, StringSet.
93   empty) classes in
94   let classes = List.fold_left build_classes StringMap.empty classes in
95   if StringSet.is_empty collisions
96     then Left({ empty_data with known = known; classes = classes })
97     else Right(collisions)
98
99 (**
100  Given an initialized class_data record, build the children map
101  from the classes that are stored within it.
102  The map is from parent to children list.
103  @param data A class_data record
104  @return data but with the children.
105 *)
106 let build_children_map data =
107   let map_builder map aklass = match aklass.klass with
108   | "Object" -> map
109   | _ -> add_map_list (klass_to_parent aklass) aklass.klass map in
110   let children_map = map_classes data map_builder in
111   { data with children = children_map }
112
113 (**
114  Given an initialized class_Data record, build the parent map
115  from the classes that are stored within it.
116  The map is from child to parent.
117  @param data A class_data record
118  @return data but with the parent map updated.
119 *)
120 let build_parent_map data =
121   let map_builder map aklass = match aklass.klass with
122   | "Object" -> map
123   | _ -> StringMap.add (aklass.klass) (klass_to_parent aklass) map in
124   let parent_map = map_classes data map_builder in
125   { data with parents = parent_map }
126
127 (**
128  Validate that the parent map in a class_data record represents a tree rooted at
129  object.
130  @param data a class_data record
131  @return An optional string (Some(string)) when there is an issue.
132 *)
133 let is_tree_hierarchy data =
134   let rec from_object klass checked =
135     match map_lookup klass checked with
136     | Some(true) -> Left(true)
137     | Some(false) -> Right("Cycle detected.")
138     | _ -> match map_lookup klass data.parents with
139     | None -> Right("Cannot find parent after building parent map: " ^ klass)
140     | Some(parent) -> match from_object parent (StringMap.add klass false
141     checked) with
142     | Left(updated) -> Left(StringMap.add klass true updated)
143     | issue -> issue in

```

```

141 let folder result aklass = match result with
142 | Left (checked) -> from_object aklass.klass checked
143 | issue -> issue in
144 let checked = StringMap.add "Object" true StringMap.empty in
145 match fold_classes data folder (Left (checked)) with
146 | Right (issue) -> Some (issue)
147 | _ -> None
148
149 (**
150 Add the class (class name - string) -> ancestors (list of ancestors - string list)
151 map to a
152 class_data record. Note that the ancestors go from 'youngest' to 'oldest' and so
153 should start
154 with the given class (hd) and end with Object (last item in the list).
155 @param data The class_data record to update
156 @return An updated class_data record with the ancestor map added.
157 *)
158 let build_ancestor_map data =
159 let rec ancestor_builder klass map =
160 if StringMap.mem klass map then map
161 else
162 let parent = StringMap.find klass data.parents in
163 let map = ancestor_builder parent map in
164 let ancestors = StringMap.find parent map in
165 StringMap.add klass (klass::ancestors) map in
166 let folder map aklass = ancestor_builder aklass.klass map in
167 let map = StringMap.add "Object" ["Object"] StringMap.empty in
168 let ancestor_map = fold_classes data folder map in
169 { data with ancestors = ancestor_map }
170
171 (**
172 For a given class, build a map of variable names to variable information.
173 If all instance variables are uniquely named, returns Left (map) where map
174 is var name -> (class_section, type) otherwise returns Right (collisions)
175 where collisions are the names of variables that are multiply declared.
176 @param aklass A parsed class
177 @return a map of instance variables in the class
178 *)
179 let build_var_map aklass =
180 let add_var section map (typeId, varId) = add_map.unique varId (section, typeId) map
181 in
182 let map_builder map (section, members) = List.fold_left (add_var section) map members
183 in
184 build_map_track_errors map_builder (klass_to_variables aklass)
185
186 (**
187 Add the class (class name - string) -> variable (var name - string) -> info (section/
188 type
189 pair - class_section * string) table to a class_data record.
190 @param data A class_data record
191 @return Either a list of collisions (in Right) or the updated record (in Left).
192 Collisions are pairs (class name, collisions (var names) for that class)
193 *)
194 let build_class_var_map data =
195 let map_builder (klass_map, collision_list) (_, aklass) =
196 match build_var_map aklass with
197 | Left (var_map) -> (StringMap.add (aklass.klass) var_map klass_map,
198 collision_list)
199 | Right (collisions) -> (klass_map, (aklass.klass, collisions)::collision_list)
200 in
201 match build_map_track_errors map_builder (StringMap.bindings data.classes) with
202 | Left (variable_map) -> Left ({ data with variables = variable_map })
203 | Right (collisions) -> Right (collisions) (* Same value different types
204 parametrically *)

```

```

198 (**
199     Given a class_data record and a class_def value, return the instance variables (just
200     the
201     var_def) that have an unknown type.
202     @param data A class_data record value
203     @param aklass A class_def value
204     @return A list of unknown-typed instance variables in the class
205 *)
206 let type_check_variables data aklass =
207   let unknown_type (var_type, _) = not (is_type data var_type) in
208   let vars = List.flatten (List.map snd (klass_to_variables aklass)) in
209   List.filter unknown_type vars
210
211 (**
212     Given a class_data record, verify that all instance variables of all classes are of
213     known
214     types. Returns the Left of the data if everything is okay, or the Right of a list of
215     pairs,
216     first item being a class, second being variables of unknown types (type, name pairs).
217     @param data A class_data record value.
218     @return Left(data) if everything is okay, otherwise Right(unknown types) where
219     unknown types
220     is a list of (class, var_def) pairs.
221 *)
222 let verify_typed data =
223   let verify_klass klass_name aklass unknowns = match type_check_variables data aklass
224   with
225   | [] -> unknowns
226   | bad -> (klass_name, bad)::unknowns in
227   match StringMap.fold verify_klass data.classes [] with
228   | [] -> Left(data)
229   | bad -> Right(bad)
230
231 (**
232     Given a function, type check the signature (Return, Params).
233     @param data A class_data record value.
234     @param func An Ast.func_def record
235     @return Left(data) if everything is alright; Right([host.]name, option string, (type,
236     name)
237     list) if wrong.
238 *)
239 let type_check_func data func =
240   let atype = is_type data in
241   let check_ret = match func.returns with
242   | Some(vtype) -> if atype vtype then None else Some(vtype)
243   | _ -> None in
244   let check_param (vtype, vname) = if not (atype vtype) then Some((vtype, vname)) else
245   None in
246   let bad_params = filter_option (List.map check_param func.formals) in
247   match check_ret, bad_params, func.host with
248   | None, [], _ -> Left(data)
249   | _, _, None -> Right((func.name, check_ret, bad_params))
250   | _, _, Some(host) -> Right((host ^ "." ^ func.name, check_ret, bad_params))
251
252 (**
253     Given a class_data object and a class, verify that all of its methods have good types
254     (Return and parameters).
255     @param data A class_data record object
256     @param aklass A class_def object
257     @return Left(data) if everything went okay; Right((klass name, (func name, option
258     string,
259     (type, name) list) list))
260 *)
261 let type_check_class data aklass =
262   let folder bad func = match type_check_func data func with
263   | Left(data) -> bad
264   | Right(_) -> bad ++ [(klass_name, [(func_name, option_string, (type, name) list) list])] in
265   StringMap.fold folder data.classes []

```

```

255 | Left(data) -> bad
256 | Right(issue) -> issue::bad in
257 let funcs = List.flatten (List.map snd (klass_to_functions aklass)) in
258 match List.fold_left folder [] funcs with
259 | [] -> Left(data)
260 | bad -> Right((aklass.klass, bad))
261
262 (**
263  Given a class_data object, verify that all classes have methods with good signatures
264  (Return and parameters)
265  @param data A class_data record object
266  @param aklass A class_def object
267  @return Left(data) if everything went okay; Right((klass name, bad_sig list) list)
268  where bad_sig is (func_name, string option, (type, var) list))
269 *)
270 let type_check_signatures data =
271 let folder klass_name aklass bad = match type_check_class data aklass with
272 | Left(data) -> bad
273 | Right(issue) -> issue::bad in
274 match StringMap.fold folder data.classes [] with
275 | [] -> Left(data)
276 | bad -> Right(bad)
277
278 (**
279  Build a map of all the methods within a class, returning either a list of collisions
280  (in Right) when there are conflicting signatures or the map (in Left) when there
281  are not. Keys to the map are function names and the values are lists of func_def's.
282  @param aklass A klass to build a method map for
283  @return Either a list of collisions or a map of function names to func_def's.
284 *)
285 let build_method_map aklass =
286 let add_method (map, collisions) fdef =
287   if List.exists (conflicting_signatures fdef) (map_lookup_list fdef.name map)
288   then (map, fdef::collisions)
289   else (add_map_list fdef.name fdef map, collisions) in
290 let map_builder map funcs = List.fold_left add_method map funcs in
291 build_map_track_errors map_builder (List.map snd (klass_to_methods aklass))
292
293 (**
294  Add the class name (string) -> method name (string) -> methods (func_def list)
295  methods table to a class_data record, given a list of classes. If there are no
296  collisions, the updated record is returned (in Left), otherwise the collision
297  list is returned (in Right).
298  @param data A class_data record
299  @return Either a list of collisions (in Right) or the updated record (in Left).
300  Collisions are pairs (class name, colliding methods for that class). Methods collide
301  if they have conflicting signatures (ignoring return type).
302 *)
303 let build_class_method_map data =
304 let map_builder (klass_map, collision_list) (_, aklass) =
305   match build_method_map aklass with
306   | Left(method_map) -> (StringMap.add aklass.klass method_map klass_map,
307    collision_list)
308   | Right(collisions) -> (klass_map, (build_collisions aklass.klass collisions
309    false)::collision_list) in
310 match build_map_track_errors map_builder (StringMap.bindings data.classes) with
311 | Left(method_map) -> Left({ data with methods = method_map })
312 | Right(collisions) -> Right(collisions) (* Same value different types
313 parametrically *)
314
315 (**
316  Build the map of refinements for a given class. Keys to the map are 'host.name'
317  @param aklass aklass A class to build a refinement map out of
318  @return Either a list of collisions (in Right) or the map (in left). Refinements
319  conflict when they have the same name ('host.name' in this case) and have the same

```

```

317     argument type sequence.
318 *)
319 let build_refinement_map aklass =
320   let add_refinement (map, collisions) func = match func.host with
321     | Some(host) =>
322       let key = func.name ^ "." ^ host in
323       if List.exists (conflicting_signatures func) (map.lookup_list key map)
324       then (map, func::collisions)
325       else (add_map_list key func map, collisions)
326     | None => raise(Failure("Compilation error — non-refinement found in searching
for refinements.")) in
327   build_map_track_errors add_refinement aklass.sections.refines
328
329 (**
330  Add the class name (string) -> refinement ('host.name' - string) -> func list
331  map to a class_data record. If there are no collisions (conflicting signatures
332  given the same host), then the updated record is returned (in Left) otherwise
333  a list of collisions is returned (in Right).
334  @param data A class_data record
335  @param classes A list of parsed classes
336  @return either a list of collisions (in Right) or the updated record (in Left).
337  Collisions are (class, (host, method, formals) list)
338 *)
339 let build_class_refinement_map data =
340   let map_builder (class_map, collision_list) (_, aklass) =
341     match build_refinement_map aklass with
342     | Left(refinement_map) -> (StringMap.add aklass.class refinement_map
343   class_map, collision_list)
344     | Right(collisions) -> (class_map, (build_collisions aklass.class collisions
true)::collision_list) in
345   match build_map_track_errors map_builder (StringMap.bindings data.classes) with
346   | Left(refinement_map) -> Left({ data with refines = refinement_map })
347   | Right(collisions) -> Right(collisions) (* Same value different types
parametrically *)
348
349 (**
350  Add a map of main functions, from class name (string) to main (func_def) to the
351  class_data record passed in. Returns a list of collisions if any class has more
352  than one main (in Right) or the updated record (in Left)
353  @param data A class_data record
354  @param classes A list of parsed classes
355  @return Either the collisions (Right) or the updated record (Left)
356 *)
357 let build_main_map data =
358   let add_class (map, collisions) (_, aklass) = match aklass.sections.mains with
359     | [] -> (map, collisions)
360     | [main] -> (StringMap.add aklass.class main map, collisions)
361     | _ -> (map, aklass.class :: collisions) in
362   match build_map_track_errors add_class (StringMap.bindings data.classes) with
363   | Left(main_map) -> Left({ data with mains = main_map })
364   | Right(collisions) -> Right(collisions) (* Same value different types
parametrically *)
365
366 (**
367  Given a class_data record, verify that there are no double declarations of instance
368  variables as you go up the tree. This means that no two classes along the same root
369  leaf path can have the same public / protected variables, and a private cannot be
370  a public/protected variable of an ancestor.
371  @param data A class_data record.
372  @return Left(data) if everything was okay or Right(collisions) where collisions is
373  a list of pairs of collision information — first item class, second item a list of
374  colliding variables for that class (name, ancestor where they collide)
375 *)
376 let check_field_collisions data =
377   let check_vars aklass var (section, _) (fields, collisions) = match map_lookup var

```

```

377     fields, section with
378     | Some(ancestor), _ -> (fields, (ancestor, var)::collisions)
379     | None, Privates -> (fields, collisions)
380     | None, _ -> (StringMap.add var aklass fields, collisions) in
381
382     let check_class_vars aklass fields =
383     let vars = StringMap.find aklass data.variables in
384     StringMap.fold (check_vars aklass) vars (fields, []) in
385
386     let dfs_explorer aklass fields collisions =
387     match check_class_vars aklass fields with
388     | (fields, []) -> (fields, collisions)
389     | (fields, cols) -> (fields, (aklass, cols)::collisions) in
390
391     match dfs_errors data dfs_explorer StringMap.empty [] with
392     | [] -> Left(data)
393     | collisions -> Right(collisions)
394
395     (**
396     Check to make sure that we don't have conflicting signatures as we go down the class
397     tree.
398     @param data A class_data record value
399     @return Left(data) if everything is okay, otherwise a list of (string
400     *)
401     let check_ancestor_signatures data =
402     let check_sigs meth_name funcs (methods, collisions) =
403     let updater (known, collisions) func =
404     if List.exists (conflicting_signatures func) known
405     then (known, func::collisions)
406     else (func::known, collisions) in
407     let apriori = map_lookup_list meth_name methods in
408     let (known, collisions) = List.fold_left updater (apriori, collisions) funcs in
409     (StringMap.add meth_name known methods, collisions) in
410
411     let skip_init meth_name funcs acc = match meth_name with
412     | "init" -> acc
413     | _ -> check_sigs meth_name funcs acc in
414
415     let check_class_meths aklass parent_methods =
416     let methods = StringMap.find aklass data.methods in
417     StringMap.fold skip_init methods (parent_methods, []) in
418
419     let dfs_explorer aklass methods collisions =
420     match check_class_meths aklass methods with
421     | (methods, []) -> (methods, collisions)
422     | (methods, cols) -> (methods, (build_collisions aklass cols false)::
423     collisions) in
424
425     match dfs_errors data dfs_explorer StringMap.empty [] with
426     | [] -> Left(data)
427     | collisions -> Right(collisions)
428
429     (**
430     Verifies that each class is able to be instantiated.
431     @param data A class_data record
432     @return Either the data is returned in Left or a list of uninstantiable classes in
433     Right
434     *)
435     let verify_instantiable data =
436     let uninstantiable klass =
437     let inits = class_method_lookup data klass "init" in
438     not (List.exists (fun func -> func.section <> Privates) inits) in
439     let klassses = StringSet.elements data.known in
440     match List.filter uninstantiable klassses with
441     | [] -> Left(data)

```

```

438 | bad -> Right(bad)
439
440 (**
441   Given a class and a list of its ancestors, build a map detailing the distance
442   between the class and any of its ancestors. The distance is the number of hops
443   one must take to get from the given class to the ancestor. The distance between
444   an Object and itself should be 0, and the largest distance should be to object.
445   @param klass The class to build the table for
446   @param ancestors The list of ancestors of the given class.
447   @return A map from class names to integers
448 *)
449 let build_distance klass ancestors =
450   let map_builder (map, i) item = (StringMap.add item i map, i+1) in
451   fst (List.fold_left map_builder (StringMap.empty, 0) ancestors)
452
453 (**
454   Add a class (class name - string) -> class (class name - string) -> distance (int
455   option)
456   table a given class_data record. The distance is always a positive integer and so the
457   first type must be either the same as the second or a subtype, else None is returned.
458   Note that this requires that the ancestor map be built.
459   @param data The class_data record to update.
460   @return The class_data record with the distance map added.
461 *)
462 let build_distance_map data =
463   let distance_map = StringMap.mapi build_distance data.ancestors in
464   { data with distance = distance_map }
465
466 (**
467   Update the refinement dispatch uid table with a given set of refinements.
468   @param parent The class the refinements will come from
469   @param refines A list of refinements
470   @param table The refinement dispatch table
471   @return The updated table
472 *)
473 let update_refinable parent refines table =
474   let toname f = match f.host with
475     | Some(host) -> host
476     | _ -> raise(Invalid_argument("Compiler error; we have refinement without host
477   for " ^ f.name ^ " in " ^ f.inclass ^ ".")) in
478   let folder amap f = add_map_list (toname f) f amap in
479   let map = if StringMap.mem parent table then StringMap.find parent table else
480   StringMap.empty in
481   let map = List.fold_left folder map refines in
482   StringMap.add parent map table
483
484 (**
485   Add the refinable (class name -> host.name -> refinables list) table to the
486   given class_data record, returning the updated record.
487   @param data A class_data record info
488   @return A class_data object with the refinable updated
489 *)
490 let build_refinable_map data =
491   let updater klass_name aklass table = match klass_name with
492     | "Object" -> table
493     | _ -> let parent = klass_to_parent aklass in update_refinable parent aklass.
494   sections.refines table in
495   let refinable = StringMap.fold updater data.classes StringMap.empty in
496   { data with refinable = refinable }
497
498 (** These are just things to pipe together building a class_data record pipeline *)
499 let initial_data classes = match initialize_class_data classes with
500   | Left(data) -> Left(data)
501   | Right(collisions) -> Right(DuplicateClasses(StringSet.elements collisions))
502 let append_children data = Left(build_children_map data)

```



```

499 let append-parent data = Left(build-parent.map data)
500 let test-tree data = match is-tree-hierarchy data with
501   | None -> Left(data)
502   | Some(problem) -> Right(HierarchyIssue(problem))
503 let append-ancestor data = Left(build-ancestor.map data)
504 let append-distance data = Left(build-distance.map data)
505 let append-variables data = match build-class-var-map data with
506   | Left(data) -> Left(data)
507   | Right(collisions) -> Right(DuplicateVariables(collisions))
508 let test-types data = match verify-typed data with
509   | Left(data) -> Left(data)
510   | Right(bad) -> Right(UnknownTypes(bad))
511 let test-fields data = match check-field-collisions data with
512   | Left(data) -> Left(data)
513   | Right(collisions) -> Right(DuplicateFields(collisions))
514 let append-methods data = match build-class-method-map data with
515   | Left(data) -> Left(data)
516   | Right(collisions) -> Right(ConflictingMethods(collisions))
517 let test-init data = match verify-instantiable data with
518   | Left(data) -> Left(data)
519   | Right(bad) -> Right(Uninstantiable(bad))
520 let test-inherited-methods data = match check-ancestor-signatures data with
521   | Left(data) -> Left(data)
522   | Right(collisions) -> Right(ConflictingInherited(collisions))
523 let append-refines data = match build-class-refinement-map data with
524   | Left(data) -> Left(data)
525   | Right(collisions) -> Right(ConflictingRefinements(collisions))
526 let test-signatures data = match type-check-signatures data with
527   | Left(data) -> Left(data)
528   | Right(bad) -> Right(PoorlyTypedSigs(bad))
529 let append-refinable data = Left(build-refinable.map data)
530 let append-mains data = match build-main-map data with
531   | Left(data) -> Left(data)
532   | Right(collisions) -> Right(MultipleMains(collisions))
533
534 let test_list =
535   [ append-children ; append-parent ; test-tree ; append-ancestor ;
536     append-distance ; append-variables ; test-fields ; test-types ;
537     append-methods ; test-init ; test-inherited-methods ; append-refines ;
538     test-signatures ; append-refinable ; append-mains ]
539
540 let production_list =
541   [ append-children ; append-parent ; test-tree ; append-ancestor ;
542     append-distance ; append-variables ; test-fields ; append-methods ;
543     test-init ; append-refines ; append-mains ]
544
545 let build-class-data classes = seq (initial_data classes) test_list (*production_list*)
546 let build-class-data-test classes = seq (initial_data classes) test_list
547
548 let append-leaf-known aklass data =
549   let updated = StringSet.add aklass.klass data.known in
550   if StringSet.mem aklass.klass data.known
551     then Right(DuplicateClasses([aklass.klass]))
552     else Left({ data with known = updated })
553 let append-leaf-classes aklass data =
554   let updated = StringMap.add aklass.klass aklass.data.classes in
555   Left({ data with classes = updated })
556 let append-leaf-tree aklass data =
557   (* If we assume data is valid and data has aklass's parent then we should be fine *)
558   let parent = klass_to_parent aklass in
559   if StringMap.mem parent data.classes
560     then Left(data)
561     else Right(HierarchyIssue("Appending a leaf without a known parent.))
562 let append-leaf-children aklass data =
563   let parent = klass_to_parent aklass in

```

```

564   let updated = add_map_list parent aklass.klass data.children in
565   Left({ data with children = updated })
566 let append_leaf_parent aklass data =
567   let parent = class_to_parent aklass in
568   let updated = StringMap.add aklass.klass parent data.parents in
569   Left({ data with parents = updated })
570 let append_leaf_variables aklass data = match build_var_map aklass with
571 | Left(vars) ->
572   let updated = StringMap.add aklass.klass vars data.variables in
573   Left({ data with variables = updated })
574 | Right(collisions) -> Right(DuplicateVariables([(aklass.klass, collisions)]))
575 let append_leaf_test_fields aklass data =
576   let folder collisions var = match class_field_lookup data (class_to_parent aklass)
577   var with
578   | Some(., -, Privates) -> collisions
579   | Some((ancestor, -, section)) -> (ancestor, var)::collisions
580   | _ -> collisions in
581   let variables = List.flatten (List.map snd (class_to_variables aklass)) in
582   let varnames = List.map snd variables in
583   match List.fold_left folder [] varnames with
584   | [] -> Left(data)
585   | collisions -> Right(DuplicateFields([(aklass.klass, collisions)]))
586 let append_leaf_type_vars aklass data =
587   match type_check_variables data aklass with
588   | [] -> Left(data)
589   | bad -> Right(UnknownTypes([(aklass.klass, bad)]))
590 let append_leaf_methods aklass data = match build_method_map aklass with
591 | Left(meths) ->
592   let updated = StringMap.add aklass.klass meths data.methods in
593   Left({ data with methods = updated })
594 | Right(collisions) -> Right(ConflictingMethods([build_collisions aklass.klass
595 collisions false]))
596 let append_leaf_test_inherited aklass data =
597   let folder collisions meth = match class_ancestor_method_lookup data aklass.klass
598   meth.name true with
599   | [] -> collisions
600   | funcs -> match List.filter (conflicting_signatures meth) funcs with
601   | [] -> collisions
602   | cols -> cols in
603   let skipinit (func : Ast.func_def) = match func.name with
604   | "init" -> false
605   | _ -> true in
606   let functions = List.flatten (List.map snd (class_to_methods aklass)) in
607   let noninits = List.filter skipinit functions in
608   match List.fold_left folder [] noninits with
609   | [] -> Left(data)
610   | collisions -> Right(ConflictingInherited([build_collisions aklass.klass
611 collisions false]))
612 let append_leaf_instantiable aklass data =
613   let is_init mem = match mem with
614   | InitMem(_) -> true
615   | _ -> false in
616   if List.exists is_init (aklass.sections.protects) then Left(data)
617   else if List.exists is_init (aklass.sections.publics) then Left(data)
618   else Right(Uninstantiable([aklass.klass]))
619 let append_leaf_refines aklass data = match build_refinement_map aklass with
620 | Left(refs) ->
621   let updated = StringMap.add aklass.klass refs data.refines in
622   Left({ data with refines = updated })
623 | Right(collisions) -> Right(ConflictingRefinements([build_collisions aklass.klass
624 collisions true]))
625 let append_leaf_mains aklass data = match aklass.sections.mains with
626 | [] -> Left(data)
627 | [main] ->
628   let updated = StringMap.add aklass.klass main data.mains in

```

```

624     Left({ data with mains = updated })
625 | _ -> Right(MultipleMains([aklass.klass]))
626 let append_leaf_signatures aklass data = match type_check_class data aklass with
627 | Left(data) -> Left(data)
628 | Right(bad) -> Right(PoorlyTypedSigs([bad]))
629 let append_leaf_ancestor aklass data =
630 let parent = class_to_parent aklass in
631 let ancestors = aklass.klass :: (StringMap.find parent data.ancestors) in
632 let updated = StringMap.add aklass.klass ancestors data.ancestors in
633 Left({ data with ancestors = updated })
634 let append_leaf_distance aklass data =
635 let ancestors = StringMap.find aklass.klass data.ancestors in
636 let distance = build_distance aklass.klass ancestors in
637 let updated = StringMap.add aklass.klass distance data.distance in
638 Left({ data with distance = updated })
639 let append_leaf_refinable aklass data =
640 let parent = class_to_parent aklass in
641 let updated = update_refinable parent aklass.sections.refines data.refinable in
642 Left({ data with refinable = updated })
643
644 let production_leaf =
645 [ append_leaf_known ; append_leaf_classes ; append_leaf_children ; append_leaf_parent
646   ; append_leaf_ancestor ; append_leaf_distance ; append_leaf_variables ;
647   append_leaf_test_fields ;
648   append_leaf_methods ; append_leaf_instantiable ; append_leaf_refines ;
649   append_leaf_signatures ;
650   append_leaf_mains ]
651 let test_leaf =
652 [ append_leaf_known ; append_leaf_classes ; append_leaf_children ; append_leaf_parent
653   ; append_leaf_ancestor ; append_leaf_distance ; append_leaf_variables ;
654   append_leaf_test_fields ;
655   append_leaf_type_vars ; append_leaf_methods ; append_leaf_instantiable ;
656   append_leaf_test_inherited ;
657   append_leaf_refines ; append_leaf_refinable ; append_leaf_mains ]
658
659 let leaf_with_class actions data class = seq (Left(data)) (List.map (fun f -> f class)
660 actions)
661 let append_leaf = leaf_with_class test_leaf (* production_leaf *)
662 let append_leaf_test = leaf_with_class test_leaf
663
664 let append_leaf_test data aklass =
665 let with_class f = f aklass in
666 let actions =
667 [ append_leaf_known ; append_leaf_classes ; append_leaf_children ;
668   append_leaf_parent ;
669   append_leaf_ancestor ; append_leaf_distance ; append_leaf_variables ;
670   append_leaf_test_fields ;
671   append_leaf_type_vars ; append_leaf_methods ; append_leaf_instantiable ;
672   append_leaf_test_inherited ;
673   append_leaf_refines ; append_leaf_refinable ; append_leaf_mains ] in
674 seq (Left(data)) (List.map with_class actions)
675
676 (**
677   Print class data out to stdout.
678 *)
679 let print_class_data data =
680 let id x = x in
681 let from_list lst = Format.sprintf "[%s]" (String.concat ", " lst) in
682 let table_printer tbl name stringer =
683 let printer p s i = Format.sprintf "\t%s : %s => %s\n" p s (stringer i) in
684 print_string (name ^ ":\n");
685 print_lookup_table tbl printer in
686 let map_printer map name stringer =

```

```

679     let printer k i = Format.sprintf "\t%s => %s\n" k (stringer i) in
680     print_string (name ^ ":\n");
681     print_lookup_map map printer in
682
683 let func_list = function
684 | [one] -> full_signature_string one
685 | list -> let sigs = List.map (fun f -> "\n\t\t" ^ (full_signature_string f))
list in
        String.concat "" sigs in
686
687
688 let func_of_list funcs =
689     let sigs = List.map (fun f -> "\n\t\t" ^ f.inklass ^ "->" ^ (
full_signature_string f)) funcs in
        String.concat "" sigs in
690
691
692 let class_printer cdef =
693     let rec count sect = function
694         | (where, members)::_ when where = sect -> List.length members
695         | _::rest -> count sect rest
696         | [] -> raise (Failure("The impossible happened — searching for a section
that should exist doesn't exist.)) in
697     let vars = klass_to_variables cdef in
698     let funcs = klass_to_functions cdef in
699     let format = ""^^"from %s:  M(%d/%d/%d) F(%d/%d/%d) R(%d) M(%d)" in
700     let parent = match cdef.klass with
701     | "Object" -> ""
702     | _ -> klass_to_parent cdef in
703     Format.sprintf format parent
704     (count Privates funcs) (count Protects funcs) (count Publics funcs)
705     (count Privates vars) (count Protects vars) (count Publics vars)
706     (count Refines funcs) (count Mains funcs) in
707
708 let print_list list =
709     let rec list_printer spaces endl space = function
710     | [] -> if endl then () else print_newline ()
711     | list when spaces = 0 -> print_string "\t"; list_printer 8 false false list
712     | list when spaces > 60 -> print_newline (); list_printer 0 true false list
713     | item::rest ->
714         if space then print_string " " else ();
715         print_string item;
716         list_printer (spaces + String.length item) false true rest in
717     list_printer 0 true false list in
718
719 Printf.printf "Types:\n";
720 print_list (StringSet.elements data.known);
721 print_newline ();
722 map_printer data.classes "Classes" class_printer;
723 print_newline ();
724 map_printer data.parents "Parents" id;
725 print_newline ();
726 map_printer data.children "Children" from_list;
727 print_newline ();
728 map_printer data.ancestors "Ancestors" from_list;
729 print_newline ();
730 table_printer data.distance "Distance" string_of_int;
731 print_newline ();
732 table_printer data.variables "Variables" (fun (sect, t) -> Format.sprintf "%s %s" (
section_string sect) t);
733 print_newline ();
734 table_printer data.methods "Methods" func_list;
735 print_newline ();
736 table_printer data.refines "Refines" func_list;
737 print_newline ();
738 map_printer data.mains "Mains" full_signature_string;
739 print_newline ();

```

```

740     table_printer data.refinable "Refinable" func_of_list
741
742
743     (* ERROR HANDLING *)
744
745     let args lst = Format.sprintf "(%s)" (String.concat ", " lst)
746     let asig (name, formals) = Format.sprintf "%s %s" name (args formals)
747     let aref (name, formals) = asig (name, formals)
748
749     let dupvar (klass, vars) = match vars with
750     | [var] -> "Class " ^ klass ^ "'s instance variable " ^ var ^ " is multiply declared"
751     | _ -> "Class " ^ klass ^ " has multiply declared variables: [" ^ (String.concat ", "
       vars) ^ "]"
752
753     let dupfield (klass, fields) = match fields with
754     | [(ancestor, var)] -> "Class " ^ klass ^ "'s instance variable " ^ var ^ " was
       declared in ancestor " ^ ancestor ^ "."
755     | _ -> "Class " ^ klass ^ " has instance variables declared in ancestors: [" ^ String
       .concat ", " (List.map (fun (a, v) -> v ^ " in " ^ a) fields) ^ "]"
756
757     let show_vdecls vs = "[" ^ String.concat ", " (List.map (fun (t,v) -> t ^ ":" ^ v) vs) ^
       "]"
758
759     let unknowntypes (klass, types) = match types with
760     | [(vtype, vname)] -> "Class " ^ klass ^ "'s instancevariable " ^ vname ^ " has
       unknown type " ^ vtype ^ "."
761     | _ -> "Class " ^ klass ^ " has instance variables with unknown types: " ^
       show_vdecls types
762
763     let badsig1 klass (func, ret, params) = match ret, params with
764     | None, params -> "Class " ^ klass ^ "'s " ^ func ^ " has poorly typed parameters: "
       ^ show_vdecls params
765     | Some(rval), [] -> "Class " ^ klass ^ "'s " ^ func ^ " has an invalid return type: "
       ^ rval ^ "."
766     | Some(rval), p -> "Class " ^ klass ^ "'s " ^ func ^ " has invalid return type " ^
       rval ^ " and poorly typed parameters: " ^ show_vdecls p
767
768     let badsig (klass, badfuncs) = String.concat "\n" (List.map (badsig1 klass) badfuncs)
769
770     let dupmeth (klass, meths) =
771     match meths with
772     | [(name, formals)] -> Format.sprintf "Class %s's method %s has multiple
       implementations taking %s" klass name (args formals)
773     | _ -> Format.sprintf "Class %s has multiple methods with conflicting signatures
       :\n\t%s" klass (String.concat "\n\t" (List.map asig meths))
774
775     let dupinherit (klass, meths) =
776     match meths with
777     | [(name, formals)] -> Format.sprintf "Class %s's method %s has conflicts with an
       inherited method taking %s" klass name (args formals)
778     | _ -> Format.sprintf "Class %s has multiple methods with conflicting with
       inherited methods:\n\t%s" klass (String.concat "\n\t" (List.map asig meths))
779
780     let dupref (klass, refines) =
781     match refines with
782     | [refine] -> Format.sprintf "Class %s refinement %s is multiply defined." klass (
       aref refine)
783     | _ -> Format.sprintf "Class %s has multiple refinements multiply defined:\n\t%s"
       klass (String.concat "\n\t" (List.map aref refines))
784
785     let errstr = function
786     | HierarchyIssue(s) -> s
787     | DuplicateClasses(klasses) -> (match klasses with
788     | [klass] -> "Multiple classes named " ^ klass
789     | _ -> "Multiple classes share the names [" ^ (String.concat ", " klasses) ^ "]")
790     | DuplicateVariables(list) -> String.concat "\n" (List.map dupvar list)

```

```

790 | DuplicateFields(list) -> String.concat "\n" (List.map dupfield list)
791 | UnknownTypes(types) -> String.concat "\n" (List.map unknowntypes types)
792 | ConflictingMethods(list) -> String.concat "\n" (List.map dupmeth list)
793 | ConflictingInherited(list) -> String.concat "\n" (List.map dupinherit list)
794 | PoorlyTypedSigs(list) -> String.concat "\n" (List.map badsig list)
795 | Uninstantial(klasses) -> (match klasses with
796 |   [klass] -> "Class " ^ klass ^ " does not have a usable init."
797 |   _ -> "Multiple classes are not instantiable: [" ^ String.concat ", " klasses ^
    "]"")
798 | ConflictingRefinements(list) -> String.concat "\n" (List.map dupref list)
799 | MultipleMains(klasses) -> (match klasses with
800 |   [klass] -> "Class " ^ klass ^ " has multiple mains defined."
801 |   _ -> "Multiple classes have more than one main: [" ^ String.concat ", " klasses
    ^ "]"")

```

Source 131: "KlassData.ml"