GAMMA

$\Gamma\alpha\gamma$

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

2 Language Tutorial

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

```
class IOTest:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := promptInteger("Please enter an integer")
      Float f := promptFloat("Please enter a float")
      p.printString("Sum of integer + float = ")
      p.printFloat(i.toF() + f)
      p.printString("\n")

  private:
    void prompt(String msg):
      system.out.printString(msg)
      system.out.printString(": ")

    Integer promptInteger(String msg):
      prompt(msg)
      return system.in.scanInteger()

    Float promptFloat(String msg):
      prompt(msg)
      return system.in.scanFloat()

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

Example 1: "A simple I/O example"

We start with a definition of our class.

```
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`.)

```
    public:
        init():
            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`.

```
  private:
    void prompt(String msg):
      system.out.printString(msg)
      system.out.printString(": ")

    Integer promptInteger(String msg):
      prompt(msg)
      return system.in.scanInteger()

    Float promptFloat(String msg):
      prompt(msg)
      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.

```
    void interact():
      Printer p := system.out
      Integer i := promptInteger("Please enter an integer")
      Float f := promptFloat("Please enter a float")
      p.printString("Sum of integer + float = ")
      p.printFloat(i.toF() + f)
      p.printString("\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.

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

3 Language Reference Manual

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 identifer 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 syntaxic 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 seperated 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 indentaiton.

```
    <<stmt1_statement>>
    <<stmt2_statement>>
    <<...>>
```

This is pattern is elementry to write.

```
    Mouse mouse = new Mouse()
    mouse.click()
    mouse.click_fast()
    mouse.click("Screen won't respond")
    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:

```
var_identifier := val_expr
```

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

```
a := 3
b := 4
```

Example 3: Variable Assignment for the Pythagorean Theorem

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

```
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:

```
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. [– only one dimensional arrays implemented –]

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

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

```
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:

```
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.

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

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

```
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`.

```
<< 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.

```
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.

```
op1_expr operator op2_expr
```

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

```
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 precidence in relation to its fellow operators. Operators of higher precedence are enacted first. Please consider the following table for determining precidence:

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.

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

Table 1: Operator Precedence

```
{{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.

```
Float pythagorean_theorem(Float a, Float b):
    Float c
    c := ( a^2 + b^2 )^0.5
    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.

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

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.

```
class class_identifier <<extends superclass_identifier>>:
    <<main_method>>
    <<{{private | protected | public | refinement}} section1>>
    <<{{private | protected | public | refinement}} section1>>
    <<...>>
```

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.

```
class Geometric_Shape:
    private:
        String name
        Float area
        Float circumfrence
    public:
        init (String name):
            this.name = name
            if (refinable(improve_name)):
              this.name += refine improve_name() to String

            return

        Float get_area():
            Float area
            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.

```
new class_identifier(<<arg1_expr>> <<,arg2_expr>> <<...>>)
```

For instance:

```
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. Note that anonymous class instantiation must be enclosed in parenthesis (parser error we need to still figure out).

```
new superclass_identifier(<<arg1_expr>> <<,arg2_expr>> <<...>>):
    <<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`.

```
if (test1_expr): if1_body
<<elsif (test2_expr) if2_body>>
<<elsif(test3_expr) if3_body>>
<<...>>
<<else if4_body>>
```

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

```
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 feature was planned but not implemented –]. This is done in addition to normal invocation. Also note that all overloaded methods of the same name share the same refinements.

```
refine refine_identifier to refine_type
```

The Refinable Test The original programmer cannot garuantee 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.

```
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`.

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 returns `false`.

Float If two Float instances have an absolute difference of greater than 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`.

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`.

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`.

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`.

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`.

3.4.7 The Operator `+`

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

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, and Boolean `:=` 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, and Boolean 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:

• *Classs may extend another class or default to extending Object*

⟨class⟩ ⇒

  class ⟨class id⟩⟨extend⟩ **:** ⟨class section⟩*****

⟨extend⟩ ⇒

  ϵ

  | **extends** ⟨class id⟩

• *Sections – private protected public refinements and main*

⟨class section⟩ ⇒

  ⟨refinement⟩

  | ⟨access group⟩

  | ⟨main⟩

• *Refinements are named method dot refinement*

⟨refinement⟩ ⇒

  refinement ⟨refine⟩*****

⟨refine⟩ ⇒

  ⟨return type⟩⟨var id⟩**.**⟨var id⟩⟨params⟩ **:** ⟨statement⟩*****

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

⟨access group⟩ ⇒

  ⟨access type⟩ **:** ⟨member⟩*****

⟨access type⟩ ⇒

  private

  | **protected**

  | **public**

⟨member⟩ ⇒

  ⟨var decl⟩

  | ⟨method⟩

  | ⟨init⟩

⟨method⟩ ⇒

  ⟨return type⟩⟨var id⟩⟨params⟩ **:** ⟨statement⟩*****

⟨init⟩ ⇒

  init ⟨params⟩ **:** ⟨statement⟩*****

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

⟨main⟩ ⇒

  main (System ¡var id¿, String[] ⟨var id⟩ **) :** ⟨statement⟩*****

• *Finally the meat and potatoes*

⟨statement⟩ ⇒

$\langle$var decl$\rangle$
| $\langle$var decl$\rangle$ **:=** $\langle$expression$\rangle$
| $\langle$super$\rangle$
| $\langle$return$\rangle$
| $\langle$conditional$\rangle$
| $\langle$loop$\rangle$
| $\langle$expression$\rangle$

• *Super invocation is so we can do constructor chaining*
$\langle$super$\rangle \Rightarrow$
super $\langle$args$\rangle$

• *Methods yield values (or just exit for void/init/main)*
$\langle$return$\rangle \Rightarrow$
return
| **return** $\langle$expression$\rangle$

• *Basic control structures*
$\langle$conditional$\rangle \Rightarrow$
if ($\langle$expression$\rangle$ **) :** $\langle$statement$\rangle$***** $\langle$else$\rangle$
$\langle$else$\rangle \Rightarrow$
ϵ
| $\langle$elseif$\rangle$ **else :** $\langle$statement$\rangle$*****
$\langle$elseif$\rangle \Rightarrow$
ϵ
| $\langle$elseif$\rangle$ **elsif (** $\langle$expression$\rangle$ **) :** $\langle$statement$\rangle$*****
$\langle$loop$\rangle \Rightarrow$
while ($\langle$expression$\rangle$ **) :** $\langle$statement$\rangle$*****

• *Anything that can result in a value*
$\langle$expression$\rangle \Rightarrow$
$\langle$assignment$\rangle$
| $\langle$invocation$\rangle$
| $\langle$field$\rangle$
| $\langle$var id$\rangle$
| $\langle$deref$\rangle$
| $\langle$arithmetic$\rangle$
| $\langle$test$\rangle$
| $\langle$instantiate$\rangle$
| $\langle$refine expr$\rangle$
| $\langle$literal$\rangle$
| **(** $\langle$expression$\rangle$ **)**
| **this**

• *Assignment – putting one thing in another*

⟨assignment⟩ ⇒
  ⟨expression⟩⟨assign op⟩⟨expression⟩
⟨assign op⟩ ⇒
  :=
 | **+=**
 | **-=**
 | ***=**
 | **/=**
 | **%=**
 | ˆ=

• *Member / data access*

⟨invocation⟩ ⇒
  ⟨expression⟩ **.** ⟨var id⟩⟨args⟩
 | ⟨var id⟩⟨args⟩
⟨field⟩ ⇒
  ⟨expression⟩ **.** ⟨var id⟩
⟨deref⟩ ⇒
  ⟨expression⟩ [⟨expression⟩]

• *Basic arithmetic can and will be done!*

⟨arithmetic⟩ ⇒
  ⟨expression⟩⟨bin op⟩⟨expression⟩
 | ⟨unary op⟩⟨expression⟩
⟨bin op⟩ ⇒
  +
 | **-**
 | *****
 | **/**
 | **%**
 | ˆ
⟨unary op⟩ ⇒
  -

• *Common boolean predicates*

⟨test⟩ ⇒
  ⟨expression⟩⟨bin pred⟩⟨expression⟩
 | ⟨unary pred⟩⟨expression⟩
 | **refinable (** ⟨var id⟩ **)**
⟨bin pred⟩ ⇒
  and
 | **or**
 | **xor**
 | **nand**

| **nor**
| <
| <=
| **=**
| <>
| **=/=**
| >=
| >
⟨unary pred⟩ ⇒
not

• *Making something*
⟨instantiate⟩ ⇒
new ⟨type⟩⟨args⟩⟨optional refinements⟩
⟨optional refinements⟩ ⇒
ϵ
| **{** ⟨refine⟩***** **}**

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

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

• *Params and args are as expected*
⟨params⟩ ⇒
( )
| **(** ⟨paramlist⟩ **)**
⟨paramlist⟩ ⇒
⟨var decl⟩
| ⟨paramlist⟩ **,** ⟨var decl⟩

⟨args⟩ ⇒
    ( )
  | **(** ⟨arglist⟩ **)**
⟨arglist⟩ ⇒
    ⟨expression⟩
  | ⟨arglist⟩ **,** ⟨expression⟩

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

⟨var decl⟩ ⇒
    ⟨type⟩⟨var id⟩
⟨return type⟩ ⇒
    void
  | ⟨type⟩
⟨type⟩ ⇒
    ⟨class id⟩
  | ⟨type⟩[]
⟨class id⟩ ⇒
    ⟨upper⟩⟨ualphanum⟩*****
⟨var id⟩ ⇒
    ⟨lower⟩⟨ualphanum⟩*****

4 Project Plan

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 consensis.

Development was heavily facilited 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 farely 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.

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.

```
let get_x =
    ...
    let n = 2 in
    let x =
        x_functor1 (x_functor2 y z) n in
    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.)

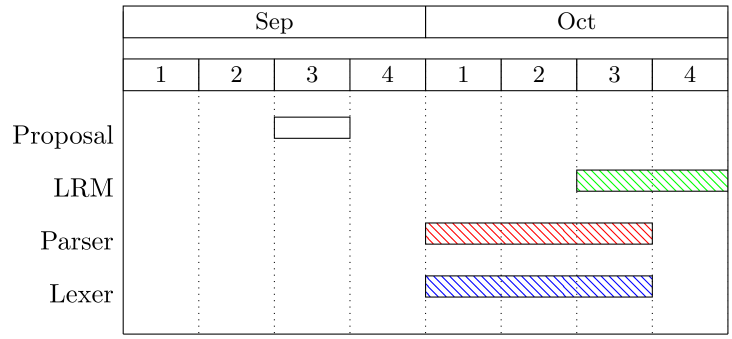
```
let unify_it var =
    match var with
    | X(y)      ->   y                              (* pop out *)
    | Y(y) :: _ ->   to_X y                         (* convert *)
    | Z(y)      ->
        to_X (to_Y (List.hd (List.rest y)))         (* mangle *)
```

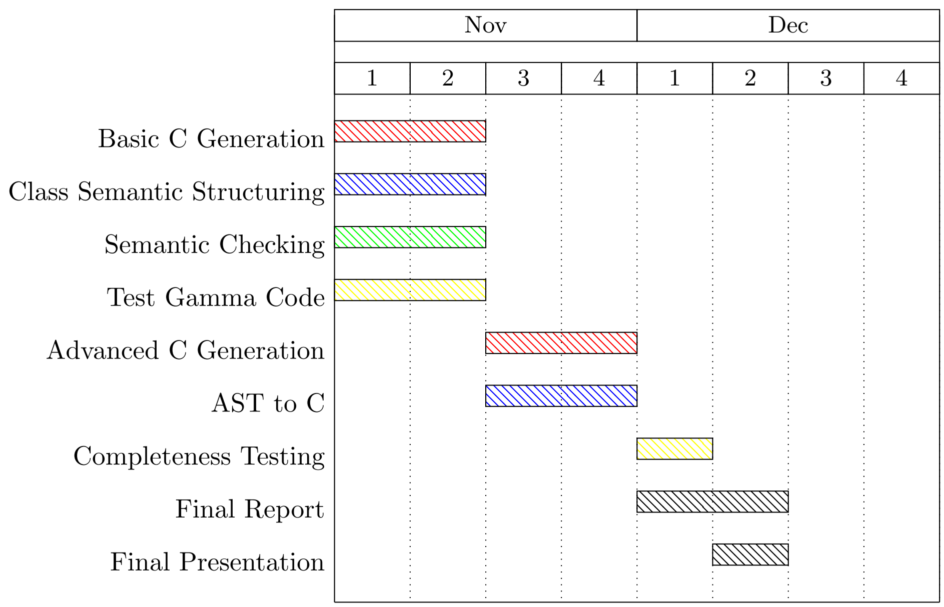
All records should maintain a basic standard of alignement and indentation for readibility. (Field names, colons, and type specs should be aligned to like.)

```
type person = {
    names  : string list;
    job    : string option;  (* Not everybody has one *)
    family : person list;
    female : bool;
    age    : int;
}
```

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 Atlassan 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 scanner output

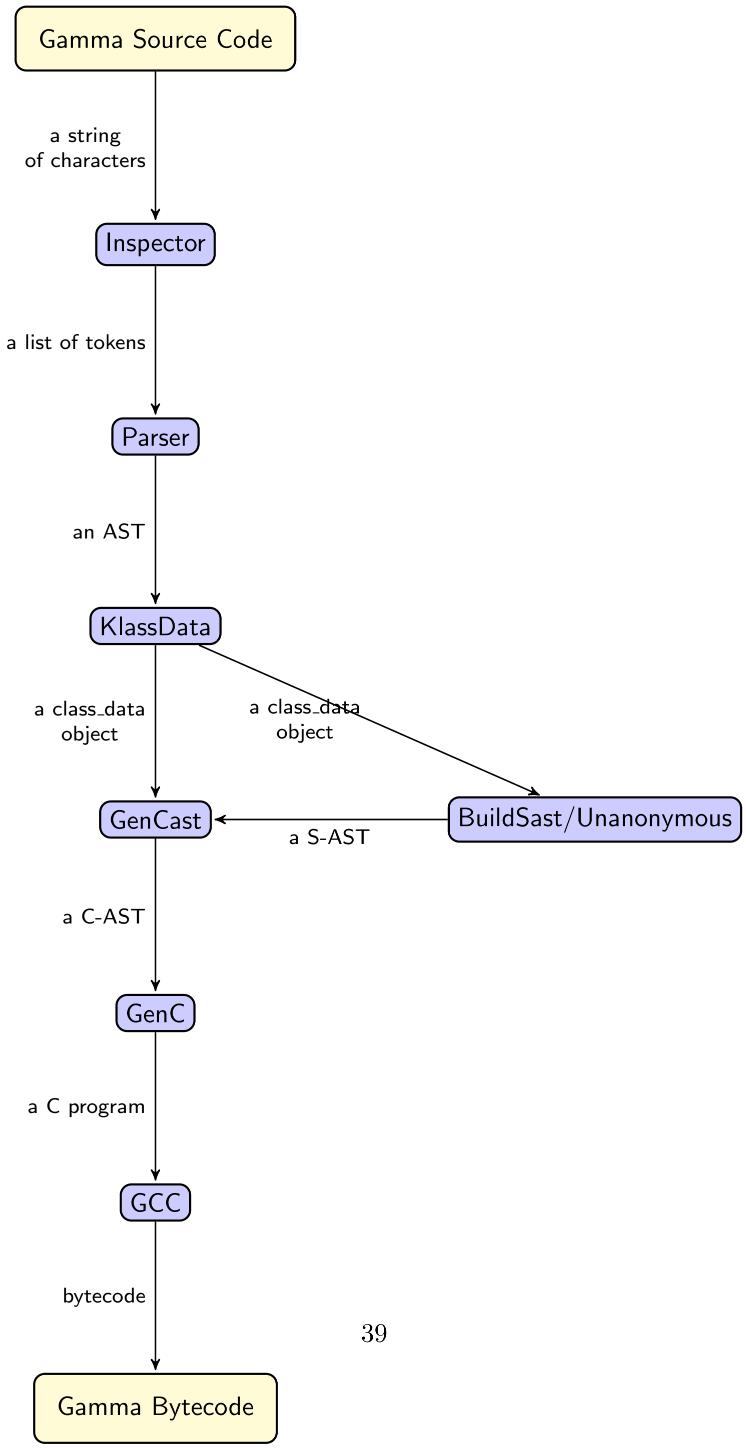
4.6 Project Log

- September 9th - Team Formed
- September 18th - Proposal drafting begins
- September 19th - A consensis 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

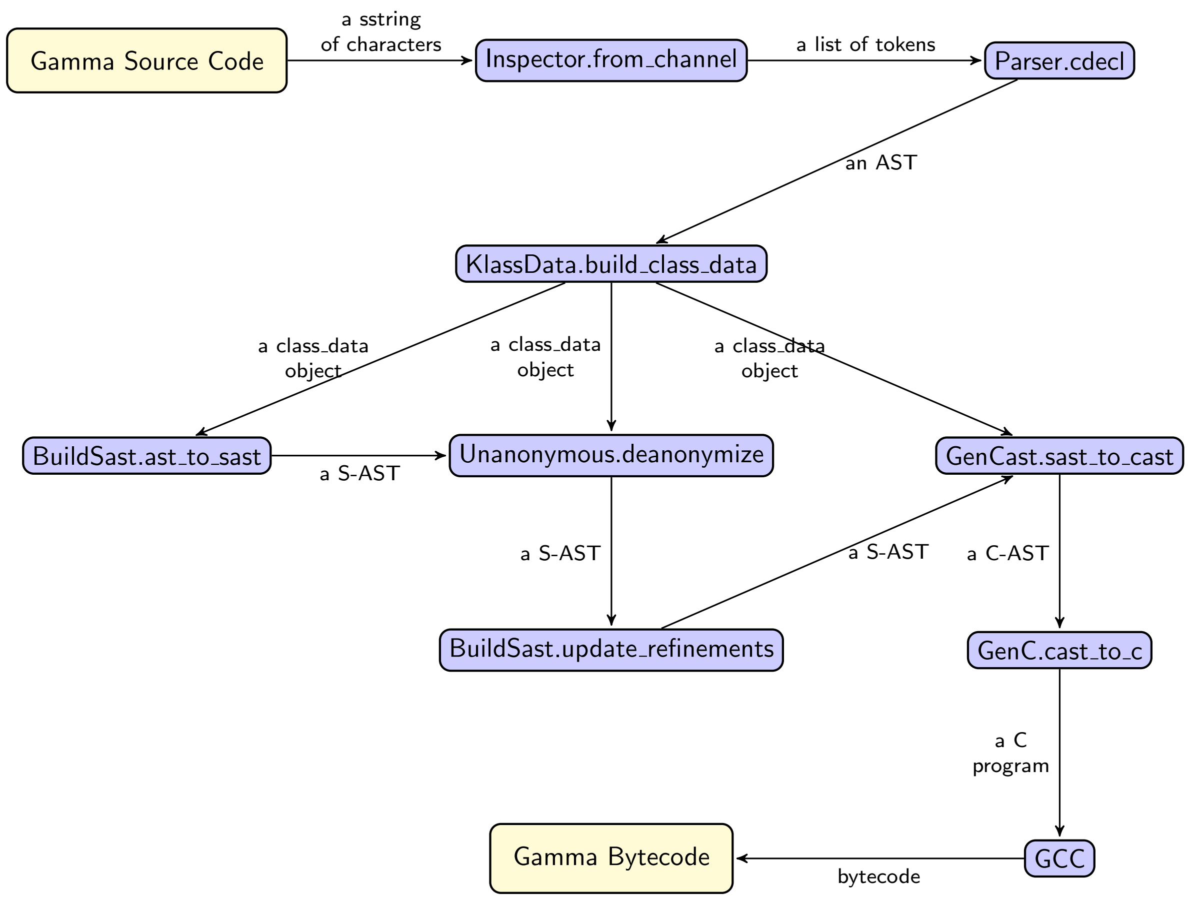
5 Architectural Design

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_deanom`. 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
- Unanonymous - 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.

```
class HelloWorld:
  public:
    String greeting
    init():
      super()
      greeting := "Hello World!"

  main(System system, String[] args):
    HelloWorld hw := new HelloWorld()
    system.out.printString(hw.greeting)
    system.out.printString("\n")
```

Example 14: ”Hello World in Gamma”

```
/* Starting Build Process...
 * Reading Tokens...
 * Parsing Tokens...
 * Generating Global Data...
 * Using Normal KlassData Builder
 * Building Semantic AST...
 * Deanonymizing Anonymous Classes.
 * Rebinding refinements.
 * Generating C AST...
 * Generating C...
 */


/*
 * Passing over code to find dispatch data.
 */


/*
 * Gamma preamble -- macros and such needed by various things
 */
#include "gamma-preamble.h"



/*
 * Ancestry meta-info to link to later.
 */
char *m_classes[] = {
```

```
    "t_Boolean", "t_Float", "t_HelloWorld", "t_Integer", "t_Object", "t_Printer",
    "t_Scanner", "t_String", "t_System"
};


/*
 * Enums used to reference into ancestry meta-info strings.
 */
enum m_class_idx {
    T_BOOLEAN = 0, T_FLOAT, T_HELLOWORLD, T_INTEGER, T_OBJECT, T_PRINTER, T_SCANNER,
    T_STRING, T_SYSTEM
};


/*
 * Header file containing meta information for built in classes.
 */
#include "gamma-builtin-meta.h"



/*
 * Meta structures for each class.
 */
ClassInfo M_HelloWorld;

void init_class_infos() {
    init_built_in_infos();
    class_info_init(&M_HelloWorld, 2, m_classes[T_OBJECT], m_classes[T_HELLOWORLD]);
}



/*
 * Header file containing structure information for built in classes.
 */
#include "gamma-builtin-struct.h"



/*
 * Structures for each of the objects.
 */
struct t_HelloWorld {
    ClassInfo *meta;

    struct {
        struct t_System *v_system;
    } Object;


    struct {
        struct t_String *v_greeting;
```

```c
    } HelloWorld;

};




/*
 * Header file containing information regarding built in
     functions.
 */
#include "gamma-builtin-functions.h"



/*
 * All of the function prototypes we need to do magic.
 */
struct t_HelloWorld *f_00000001_init(struct t_HelloWorld *);
void f_00000002_main(struct t_System *, struct t_String **);


/*
 * All the dispatching functions we need to continue the magic.
 */


/*
 * Array allocators also do magic.
 */


/*
 * All of the functions we need to run the program.
 */
/* Place-holder for struct t_Boolean *boolean_init(struct
     t_Boolean *this) */
/* Place-holder for struct t_Float *float_init(struct t_Float *
     this) */
/* Place-holder for struct t_Integer *float_to_i(struct t_Float
     *this) */
/* Place-holder for struct t_Integer *integer_init(struct
     t_Integer *this) */
/* Place-holder for struct t_Float *integer_to_f(struct
     t_Integer *this) */
/* Place-holder for struct t_Object *object_init(struct t_Object
     *this) */
/* Place-holder for struct t_Printer *printer_init(struct
     t_Printer *this, struct t_Boolean *v_stdout) */
/* Place-holder for void printer_print_float(struct t_Printer *
     this, struct t_Float *v_arg) */
/* Place-holder for void printer_print_integer(struct t_Printer
     *this, struct t_Integer *v_arg) */
/* Place-holder for void printer_print_string(struct t_Printer *
     this, struct t_String *v_arg) */
/* Place-holder for struct t_Scanner *scanner_init(struct
     t_Scanner *this) */
```

```c
/* Place-holder for struct t_Float *scanner_scan_float(struct t_Scanner *this) */
/* Place-holder for struct t_Integer *scanner_scan_integer(struct t_Scanner *this) */
/* Place-holder for struct t_String *scanner_scan_string(struct t_Scanner *this) */
/* Place-holder for struct t_String *string_init(struct t_String *this) */
/* Place-holder for void system_exit(struct t_System *this, struct t_Integer *v_code) */
/* Place-holder for struct t_System *system_init(struct t_System *this) */

struct t_HelloWorld *f_00000001_init(struct t_HelloWorld *this)
{
    object_init((struct t_Object *)(this));
    ( (this->HelloWorld).v_greeting = ((struct t_String *)(LIT_STRING("Hello World!"))) );
    return ( this );
}


void f_00000002_main(struct t_System *v_system, struct t_String **v_args)
{
    struct t_HelloWorld *v_hw = ((struct t_HelloWorld *)(f_00000001_init(MAKE_NEW(HelloWorld))));
    ( printer_print_string(((struct t_Printer *)((v_system)->System.v_out)), (v_hw)->HelloWorld.v_greeting) );
    ( printer_print_string(((struct t_Printer *)((v_system)->System.v_out)), LIT_STRING("\n")) );
}



/*
 * Dispatch looks like this.
 */


/*
 * Array allocators.
 */


/*
 * The main.
 */
#define CASES "HelloWorld"

int main(int argc, char **argv) {
    INIT_MAIN(CASES)
    if (!strncmp(gmain, "HelloWorld", 11)) { f_00000002_main(&global_system, str_args); return 0; }
    FAIL_MAIN(CASES)
    return 1;
}
```

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.)

```
class IOTest:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := promptInteger("Please enter an integer")
      Float f := promptFloat("Please enter a float")
      p.printString("Sum of integer + float = ")
      p.printFloat(i.toF() + f)
      p.printString("\n")

  private:
    void prompt(String msg):
      system.out.printString(msg)
      system.out.printString(": ")

    Integer promptInteger(String msg):
      prompt(msg)
      return system.in.scanInteger()

    Float promptFloat(String msg):
      prompt(msg)
      return system.in.scanFloat()

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

Example 16: "I/O in Gamma"

```
/* Starting Build Process...
 * Reading Tokens...
 * Parsing Tokens...
 * Generating Global Data...
 * Using Normal KlassData Builder
 * Building Semantic AST...
 * Deanonymizing Anonymous Classes.
 * Rebinding refinements.
 * Generating C AST...
 * Generating C...
```

```c
 */


/*
 * Passing over code to find dispatch data.
 */


/*
 * Gamma preamble -- macros and such needed by various things
 */
#include "gamma-preamble.h"



/*
 * Ancestry meta-info to link to later.
 */
char *m_classes[] = {
    "t_Boolean", "t_Float", "t_IOTest", "t_Integer", "t_Object",
     "t_Printer", "t_Scanner",
    "t_String", "t_System"
};


/*
 * Enums used to reference into ancestry meta-info strings.
 */
enum m_class_idx {
    T_BOOLEAN = 0, T_FLOAT, T_IOTEST, T_INTEGER, T_OBJECT,
    T_PRINTER, T_SCANNER,
    T_STRING, T_SYSTEM
};


/*
 * Header file containing meta information for built in classes.
 */
#include "gamma-builtin-meta.h"



/*
 * Meta structures for each class.
 */
ClassInfo M_IOTest;

void init_class_infos() {
    init_built_in_infos();
    class_info_init(&M_IOTest, 2, m_classes[T_OBJECT], m_classes
    [T_IOTEST]);
}



/*
```

```
 * Header file containing structure information for built in
   classes.
 */
#include "gamma-builtin-struct.h"



/*
 * Structures for each of the objects.
 */
struct t_IOTest {
    ClassInfo *meta;

    struct {
        struct t_System *v_system;
    } Object;


    struct { BYTE empty_vars; } IOTest;
};




/*
 * Header file containing information regarding built in
   functions.
 */
#include "gamma-builtin-functions.h"



/*
 * All of the function prototypes we need to do magic.
 */
struct t_IOTest *f_00000001_init(struct t_IOTest *);
void f_00000002_interact(struct t_IOTest *);
void f_00000003_prompt(struct t_IOTest *, struct t_String *);
struct t_Integer *f_00000004_promptInteger(struct t_IOTest *,
    struct t_String *);
struct t_Float *f_00000005_promptFloat(struct t_IOTest *, struct
     t_String *);
void f_00000006_main(struct t_System *, struct t_String **);


/*
 * All the dispatching functions we need to continue the magic.
 */


/*
 * Array allocators also do magic.
 */


/*
 * All of the functions we need to run the program.
```

```c
 */
/* Place-holder for struct t_Boolean *boolean_init(struct
    t_Boolean *this) */
/* Place-holder for struct t_Float *float_init(struct t_Float *
    this) */
/* Place-holder for struct t_Integer *float_to_i(struct t_Float
    *this) */
/* Place-holder for struct t_Integer *integer_init(struct
    t_Integer *this) */
/* Place-holder for struct t_Float *integer_to_f(struct
    t_Integer *this) */
/* Place-holder for struct t_Object *object_init(struct t_Object
     *this) */
/* Place-holder for struct t_Printer *printer_init(struct
    t_Printer *this, struct t_Boolean *v_stdout) */
/* Place-holder for void printer_print_float(struct t_Printer *
    this, struct t_Float *v_arg) */
/* Place-holder for void printer_print_integer(struct t_Printer
    *this, struct t_Integer *v_arg) */
/* Place-holder for void printer_print_string(struct t_Printer *
    this, struct t_String *v_arg) */
/* Place-holder for struct t_Scanner *scanner_init(struct
    t_Scanner *this) */
/* Place-holder for struct t_Float *scanner_scan_float(struct
    t_Scanner *this) */
/* Place-holder for struct t_Integer *scanner_scan_integer(
    struct t_Scanner *this) */
/* Place-holder for struct t_String *scanner_scan_string(struct
    t_Scanner *this) */
/* Place-holder for struct t_String *string_init(struct t_String
     *this) */
/* Place-holder for void system_exit(struct t_System *this,
    struct t_Integer *v_code) */
/* Place-holder for struct t_System *system_init(struct t_System
     *this) */

struct t_IOTest *f_00000001_init(struct t_IOTest *this)
{
    object_init((struct t_Object *)(this));
    return ( this );
}


void f_00000002_interact(struct t_IOTest *this)
{
    struct t_Printer *v_p = ((struct t_Printer *)(((this->Object
    ).v_system)->System.v_out));
    struct t_Integer *v_i = ((struct t_Integer *)(
    f_00000004_promptInteger(((struct t_IOTest *)(this)),
    LIT_STRING("Please enter an integer"))));
    struct t_Float *v_f = ((struct t_Float *)(
    f_00000005_promptFloat(((struct t_IOTest *)(this)),
    LIT_STRING("Please enter a float"))));
    ( printer_print_string(((struct t_Printer *)(v_p)),
    LIT_STRING("Sum of integer + float = ")) );
    ( printer_print_float(((struct t_Printer *)(v_p)),
    ADD_FLOAT_FLOAT( integer_to_f(((struct t_Integer *)(v_i))) ,
```

```
    v_f )) );
    ( printer_print_string(((struct t_Printer *)(v_p)),
    LIT_STRING("\n")) );
}


void f_00000003_prompt(struct t_IOTest *this, struct t_String *
    v_msg)
{
    ( printer_print_string(((struct t_Printer *)(((this->Object)
    .v_system)->System.v_out)), v_msg) );
    ( printer_print_string(((struct t_Printer *)(((this->Object)
    .v_system)->System.v_out)), LIT_STRING(": ")) );
}


struct t_Integer *f_00000004_promptInteger(struct t_IOTest *this
    , struct t_String *v_msg)
{
    ( f_00000003_prompt(((struct t_IOTest *)(this)), v_msg) );
    return ( scanner_scan_integer(((struct t_Scanner *)(((this->
    Object).v_system)->System.v_in))) );
}


struct t_Float *f_00000005_promptFloat(struct t_IOTest *this,
    struct t_String *v_msg)
{
    ( f_00000003_prompt(((struct t_IOTest *)(this)), v_msg) );
    return ( scanner_scan_float(((struct t_Scanner *)(((this->
    Object).v_system)->System.v_in))) );
}


void f_00000006_main(struct t_System *v_system, struct t_String
    **v_args)
{
    struct t_IOTest *v_test = ((struct t_IOTest *)(
    f_00000001_init(MAKE_NEW(IOTest))));
    ( f_00000002_interact(((struct t_IOTest *)(v_test))) );
}



/*
 * Dispatch looks like this.
 */


/*
 * Array allocators.
 */


/*
 * The main.
 */
```

```
#define CASES "IOTest"

int main(int argc, char **argv) {
    INIT_MAIN(CASES)
    if (!strncmp(gmain, "IOTest", 7)) { f_00000006_main(&
    global_system, str_args); return 0; }
    FAIL_MAIN(CASES)
    return 1;
}
```

Example 17: "I/O in Compiled C"

6.1.3 Argument Reading

This program prints out each argument passed to the program.

```
class Test:
  public:
    init():
      super()

  main(System sys, String[] args):
    Integer i := 0
    Printer p := sys.out

    while (i < sys.argc):
      p.printString("arg[")
      p.printInteger(i)
      p.printString("] = ")
      p.printString(args[i])
      p.printString("\n")
      i += 1
```

Example 18: "Argument Reading in Gamma"

```
/* Starting Build Process...
 * Reading Tokens...
 * Parsing Tokens...
 * Generating Global Data...
 * Using Normal KlassData Builder
 * Building Semantic AST...
 * Deanonymizing Anonymous Classes.
 * Rebinding refinements.
 * Generating C AST...
 * Generating C...
 */


/*
 * Passing over code to find dispatch data.
 */

```

```
/*
 * Gamma preamble -- macros and such needed by various things
 */
#include "gamma-preamble.h"



/*
 * Ancestry meta-info to link to later.
 */
char *m_classes[] = {
    "t_Boolean", "t_Float", "t_Integer", "t_Object", "t_Printer", "t_Scanner",
    "t_String", "t_System", "t_Test"
};


/*
 * Enums used to reference into ancestry meta-info strings.
 */
enum m_class_idx {
    T_BOOLEAN = 0, T_FLOAT, T_INTEGER, T_OBJECT, T_PRINTER, T_SCANNER, T_STRING,
    T_SYSTEM, T_TEST
};


/*
 * Header file containing meta information for built in classes.
 */
#include "gamma-builtin-meta.h"



/*
 * Meta structures for each class.
 */
ClassInfo M_Test;

void init_class_infos() {
    init_built_in_infos();
    class_info_init(&M_Test, 2, m_classes[T_OBJECT], m_classes[T_TEST]);
}



/*
 * Header file containing structure information for built in classes.
 */
#include "gamma-builtin-struct.h"



/*
```

```
 * Structures for each of the objects.
 */
struct t_Test {
    ClassInfo *meta;

    struct {
        struct t_System *v_system;
    } Object;


    struct { BYTE empty_vars; } Test;
};




/*
 * Header file containing information regarding built in
     functions.
 */
#include "gamma-builtin-functions.h"



/*
 * All of the function prototypes we need to do magic.
 */
struct t_Test *f_00000001_init(struct t_Test *);
void f_00000002_main(struct t_System *, struct t_String **);


/*
 * All the dispatching functions we need to continue the magic.
 */


/*
 * Array allocators also do magic.
 */


/*
 * All of the functions we need to run the program.
 */
/* Place-holder for struct t_Boolean *boolean_init(struct
     t_Boolean *this) */
/* Place-holder for struct t_Float *float_init(struct t_Float *
     this) */
/* Place-holder for struct t_Integer *float_to_i(struct t_Float
     *this) */
/* Place-holder for struct t_Integer *integer_init(struct
     t_Integer *this) */
/* Place-holder for struct t_Float *integer_to_f(struct
     t_Integer *this) */
/* Place-holder for struct t_Object *object_init(struct t_Object
     *this) */
```

```c
/* Place-holder for struct t_Printer *printer_init(struct t_Printer *this, struct t_Boolean *v_stdout) */
/* Place-holder for void printer_print_float(struct t_Printer *this, struct t_Float *v_arg) */
/* Place-holder for void printer_print_integer(struct t_Printer *this, struct t_Integer *v_arg) */
/* Place-holder for void printer_print_string(struct t_Printer *this, struct t_String *v_arg) */
/* Place-holder for struct t_Scanner *scanner_init(struct t_Scanner *this) */
/* Place-holder for struct t_Float *scanner_scan_float(struct t_Scanner *this) */
/* Place-holder for struct t_Integer *scanner_scan_integer(struct t_Scanner *this) */
/* Place-holder for struct t_String *scanner_scan_string(struct t_Scanner *this) */
/* Place-holder for struct t_String *string_init(struct t_String *this) */
/* Place-holder for void system_exit(struct t_System *this, struct t_Integer *v_code) */
/* Place-holder for struct t_System *system_init(struct t_System *this) */

struct t_Test *f_00000001_init(struct t_Test *this)
{
    object_init((struct t_Object *)(this));
    return ( this );
}


void f_00000002_main(struct t_System *v_sys, struct t_String **v_args)
{
    struct t_Integer *v_i = ((struct t_Integer *)(LIT_INT(0)));
    struct t_Printer *v_p = ((struct t_Printer *)((v_sys)->System.v_out));
    while ( BOOL_OF( NTEST_LESS_INT_INT( v_i , (v_sys)->System.v_argc ) ) ) {
        ( printer_print_string(((struct t_Printer *)(v_p)), LIT_STRING("arg[")) );
        ( printer_print_integer(((struct t_Printer *)(v_p)), v_i ) );
        ( printer_print_string(((struct t_Printer *)(v_p)), LIT_STRING("] = ")) );
        ( printer_print_string(((struct t_Printer *)(v_p)), ((struct t_String **)(v_args))[INTEGER_OF((v_i))]) );
        ( printer_print_string(((struct t_Printer *)(v_p)), LIT_STRING("\n")) );
        ( v_i = ((struct t_Integer *)(ADD_INT_INT( v_i , LIT_INT(1) ))) );
    }
}



/*
 * Dispatch looks like this.
```

```
 */


/*
 * Array allocators.
 */


/*
 * The main.
 */
#define CASES "Test"

int main(int argc, char **argv) {
    INIT_MAIN(CASES)
    if (!strncmp(gmain, "Test", 5)) { f_00000002_main(&
    global_system, str_args); return 0; }
    FAIL_MAIN(CASES)
    return 1;
}
```

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.

```
class Parent:
  public:
    init():
      super()

class Child extends Parent:
  public:
    init():
      super()

class Test:
  public:
    init():
      super()

  main(System system, String[] args):
    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.

```
class BadDecl:
  public:
    init():
      super()
      Integer a := 3.4
```

Test Source 2: "Improper Variable Declaration/Assignment"

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

```
class Test:
  public:
    Float a
    Float b
    Integer c

    init():
      super()
```

```
      a := 1.5
      b := 2.2
      c := 3

    Float overview():
      Float success := a+b+c
      return success

  main(System system, String[] args):
    Test ab := new Test()
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printFloat(ab.overview())
    p.printString("\n")
```

Test Source 3: "Binary Operations Between Incompatible Types"

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

```
class BadReturn:
  public:
    init():
      super()

    Integer badReturn():
      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.

```
class BadReturn:
  public:
    init():
      super()

    Integer badReturn():
      return
```

Test Source 5: "Empty Return Statement"

A return statement should return something.

```
class BadReturn:
  public:
    init():
      super()

    void badReturn():
      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.

```
class BadAssign:
  public:
    init():
      super()
      Integer a
      a := 3.4
```

Test Source 7: "Improper Literal Assignment"

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

```
class BadStatic:
  public:
    Integer getZero():
      return 0
    init():
      super()
  main(System system, String[] args):
    getZero()  /* This is supposed to fail. DON'T CHANGE */
```

Test Source 8: "Static Method Calls"

A method must be called on an object.

```
class Parent:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 0

    Integer overview():
      Integer success := refine toExtra(a,b) to Integer
      return success

class Child extends Parent:
  refinement:
    Integer overview.toExtra(Integer a, Integer b):
      Integer success :=  a + b
      Printer p := new Printer(true)
      p.printInteger(a)
      p.printInteger(b)
```

```
      p.printInteger(c)
      return success
  public:
    Integer a1
    Integer b1
    Integer c1

    init():
      super()
      a1 := 1
      b1 := 2
      c1 := 0

class Test:
  public:
    init():
      super()

  main(System system, String[] args):
    Parent ab := new Parent
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printInteger(ab.overview())
    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.

```
class Parent:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 0

    Integer overview():
      Integer success := -1
      if (refinable(toExtra)) {
          success := refine toExtra(a,b) to Integer;
      }
      return success

class Child extends Parent:
  refinement:
    Integer overview.toExtra(Integer a, Integer b):
      Integer success :=  a + b
      Printer p := new Printer(true)
      p.printInteger(a)
```

```
      p.printInteger(b)
      p.printInteger(c)
      return success
  public:
    Integer a1
    Integer b1
    Integer c1

    init():
      super()
      a1 := 1
      b1 := 2
      c1 := 0

class Test:
  public:
    init():
      super()

  main(System system, String[] args):
    Parent ab := new Parent()
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printInteger(ab.overview())
    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.

```

class WhileLoopTest:
  public:
    init():
      super()
      Integer a := 0
      while((a>=0) and (a<10)):
        system.out.printInteger(a)
        system.out.printString("\n")
        a := a + 1

  main(System system, String[] args):
    new WhileLoopTest()
```

Test Source 11: "Conditioned While Statements"

This test makes sure while loops function.

```

class WhileLoopTest:
  public:
    init():
      super()
      Integer a := 0
      while(true):
        system.out.printInteger(a)
        system.out.printString("\n")
        a := a + 1

  main(System system, String[] args):
    new WhileLoopTest()
```

Test Source 12: "Infinite While Statement"

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

```
class IfTest:
  private:
    void line():
      system.out.printString("\n")

    void out(String msg):
      system.out.printString(msg)
      line()

    void yes():
      out("This should print.")
    void no():
      out("This should not print.")

  public:
    init():
      super()

      out("Simple (1/2)")
      if (true)  { yes(); }
      if (false) { no();  }
      line()

      out("Basic (2/2)")
      if (true)  { yes(); } else { no();  }
      if (false) { no();  } else { yes(); }
      line()

      out("Multiple (3/3)")
      if (true)  { yes(); } elsif (false) { no();  } else { no();  }
      if (false) { no();  } elsif (true)  { yes(); } else { no();  }
      if (false) { no();  } elsif (false) { no();  } else { yes(); }
      line()
```

```

      out("Non-exhaustive (2/3)")
      if (true)  { yes(); } elsif (false) { no();  }
      if (false) { no();  } elsif (true)  { yes(); }
      if (false) { no();  } elsif (false) { no();  }

  main(System system, String[] args):
    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.

```
class Test:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 3

    Integer overview():
      Integer success := a+b
      return success

  main(System system, String[] args):
    Test ab := new Test()
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printInteger(ab.overview())
    p.printString("\n")
```

Test Source 14: "Add Integers"

```
class Test:
  public:
    Float a
    Float b
    Integer c

    init():
      super()
      a := 1.5
      b := 2.2
```

```
      c := 0

    Float overview():
      Float success := a+b
      return success

  main(System system, String[] args):
    Test ab := new Test()
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printFloat(ab.overview())
    p.printString("\n")
```

Test Source 15: "Add Floats"

These tests add numeric literal objects together.

```
class Test:
  public:
    Integer a
    Float   b

    init():
      super()

    Integer add():
      a := 10 * 2 * 9
      b := 6.0 * 0.5 * (-2.0)
      return 0

  main(System sys, String[] args):
```

Test Source 16: "Multiplication"

```
class Test:
  public:
    Integer a
    Float   b

    init():
      super()

    Integer add():
      a := (10 / 5) / -2
      b := (10.0 / 5.0) / -2.0
      return 0

  main(System sys, String[] args):
    Test t := new Test()
    Printer p := sys.out

    t.add()
    p.printString("A is ")
    p.printInteger(t.a)
```

```
    p.printString(", B is ")
    p.printFloat(t.b)
    p.printString("\n")
```

Test Source 17: "Divition"

These tests form products/quotions of Floats/Integers.

```
class Test:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 3

    Integer overview():
      Integer success := a%b
      return success

  main(System system, String[] args):
    Test ab := new Test()
    Printer p := system.out
    p.printString(" 1 % 2 = ")
    p.printInteger(ab.overview())
    p.printString("\n")
```

Test Source 18: "Modulus"

This test forms the modulus of Integers.

```
class Test:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := 5
      Float f := 1.5
      p.printString("Sum of integer + float = ")
      p.printFloat(i.toF() + f)
      p.printString("\n")

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

Test Source 19: "Literal Casting and Addition"

```
class Test:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := 5
      Float f := 1.5
      p.printString("integer - float = ")
      p.printFloat(i.toF() - f)
      p.printString("\n")

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

Test Source 20: "Literal Casting and Subtraction"

```
class Test:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := 5
      Float f := 1.5
      p.printString("integer * float = ")
      p.printFloat(i.toF() * f)
      p.printString("\n")

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

Test Source 21: "Literal Casting and Multiplication"

```
class Test:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := 5
      Float f := 1.5
      p.printString("float/Integer = ")
      p.printFloat(f/i.toF())
      p.printString("\n")

  main(System system, String[] args):
```

```
    Test test := new Test()
    test.interact()
```

Test Source 22: ”Literal Casting and Divition”

```
class Test:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := 5
      Float f := 1.5
      p.printString("integer ^ float = ")
      p.printFloat(i.toF() ^ f)
      p.printString("\n")

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

Test Source 23: ”Literal Casting and Exponentiation”

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

```
class Parent:
  public:
    init():
      super()

class Child extends Parent:
  public:
    init():
      super()

class Test:
  public:
    init():
      super()

  main(System system, String[] args):
    Parent child := new Child()
```

Test Source 24: ”Superclass Typing”

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

```
class Test:
  private:
    void line():
```

```
      system.out.printString("\n")

    void out(String msg):
      system.out.printString(msg)
      line()

  public:
    init():
      super()
      Integer a:=2
      Integer b:=3
      Integer c

      /* less and less and equal*/
      if (a<2)  { system.out.printString("1. a=2 a<2 shouldnot print\n"); }
      elsif (a<=2)  { system.out.printString("1. a=2 a<=2 success\n"); }
      else { system.out.printString("1. should never hit here\n"); }


      /* greater and greater than equal */
      if (b>3)  { system.out.printString("2. b=3 b>3 shouldnot print\n"); }
      else  { system.out.printString("2. b=3 b>=3 success\n"); }

      /*Equal and not equal*/
      if (a <> b)  { system.out.printString("3. a!=b success \n"); }
      a:=b
      if (a=b)  { system.out.printString("4. a=b success\n"); }

      /*And or */
      if(a=3 and b=3) {  system.out.printString("5. a=3 and b=3 success\n"); }

      b:=5
      if(b=3 or a=3) {  system.out.printString("6. b=3 or a=3 success\n"); }

      /*nand and nor and not*/
      b:=4
      a:=4
      if(b=3 nor a=3) {  system.out.printString("7. b=10 nor a=10 success\n"); }
      if(not(b=4 nand a=4)) {  system.out.printString("8. not(b=4 nand a=4) success\n"); }
      b:=3
      if(b=4 nand a=4) {  system.out.printString("9. b=4 nand a=4 success\n"); }
      if(b=3 xor a=3) {  system.out.printString("10. b=3 xor a=3 success\n"); }
      c:=10
      if((a<>b or b=c) and c=10)  {  system.out.printString("11. (a<>b or b=c) and c=10 success\n"); }
      line()
```

```

  main(System system, String[] args):
    Test theif := new Test()
```

Test Source 25: "Boolean Comparison"

This test performs boolean comparisons between numeric literal objects.

```

class Person:
  protected:
    String name

  public:
    init(String name):
      super()
      this.name := name

    void introduce():
      Printer p := system.out
      p.printString("Hello, my name is ")
      p.printString(name)
      p.printString(", and I am from ")
      p.printString(refine origin() to String)
      p.printString(". I am ")
      p.printInteger(refine age() to Integer)
      p.printString(" years old. My occupation is ")
      p.printString(refine work() to String)
      p.printString(". It was nice meeting you.\n")

class Test:
  protected:
    init():
      super()

  main(System sys, String[] args):
    (new Person("Matthew") {
      String introduce.origin() { return "New Jersey"; }
      Integer introduce.age() { return 33; }
      String introduce.work() { return "Student"; }
    }).introduce()

    (new Person("Arthy") {
      String introduce.origin() { return "India"; }
      Integer introduce.age() { return 57; }
      String introduce.work() { return "Student"; }
    }).introduce()

    (new Person("Weiyuan") {
      String introduce.origin() { return "China"; }
      Integer introduce.age() { return 24; }
      String introduce.work() { return "Student"; }
    }).introduce()

    (new Person("Ben") {
      String introduce.origin() { return "New York"; }
```

```
      Integer introduce.age() { return 24; }
      String introduce.work() { return "Student"; }
    }).introduce()
```

Test Source 26: "Anonymous objects"

This tests forms anonymous objects.

```
class Test:
  private:
    void print(Integer i):
      Printer p := system.out
      p.printString("a[")
      p.printInteger(i)
      p.printString("] = ")
      p.printInteger(a[i])
      p.printString("\n")

  public:
    Integer[] a
    init():
      super()
      a := new Integer[](4)
      a[0] := 3
      a[1] := 2
      a[2] := 1
      a[3] := 0

    void print():
      Integer i := 0
      while (i < 4):
        print(i)
        i += 1

  main(System system, String[] args):
    Test f
    f := new Test()
    f.print()
```

Test Source 27: "Arrays"

This test forms an array.

```
class Parent:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 0
```

```

    Integer overview():
      Integer success := refine toExtra(a,b) to Integer
      return success

class Child extends Parent:
  refinement:
    Integer overview.toExtra(Integer a, Integer b):
      Integer success :=  a + b
      Printer p := new Printer(true)
      p.printInteger(a)
      p.printInteger(b)
      p.printInteger(c)
      return success
  public:
    Integer a1
    Integer b1
    Integer c1

    init():
      super()
      a1 := 1
      b1 := 2
      c1 := 0

class Test:
  public:
    init():
      super()

  main(System system, String[] args):
    Parent ab := new Child()
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printInteger(ab.overview())
    p.printString("\n")
```

Test Source 28: "Refinement"

This test checks that basic refinement works.

```
class Parent:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 0

    Integer overview():
      Integer success := -1
      if (refinable(toExtra)) {
```

```
          success := refine toExtra(a,b) to Integer;
        }
        return success

class Child extends Parent:
  refinement:
    Integer overview.toExtra(Integer a, Integer b):
      Integer success :=  a + b
      Printer p := new Printer(true)
      p.printInteger(a)
      p.printInteger(b)
      p.printInteger(c)
      return success
  public:
    Integer a1
    Integer b1
    Integer c1

    init():
      super()
      a1 := 1
      b1 := 2
      c1 := 0

class Test:
  public:
    init():
      super()

  main(System system, String[] args):
    Parent ab := new Child()
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printInteger(ab.overview())
    p.printString("\n")
```

Test Source 29: "Refinable"

This test checks that the refinable keyword works.

```
class Parent:
  protected:
    Integer a
    Integer b
    String name

  public:
    init(String name):
      super()

      this.name := name
      a := 1
      b := 2

    void print():
      Printer p := system.out
```

```
      p.printString(name)
      p.printString(":  A is ")
      p.printInteger(a)
      p.printString(", B is ")
      p.printInteger(b)
      p.printString("\n")

    void update():
      if (refinable(setA)):
        a := refine setA() to Integer
      if (refinable(setB)):
        b := refine setB() to Integer

class Son extends Parent:
  public:
    init(String name):
      super(name)

  refinement:
    Integer update.setA():
      return -1
    Integer update.setB():
      return -2

class Daughter extends Parent:
  public:
    init(String name):
      super(name)

  refinement:
    Integer update.setA():
      return 10
    Integer update.setB():
      return -5


class Test:
  protected:
    init():
      super()

  main(System sys, String[] args):
    Parent pop := new Parent("Father")
    Son son := new Son("Son")
    Daughter daughter := new Daughter("Daughter")

    pop.print()
    son.print()
    daughter.print()
    sys.out.printString("------------------------\n")
    pop.update()
    son.update()
    daughter.update()

    pop.print()
    son.print()
    daughter.print()
```

```
```

Test Source 30: ”Refinements”

This test makes multiple trivial refinements.

6.2.4 Structure Testing

```
class MainTest:
  public:
    init():
      super()
  main(System system, String[] args):
    Integer a
    a := 0
    a += 1
```

Test Source 31: ”Main Method”

This test forms a main method

```
class Math:
  private:
    Float xyz
  public:
    init():
      super()
    Integer add(Integer a, Integer b):
      return 6
    Integer sub(Integer a, Integer c):
      return 4
  main(System sys, String[] args):

class NonMath:
  private:
    String shakespeare
  public:
    init():
      super()
    String recite():
      return "hey"
  main(System sys, String[] hey):
```

Test Source 32: ”Empty Bodies”

This test presents minimalistic bodies for a variety of methods.

```
class FuncTest:
  public:
      Integer a

```

```
      init():
          super()
          a := 1

  private:
      Integer incre_a(Integer b):
          a := a + b
          return a

      Integer incre_a_twice(Integer b):
          incre_a(b)
          incre_a(b)
          return a

  main(System system, String[] args):
      FuncTest test := new FuncTest()
```

Test Source 33: "Functions"

This test probes function scope.

6.2.5 A Complex Test

```
class IOTest:
  public:
    Integer a
    Integer b
    Integer c
    init():
      super()
      a := 1
      b := 2
      c := 0
    void overview():
      Printer p := new Printer(true)
      p.printInteger(a)
      p.printInteger(b)
      p.printInteger(c)
    Integer incre_ab():
      Scanner s := new Scanner()
      Integer delta
      delta := s.scanInteger()
      a := a + delta
      b := b + delta
      return c
    Integer arith():
      c := -(a + b)
      return c

class Main:
  public:
    init():
      super()
  main(String[] args):
```

```
    IOTest ab := new IOTest()
    ab.overview()
    ab.incre_ab()
    ab.overview()
    ab.arith()
    ab.overview()
```

Test Source 34: "Complex Scanning"

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

7 Lessons Learnt

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 tatooed 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 procastrination - 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

```
class IOTest:
  public:
    Integer a
    Integer b
    Integer c
    init():
      super()
      a := 1
      b := 2
      c := 0
    void overview():
      Printer p := new Printer(true)
      p.printInteger(a)
      p.printInteger(b)
      p.printInteger(c)
    Integer incre_ab():
      Scanner s := new Scanner()
      Integer delta
      delta := s.scanInteger()
      a := a + delta
      b := b + delta
      return c
    Integer arith():
      c := -(a + b)
      return c

class Main:
  public:
    init():
      super()
  main(String[] args):
    IOTest ab := new IOTest()
    ab.overview()
    ab.incre_ab()
    ab.overview()
    ab.arith()
    ab.overview()
```

Source 1: `compiler-tests/mix.gamma`

```
class IOTest:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := promptInteger("Please enter an integer")
      Float f := promptFloat("Please enter a float")
      p.printString("Sum of integer + float = ")
      p.printFloat(i.toF() + f)
```

```
      p.printString("\n")

  private:
    void prompt(String msg):
      system.out.printString(msg)
      system.out.printString(": ")

    Integer promptInteger(String msg):
      prompt(msg)
      return system.in.scanInteger()

    Float promptFloat(String msg):
      prompt(msg)
      return system.in.scanFloat()

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

Source 2: `compiler-tests/programs/io.gamma`

```
class HelloWorld:
  public:
    String greeting
    init():
      super()
      greeting := "Hello World!"

  main(System system, String[] args):
    HelloWorld hw := new HelloWorld()
    system.out.printString(hw.greeting)
    system.out.printString("\n")
```

Source 3: `compiler-tests/programs/helloworld.gamma`

```
class Test:
  public:
    init():
      super()

  main(System sys, String[] args):
    Integer i := 0
    Printer p := sys.out

    while (i < sys.argc):
      p.printString("arg[")
      p.printInteger(i)
      p.printString("] = ")
      p.printString(args[i])
      p.printString("\n")
      i += 1
```

Source 4: `compiler-tests/programs/args.gamma`

```
class Parent:
  public:
    init():
      super()

class Child extends Parent:
  public:
    init():
      super()

class Test:
  public:
    init():
      super()

  main(System system, String[] args):
    Child child := new Parent()
```

Source 5: `compiler-tests/bad/super-assign.gamma`

```
class BadDecl:
  public:
    init():
      super()
      Integer a := 3.4
```

Source 6: `compiler-tests/bad/decl.gamma`

```
class Test:
  public:
    Float a
    Float b
    Integer c

    init():
      super()
      a := 1.5
      b := 2.2
      c := 3

    Float overview():
      Float success := a+b+c
      return success

  main(System system, String[] args):
    Test ab := new Test()
    Printer p := system.out
```

```
    p.printString("Sum of integer = ")
    p.printFloat(ab.overview())
    p.printString("\n")
```

Source 7: `compiler-tests/bad/addMix.gamma`

```
class BadReturn:
  public:
    init():
      super()

    Integer badReturn():
      return "Hey There"
```

Source 8: `compiler-tests/bad/return1.gamma`

```
class BadAssign:
  public:
    init():
      super()
      Integer a
      a := 3.4
```

Source 9: `compiler-tests/bad/assign.gamma`

```
class BadStatic:
  public:
    Integer getZero():
      return 0
    init():
      super()
  main(System system, String[] args):
    getZero()  /* This is supposed to fail. DON'T CHANGE */
```

Source 10: `compiler-tests/bad/static.gamma`

```
class Parent:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 0

    Integer overview():
```

```
      Integer success := refine toExtra(a,b) to Integer
      return success

class Child extends Parent:
  refinement:
    Integer overview.toExtra(Integer a, Integer b):
      Integer success :=  a + b
      Printer p := new Printer(true)
      p.printInteger(a)
      p.printInteger(b)
      p.printInteger(c)
      return success
  public:
    Integer a1
    Integer b1
    Integer c1

    init():
      super()
      a1 := 1
      b1 := 2
      c1 := 0

class Test:
  public:
    init():
      super()

  main(System system, String[] args):
    Parent ab := new Parent
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printInteger(ab.overview())
    p.printString("\n")
```

Source 11: `compiler-tests/bad/refine_refinable.gamma`

```
class BadReturn:
  public:
    init():
      super()

    Integer badReturn():
      return
```

Source 12: `compiler-tests/bad/return2.gamma`

```
class BadReturn:
  public:
    init():
      super()

    void badReturn():
```

```
      return "Hey There"
```

Source 13: `compiler-tests/bad/return3.gamma`

```
class Parent:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 0

    Integer overview():
      Integer success := -1
      if (refinable(toExtra)) {
          success := refine toExtra(a,b) to Integer;
      }
      return success

class Child extends Parent:
  refinement:
    Integer overview.toExtra(Integer a, Integer b):
      Integer success :=  a + b
      Printer p := new Printer(true)
      p.printInteger(a)
      p.printInteger(b)
      p.printInteger(c)
      return success
  public:
    Integer a1
    Integer b1
    Integer c1

    init():
      super()
      a1 := 1
      b1 := 2
      c1 := 0

class Test:
  public:
    init():
      super()

  main(System system, String[] args):
    Parent ab := new Parent()
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printInteger(ab.overview())
    p.printString("\n")
```

Source 14: `compiler-tests/bad/refinable.gamma`

```

class WhileLoopTest:
  public:
    init():
      super()
      Integer a := 0
      while((a>=0) and (a<10)):
        system.out.printInteger(a)
        system.out.printString("\n")
        a := a + 1

  main(System system, String[] args):
    new WhileLoopTest()
```

Source 15: `compiler-tests/stmts/while_condn.gamma`

```

class WhileLoopTest:
  public:
    init():
      super()
      Integer a := 0
      while(true):
        system.out.printInteger(a)
        system.out.printString("\n")
        a := a + 1

  main(System system, String[] args):
    new WhileLoopTest()
```

Source 16: `compiler-tests/stmts/while.gamma`

```
class IfTest:
  private:
    void line():
      system.out.printString("\n")

    void out(String msg):
      system.out.printString(msg)
      line()

    void yes():
      out("This should print.")
    void no():
      out("This should not print.")

  public:
```

```
    init():
      super()

      out("Simple (1/2)")
      if (true)  { yes(); }
      if (false) { no();  }
      line()

      out("Basic (2/2)")
      if (true)  { yes(); } else { no();  }
      if (false) { no();  } else { yes(); }
      line()

      out("Multiple (3/3)")
      if (true)  { yes(); } elsif (false) { no();  } else { no();  }
      if (false) { no();  } elsif (true)  { yes(); } else { no();  }
      if (false) { no();  } elsif (false) { no();  } else { yes(); }
      line()

      out("Non-exhaustive (2/3)")
      if (true)  { yes(); } elsif (false) { no();  }
      if (false) { no();  } elsif (true)  { yes(); }
      if (false) { no();  } elsif (false) { no();  }

  main(System system, String[] args):
    IfTest theif := new IfTest()
```

Source 17: `compiler-tests/stmts/if.gamma`

```
class Test:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 3

    Integer overview():
      Integer success := a+b
      return success

  main(System system, String[] args):
    Test ab := new Test()
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printInteger(ab.overview())
    p.printString("\n")
```

Source 18: `compiler-tests/exprs/addInt.gamma`

```
class Test:
  public:
    Integer a
    Float   b

    init():
      super()

    Integer add():
      a := 10 * 2 * 9
      b := 6.0 * 0.5 * (-2.0)
      return 0

  main(System sys, String[] args):
```

Source 19: `compiler-tests/exprs/prod.gamma`

```
class Test:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := 5
      Float f := 1.5
      p.printString("integer - float = ")
      p.printFloat(i.toF() - f)
      p.printString("\n")

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

Source 20: `compiler-tests/exprs/subMix.gamma`

```
class Parent:
  public:
    init():
      super()

class Child extends Parent:
  public:
    init():
      super()

class Test:
```

```
  public:
    init():
      super()

  main(System system, String[] args):
    Parent child := new Child()
```

Source 21: `compiler-tests/exprs/super-assign.gamma`

```
class Test:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := 5
      Float f := 1.5
      p.printString("float/Integer = ")
      p.printFloat(f/i.toF())
      p.printString("\n")

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

Source 22: `compiler-tests/exprs/divMix.gamma`

```
class Test:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := 5
      Float f := 1.5
      p.printString("Sum of integer + float = ")
      p.printFloat(i.toF() + f)
      p.printString("\n")

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

Source 23: `compiler-tests/exprs/addMix.gamma`

```
class Test:
  private:
    void line():
      system.out.printString("\n")
```

```

    void out(String msg):
      system.out.printString(msg)
      line()

  public:
    init():
      super()
      Integer a:=2
      Integer b:=3
      Integer c

      /* less and less and equal*/
      if (a<2)  { system.out.printString("1. a=2 a<2 shouldnot print\n"); }
      elsif (a<=2)  { system.out.printString("1. a=2 a<=2 success\n"); }
      else { system.out.printString("1. should never hit here\n"); }


      /* greater and greater than equal */
      if (b>3)  { system.out.printString("2. b=3 b>3 shouldnot print\n"); }
      else  { system.out.printString("2. b=3 b>=3 success\n"); }

      /*Equal and not equal*/
      if (a <> b)  { system.out.printString("3. a!=b success \n"); }
      a:=b
      if (a=b)  { system.out.printString("4. a=b success\n"); }

      /*And or */
      if(a=3 and b=3) {  system.out.printString("5. a=3 and b=3 success\n"); }

      b:=5
      if(b=3 or a=3) {  system.out.printString("6. b=3 or a=3 success\n"); }

      /*nand and nor and not*/
      b:=4
      a:=4
      if(b=3 nor a=3) {  system.out.printString("7. b=10 nor a=10 success\n"); }
      if(not(b=4 nand a=4)) {  system.out.printString("8. not(b=4 nand a=4) success\n"); }
      b:=3
      if(b=4 nand a=4) {  system.out.printString("9. b=4 nand a=4 success\n"); }
      if(b=3 xor a=3) {  system.out.printString("10. b=3 xor a=3 success\n"); }
      c:=10
      if((a<>b or b=c) and c=10)  {  system.out.printString("11. (a<>b or b=c) and c=10 success\n"); }
      line()

```

```
  main(System system, String[] args):
    Test theif := new Test()
```

Source 24: `compiler-tests/exprs/ifeq.gamma`

```
class Test:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 3

    Integer overview():
      Integer success := a%b
      return success

  main(System system, String[] args):
    Test ab := new Test()
    Printer p := system.out
    p.printString(" 1 % 2 = ")
    p.printInteger(ab.overview())
    p.printString("\n")
```

Source 25: `compiler-tests/exprs/mod.gamma`

```

class Person:
  protected:
    String name

  public:
    init(String name):
      super()
      this.name := name

    void introduce():
      Printer p := system.out
      p.printString("Hello, my name is ")
      p.printString(name)
      p.printString(", and I am from ")
      p.printString(refine origin() to String)
      p.printString(". I am ")
      p.printInteger(refine age() to Integer)
      p.printString(" years old. My occupation is ")
      p.printString(refine work() to String)
      p.printString(". It was nice meeting you.\n")

class Test:
```

```
  protected:
    init():
      super()

  main(System sys, String[] args):
    (new Person("Matthew") {
      String introduce.origin() { return "New Jersey"; }
      Integer introduce.age() { return 33; }
      String introduce.work() { return "Student"; }
    }).introduce()

    (new Person("Arthy") {
      String introduce.origin() { return "India"; }
      Integer introduce.age() { return 57; }
      String introduce.work() { return "Student"; }
    }).introduce()

    (new Person("Weiyuan") {
      String introduce.origin() { return "China"; }
      Integer introduce.age() { return 24; }
      String introduce.work() { return "Student"; }
    }).introduce()

    (new Person("Ben") {
      String introduce.origin() { return "New York"; }
      Integer introduce.age() { return 24; }
      String introduce.work() { return "Student"; }
    }).introduce()
```

Source 26: `compiler-tests/exprs/anonymous.gamma`

```
class Test:
  public:
    init():
      super()

    void interact():
      Printer p := system.out
      Integer i := 5
      Float f := 1.5
      p.printString("integer ^ float = ")
      p.printFloat(i.toF() ^ f)
      p.printString("\n")

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

Source 27: `compiler-tests/exprs/powMix.gamma`

```
class Test:
  public:
    init():
```

```
      super()

    void interact():
      Printer p := system.out
      Integer i := 5
      Float f := 1.5
      p.printString("integer * float = ")
      p.printFloat(i.toF() * f)
      p.printString("\n")

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

Source 28: `compiler-tests/exprs/prodMix.gamma`

```
class Parent:
  protected:
    Integer a
    Integer b
    String name

  public:
    init(String name):
      super()

      this.name := name
      a := 1
      b := 2

    void print():
      Printer p := system.out
      p.printString(name)
      p.printString(":  A is ")
      p.printInteger(a)
      p.printString(", B is ")
      p.printInteger(b)
      p.printString("\n")

    void update():
      if (refinable(setA)):
        a := refine setA() to Integer
      if (refinable(setB)):
        b := refine setB() to Integer

class Son extends Parent:
  public:
    init(String name):
      super(name)

  refinement:
    Integer update.setA():
      return -1
    Integer update.setB():
      return -2
```

```

class Daughter extends Parent:
  public:
    init(String name):
      super(name)

  refinement:
    Integer update.setA():
      return 10
    Integer update.setB():
      return -5


class Test:
  protected:
    init():
      super()

  main(System sys, String[] args):
    Parent pop := new Parent("Father")
    Son son := new Son("Son")
    Daughter daughter := new Daughter("Daughter")

    pop.print()
    son.print()
    daughter.print()
    sys.out.printString("------------\n")
    pop.update()
    son.update()
    daughter.update()

    pop.print()
    son.print()
    daughter.print()
```

Source 29: `compiler-tests/exprs/simple-refine.gamma`

```
class Test:
  private:
    void print(Integer i):
      Printer p := system.out
      p.printString("a[")
      p.printInteger(i)
      p.printString("] = ")
      p.printInteger(a[i])
      p.printString("\n")

  public:
    Integer[] a
    init():
      super()
      a := new Integer[](4)
      a[0] := 3
      a[1] := 2
      a[2] := 1
```

```
      a[3] := 0

    void print():
      Integer i := 0
      while (i < 4):
        print(i)
        i += 1

  main(System system, String[] args):
    Test f
    f := new Test()
    f.print()
```

Source 30: `compiler-tests/exprs/newarr.gamma`

```
class Test:
  public:
    Float a
    Float b
    Integer c

    init():
      super()
      a := 1.5
      b := 2.2
      c := 0

    Float overview():
      Float success := a+b
      return success

  main(System system, String[] args):
    Test ab := new Test()
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printFloat(ab.overview())
    p.printString("\n")
```

Source 31: `compiler-tests/exprs/addFloat.gamma`

```
class Test:
  public:
    Integer a
    Float   b

    init():
      super()

    Integer add():
      a := (10 / 5) / -2
      b := (10.0 / 5.0) / -2.0
      return 0

```

```
  main(System sys, String[] args):
    Test t := new Test()
    Printer p := sys.out

    t.add()
    p.printString("A is ")
    p.printInteger(t.a)
    p.printString(", B is ")
    p.printFloat(t.b)
    p.printString("\n")
```

Source 32: `compiler-tests/exprs/div.gamma`

```
class Parent:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 0

    Integer overview():
      Integer success := refine toExtra(a,b) to Integer
      return success

class Child extends Parent:
  refinement:
    Integer overview.toExtra(Integer a, Integer b):
      Integer success :=  a + b
      Printer p := new Printer(true)
      p.printInteger(a)
      p.printInteger(b)
      p.printInteger(c)
      return success
  public:
    Integer a1
    Integer b1
    Integer c1

    init():
      super()
      a1 := 1
      b1 := 2
      c1 := 0

class Test:
  public:
    init():
      super()

  main(System system, String[] args):
```

```
    Parent ab := new Child()
    Printer p := system.out
    p.printString("Sum of integer = ")
    p.printInteger(ab.overview())
    p.printString("\n")
```

Source 33: `compiler-tests/exprs/refine_refinable.gamma`

```
class Parent:
  public:
    Integer a
    Integer b
    Integer c

    init():
      super()
      a := 1
      b := 2
      c := 0

    Integer overview():
      Integer success := -1
      if (refinable(toExtra)) {
          success := refine toExtra(a,b) to Integer;
      }
      return success

class Child extends Parent:
  refinement:
    Integer overview.toExtra(Integer a, Integer b):
      Integer success :=  a + b
      Printer p := new Printer(true)
      p.printInteger(a)
      p.printInteger(b)
      p.printInteger(c)
      return success
  public:
    Integer a1
    Integer b1
    Integer c1

    init():
      super()
      a1 := 1
      b1 := 2
      c1 := 0

class Test:
  public:
    init():
      super()

  main(System system, String[] args):
    Parent ab := new Child()
    Printer p := system.out
```

```
    p.printString("Sum of integer = ")
    p.printInteger(ab.overview())
    p.printString("\n")
```

Source 34: `compiler-tests/exprs/refinable.gamma`

```
class MainTest:
  public:
    init():
      super()
  main(System system, String[] args):
    Integer a
    a := 0
    a += 1
```

Source 35: `compiler-tests/structure/main.gamma`

```
class Math:
  private:
    Float xyz
  public:
    init():
      super()
    Integer add(Integer a, Integer b):
      return 6
    Integer sub(Integer a, Integer c):
      return 4
  main(System sys, String[] args):

class NonMath:
  private:
    String shakespeare
  public:
    init():
      super()
    String recite():
      return "hey"
  main(System sys, String[] hey):
```

Source 36: `compiler-tests/structure/no-bodies.gamma`

```
class FuncTest:
  public:
      Integer a

      init():
          super()
          a := 1

  private:
      Integer incre_a(Integer b):
```

```
            a := a + b
            return a

        Integer incre_a_twice(Integer b):
            incre_a(b)
            incre_a(b)
            return a

    main(System system, String[] args):
        FuncTest test := new FuncTest()
```

Source 37: `compiler-tests/structure/func.gamma`

```
open Ast
open Klass

(** Functions to be used with testing in the interpreter (or
    test scripts we write later) *)

let get_example_path dir example = String.concat Filename.
    dir_sep ["test"; "tests"; "Brace"; dir; example]

let get_example_scan dir example =
    let input = open_in (get_example_path dir example) in
    let tokens = Inspector.from_channel input in
    let _ = close_in input in
    tokens

let get_example_parse dir example =
    let tokens = get_example_scan dir example in
    Parser.cdecls (WhiteSpace.lextoks tokens) (Lexing.
    from_string "")

let get_example_longest_body dir example =
    let klasses = get_example_parse dir example in
    let methods aklass = List.flatten (List.map snd (Klass.
    klass_to_functions aklass)) in
    let all_methods = List.flatten (List.map methods klasses) in
    let with_counts = List.map (function func -> (Util.
    get_statement_count func.body, func)) all_methods in
    let maximum = List.fold_left max 0 (List.map fst with_counts
    ) in
    List.map snd (List.filter (function (c, _) -> c == maximum)
    with_counts)
```

Source 38: `Debug.ml`

```
open Printf
open Util

let output_string whatever =
    print_string whatever;
    print_newline()
```

```
let load_file filename =
    if Sys.file_exists filename
        then open_in filename
        else raise(Failure("Could not find file " ^ filename ^ "."))

let with_file f file =
    let input = load_file file in
    let result = f input in
    close_in input;
    result

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

let _ = main ()
```

Source 39: `ray.ml`

```
module StringMap = Map.Make (String);;

type class_def = { klass : string; parent : string option};;

let d1 = { klass = "myname"; parent = "Object" };;
let d3 = { klass = "myname2"; parent = "Object1" };;
let d4 = { klass = "myname3"; parent = "Object2" };;
let d2 = { klass = "myname1"; parent = "Object" };;

(*let myfunc cnameMap cdef =
    if StringMap.mem cdef.parent cnameMap then
        let cur = StringMap.find cdef.parent cnameMap in
        StringMap.add cdef.parent (cdef.klass::cur) cnameMap
    else
            StringMap.add cdef.parent [cdef.klass] cnameMap;;

*)
let rec print_list = function
[] -> ()
| e::l -> print_string e ; print_string " " ; print_list l;;

let rec spitmap fst scnd = print_string fst; print_list scnd;;

let cnameMap =

let myfunc cnameMap cdef =
    if StringMap.mem cdef.parent cnameMap then
        let cur = StringMap.find cdef.parent cnameMap in
        StringMap.add cdef.parent (cdef.klass::cur) cnameMap
    else
            StringMap.add cdef.parent [cdef.klass] cnameMap

in
   List.fold_left
    myfunc
   StringMap.empty [d1;d2;d3;d4];;
```

```
StringMap.iter spitmap cnameMap;;

print_newline
```

Source 40: `unittest/bkup.ml`

```
module StringMap = Map.Make (String);;



type var_def = string * string;;
type func_def = {
  returns : string option;
  host    : string option;
  name    : string;
  static  : bool;
  formals : var_def list;
  (*body    : stmt list;*)
};;
type member_def = VarMem of var_def | MethodMem of func_def |
    InitMem of func_def;;

(* Things that can go in a class *)
type class_sections_def = {
  privates : member_def list;
  protects : member_def list;
  publics  : member_def list;
(*  refines  : func_def list;
  mains    : func_def list;*)
};;

type class_def = { klass : string; parent : string option;
    sections : class_sections_def; };;

let sdef1 = {
 privates = [VarMem("int","a"); VarMem("int","b");];
 protects = [VarMem("int","c"); VarMem("int","d");];
 publics  = [VarMem("int","e"); VarMem("int","f");];
};;

let sdef2 = {
 privates = [ VarMem("int","g"); VarMem("int","h");];
 protects = [ VarMem("int","j"); VarMem("int","i");];
 publics = [ VarMem("int","k"); VarMem("int","l");];
};;

let sdef3 = {
 privates = [ VarMem("int","m"); VarMem("int","n");];
 protects = [ VarMem("int","p"); VarMem("int","o");];
 publics = [ VarMem("int","q"); VarMem("int","r");];
};;

let sdef4 = {
 privates = [VarMem("int","x"); VarMem("int","s");];
 protects = [VarMem("int","w"); VarMem("int","t");];
```

```
 publics = [VarMem("int","v"); VarMem("int","u");];
};;
let d1 = { klass = "myname"; parent = Some("Object"); sections = sdef1 };;
let d3 = { klass = "myname2"; parent = Some("myname1"); sections = sdef3; };;
let d4 = { klass = "myname3"; parent = Some("myname2"); sections = sdef4; };;
let d2 = { klass = "myname1"; parent = Some("myname"); sections = sdef2; };;
(*
let myfunc cnameMap cdef =
    if StringMap.mem cdef.parent cnameMap then
        let cur = StringMap.find cdef.parent cnameMap in
        StringMap.add cdef.parent (cdef.klass::cur) cnameMap
    else
            StringMap.add cdef.parent [cdef.klass] cnameMap;;

*)
let rec print_list = function
[] -> print_string "No more subclasses\n";
| e::l -> print_string e ; print_string "," ; print_list l;;

let rec spitmap fst scnd = print_string fst; print_string "->"; print_list scnd;;

let cnameMap =

let myfunc cnameMap cdef =

    let cnameMap = StringMap.add cdef.klass [] cnameMap
    in
    let myparent =
        match cdef.parent with
        None -> "Object"
        | Some str -> str
    in
    if StringMap.mem myparent cnameMap then
        let cur = StringMap.find myparent cnameMap in
        StringMap.add myparent (cdef.klass::cur) cnameMap
    else
            StringMap.add myparent [cdef.klass] cnameMap;


in
   List.fold_left myfunc StringMap.empty [d1;d2;d3;d4];;
StringMap.iter spitmap cnameMap;;

let s2bmap =

    let subtobase s2bmap cdef =
        if StringMap.mem cdef.klass s2bmap then
                (*how to raise exception*)
            s2bmap
        else
            StringMap.add cdef.klass cdef.parent s2bmap

```

```
    in
    List.fold_left
        subtobase
        StringMap.empty [d1;d2;d3;d4];;

let rec spitmap fst snd = print_string fst; print_string "->";
        match snd with
          Some str ->  print_string str;print_string "\n"
        | None -> print_string "Object's parent is none\n";
in
StringMap.iter spitmap s2bmap;;

print_newline;;


print_string "getclassdef test\n\n";;
let rec getclassdef cname clist =
    match clist with
    [] -> None
    | hd::tl -> if hd.klass = cname then Some(hd) else
    getclassdef cname tl;;

let print_cdef c = match c with None -> "No classdef" | Some c1
    -> c1.klass;;
let print_pdef p = match p with None -> "No classdef" | Some p1
    ->
            (match p1.parent with None -> "No parent" | Some x
    -> x);;

let def1 = getclassdef "myname" [d1;d2;d3;d4];;
print_string (print_cdef def1);;
print_string "\n";;
print_string(print_pdef def1);;

print_string "\n\ngetmethoddef test\n";;



let rec getmemdef mname mlist =
    match mlist with
    [] -> None
    | hd::tl -> match hd with
            VarMem(typeid, varname) -> if varname = mname then
    Some(typeid) else getmemdef mname tl
            | _ -> None
;;

(*Given a class definition and variable name, the lookupfield
looksup for the field in the privates, publics and protects list
    .
If found returns a (classname, accessspecifier, typeid,
    variablename) tuple
If not found returns a None*)
let lookupfield cdef vname =
    let pmem = getmemdef vname cdef.sections.privates
    in
    match pmem with
```

```
        Some def -> Some(cdef.klass, "private", vname, def)
         |  None     ->
            let pubmem = getmemdef vname cdef.sections.publics
            in
            match pubmem with
                Some def -> Some(cdef.klass, "public", vname, def)
                 |  None     ->
                    let promem = getmemdef vname cdef.sections.protects
                    in
                    match promem with
                        Some def -> Some(cdef.klass, "protect", vname, def)
                       |    None  -> None
;;

(*getfield takes classname and variablename;
  looks for the class with the classname;
  If classname found, looksup the variable in the class;
  Else returns None
*)
let fstoffour (x,_,_,_) = x;;
let sndoffour (_,x,_,_) = x;;
let throffour (_,_,x,_) = x;;
let lstoffour (_,_,_,x) = x;;

let rec getfield cname vname cdeflist =
    let classdef = getclassdef cname cdeflist
    in
    match classdef with
              None ->
        if cname = "Object" then
            None
        else
            let basename = match(StringMap.find cname s2bmap) with Some b -> b | None -> "Object"
            in
            getfield basename vname cdeflist
    |    Some (cdef) -> lookupfield cdef vname;;

let field = getfield "myname3" "a" [d1;d2;d3;d4]
in
match field with
None -> print_string "field not found\n";
| Some tup -> print_string (fstoffour(tup));;
```

Source 41: `unittest/sast.ml`

```
%{
open Ast

(** Parser that reads from the scanner and produces an AST. *)

(** Set a single function to belong to a certain section *)
let set_func_section_to sect f = { f with section = sect }
```

```
(** Set a list of functions to belong to a certain section *)
let set_func_section sect = List.map (set_func_section_to sect)

(** Set a single member to belong to a certain subset of class memory.
    This is necessary as a complicated function because init and main
    can live in one of the several access levels. *)
let set_mem_section_to sect = function
  | VarMem(v) -> VarMem(v)
  | InitMem(func) -> InitMem({ func with section = sect })
  | MethodMem(func) -> MethodMem({ func with section = sect })

(** Set a list of members to belong to a certain subset of class memory *)
let set_mem_section sect = List.map (set_mem_section_to sect)


(** Set the klass of a func_def *)
let set_func_klass aklass func = { func with inklass = aklass }

(** Set the klass of a function member *)
let set_member_klass aklass = function
  | InitMem(func) -> InitMem(set_func_klass aklass func)
  | MethodMem(func) -> MethodMem(set_func_klass aklass func)
  | v -> v

(** Set the klass of all sections *)
let set_func_class aklass sections =
  let set_mems = List.map (set_member_klass aklass) in
  let set_funcs = List.map (set_func_klass aklass) in
  { privates = set_mems  sections.privates;
    publics  = set_mems  sections.publics;
    protects = set_mems  sections.protects;
    refines  = set_funcs sections.refines;
    mains    = set_funcs sections.mains }
%}

%token <int> SPACE
%token COLON NEWLINE
%token LPAREN RPAREN LBRACKET RBRACKET COMMA LBRACE RBRACE
%token PLUS MINUS TIMES DIVIDE MOD POWER
%token PLUSA MINUSA TIMESA DIVIDEA MODA POWERA
%token EQ NEQ GT LT GEQ LEQ AND OR NAND NOR XOR NOT
%token IF ELSE ELSIF WHILE
%token ASSIGN RETURN CLASS EXTEND SUPER INIT PRIVATE PROTECTED PUBLIC
%token NULL VOID THIS
%token NEW MAIN ARRAY
%token REFINABLE REFINE REFINES TO
%token SEMI COMMA DOT EOF

%token <string> TYPE
%token <int> ILIT
%token <float> FLIT
%token <bool> BLIT
%token <string> SLIT
```

```
%token <string> ID

/* Want to work on associtivity when I'm a bit fresher */
%right ASSIGN PLUSA MINUSA TIMESA DIVIDEA MODA POWERA
%left OR NOR XOR
%left AND NAND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%nonassoc UMINUS
%left NOT POWER
%left LPAREN RPAREN LBRACKET RBRACKET
%left DOT

%start cdecls
%type <Ast.program> cdecls

%%

/* Classe and subclassing */
cdecls:
  | cdecl { [$1] }
  | cdecls cdecl { $2 :: $1 }
cdecl:
  | CLASS TYPE extend_opt class_section_list
    { { klass     = $2;
        parent    = $3;
        sections  = set_func_class $2 $4  } }
extend_opt:
  | /* default */  { Some("Object") }
  | EXTEND TYPE    { Some($2) }

/* Class sections */
class_section_list:
  | LBRACE class_sections RBRACE  { $2 }
class_sections:
  | /* Base Case */
    { { privates = [];
        protects = [];
        publics  = [];
        refines  = [];
        mains    = [] } }
  | class_sections private_list  { { $1 with privates = (
    set_mem_section Privates $2)  @  $1.privates } }
  | class_sections protect_list  { { $1 with protects = (
    set_mem_section Protects $2)  @  $1.protects } }
  | class_sections public_list   { { $1 with publics  = (
    set_mem_section Publics $2)   @  $1.publics  } }
  | class_sections refine_list   { { $1 with refines  = (
    set_func_section Refines $2)  @  $1.refines  } }
  | class_sections main_method   { { $1 with mains    = (
    set_func_section_to Mains $2) :: $1.mains    } }

/* Refinements */
refine_list:
  | REFINES LBRACE refinements RBRACE { $3 }
```

```
refinements:
  | /* Can be empty */      { [] }
  | refinements refinement  { $2 :: $1 }
refinement:
  | vartype ID DOT invocable  { { $4 with returns = Some($1);
    host = Some($2) } }
  | VOID ID DOT invocable     { { $4 with host = Some($2) } }

/* Private, protected, public members */
private_list:
  | PRIVATE member_list    { $2 }
protect_list:
  | PROTECTED member_list  { $2 }
public_list:
  | PUBLIC member_list     { $2 }

/* Members of such access groups */
member_list:
  | LBRACE members RBRACE  { $2 }
members:
  | { [] }
  | members member  { $2 :: $1 }
member:
  | vdecl semi  { VarMem($1)    }
  | mdecl       { MethodMem($1) }
  | init        { InitMem($1)   }

/* Methods */
mdecl:
  | vartype invocable  { { $2 with returns = Some($1) } }
  | VOID invocable     { $2 }

/* Constructors */
init:
  | INIT callable  { { $2 with name = "init" } }

/* Each class has an optional main */
main_method:
  | MAIN callable  { { $2 with name = "main"; static = true } }

/* Anything that is callable has these forms */
invocable:
  | ID callable  { { $2 with name = $1 } }
callable:
  | formals stmt_block
    { { returns = None;
        host    = None;
        name    = "";
        static  = false;
        formals = $1;
        body    = $2;
        section = Privates;
        inklass = "";
        uid     = UID.uid_counter ();
        builtin = false } }

/* Statements */
```

```
stmt_block:
  | LBRACE stmt_list RBRACE  { List.rev $2 }
stmt_list:
  | /* nada */      { [] }
  | stmt_list stmt  { $2 :: $1 }
stmt:
  | vdecl semi               { Decl($1, None) }
  | vdecl ASSIGN expr semi   { Decl($1, Some($3)) }
  | SUPER actuals semi       { Super($2) }
  | RETURN expr semi         { Return(Some($2)) }
  | RETURN semi;             { Return(None) }
  | conditional              { $1 }
  | loop                     { $1 }
  | expr semi                { Expr($1) }

/* Control Flow */
conditional:
  | IF pred stmt_block else_list  { If((Some($2), $3) :: $4) }
else_list:
  | /* nada */                          { [] }
  | ELSE stmt_block                     { [(None, $2)] }
  | ELSIF pred stmt_block else_list     { (Some($2), $3) :: $4 }
loop:
  | WHILE pred stmt_block  { While($2, $3) }
pred:
  | LPAREN expr RPAREN  { $2 }


/* Expressions */
expr:
  | assignment          { $1 }
  | invocation          { $1 }
  | field               { $1 }
  | value               { $1 }
  | arithmetic          { $1 }
  | test                { $1 }
  | instantiate         { $1 }
  | refineexpr          { $1 }
  | literal             { $1 }
  | LPAREN expr RPAREN  { $2 }
  | THIS                { This }
  | NULL                { Null }

assignment:
  | expr ASSIGN expr   { Assign($1, $3) }
  | expr PLUSA expr    { Assign($1, Binop($1, Arithmetic(Add),
    $3)) }
  | expr MINUSA expr   { Assign($1, Binop($1, Arithmetic(Sub),
    $3)) }
  | expr TIMESA expr   { Assign($1, Binop($1, Arithmetic(Prod),
    $3)) }
  | expr DIVIDEA expr  { Assign($1, Binop($1, Arithmetic(Div),
    $3)) }
  | expr MODA expr     { Assign($1, Binop($1, Arithmetic(Mod),
    $3)) }
  | expr POWERA expr   { Assign($1, Binop($1, Arithmetic(Pow),
    $3)) }
```

```

invocation:
  | expr DOT ID actuals { Invoc($1, $3, $4) }
  | ID actuals { Invoc(This, $1, $2) }

field:
  | expr DOT ID  { Field($1, $3) }

value:
  | ID    { Id($1) }
  | expr LBRACKET expr RBRACKET  { Deref($1, $3) }

arithmetic:
  | expr PLUS expr           { Binop($1, Arithmetic(Add), $3) }
  | expr MINUS expr          { Binop($1, Arithmetic(Sub), $3) }
  | expr TIMES expr          { Binop($1, Arithmetic(Prod), $3) }
  | expr DIVIDE expr         { Binop($1, Arithmetic(Div), $3) }
  | expr MOD expr            { Binop($1, Arithmetic(Mod), $3) }
  | expr POWER expr          { Binop($1, Arithmetic(Pow), $3) }
  | MINUS expr %prec UMINUS  { Unop(Arithmetic(Neg), $2) }

test:
  | expr AND expr   { Binop($1, CombTest(And), $3) }
  | expr OR expr    { Binop($1, CombTest(Or), $3) }
  | expr XOR expr   { Binop($1, CombTest(Xor), $3) }
  | expr NAND expr  { Binop($1, CombTest(Nand), $3) }
  | expr NOR expr   { Binop($1, CombTest(Nor), $3) }
  | expr LT expr    { Binop($1, NumTest(Less), $3) }
  | expr LEQ expr   { Binop($1, NumTest(Leq), $3) }
  | expr EQ expr    { Binop($1, NumTest(Eq), $3) }
  | expr NEQ expr   { Binop($1, NumTest(Neq), $3) }
  | expr GEQ expr   { Binop($1, NumTest(Geq), $3) }
  | expr GT expr    { Binop($1, NumTest(Grtr), $3) }
  | NOT expr        { Unop(CombTest(Not), $2) }
  | REFINABLE LPAREN ID RPAREN { Refinable($3) }

instantiate:
  | NEW vartype actuals  { NewObj($2, $3) }
  | NEW vartype actuals LBRACE refinements RBRACE  { Anonymous(
    $2, $3, List.map (set_func_klass $2) $5) }

refineexpr:
  | REFINE ID actuals TO vartype  { Refine($2, $3, Some($5)) }
  | REFINE ID actuals TO VOID     { Refine($2, $3, None) }

literal:
  | lit  { Literal($1) }

/* Literally necessary */
lit:
  | SLIT { String($1) }
  | ILIT { Int($1) }
  | FLIT { Float($1) }
  | BLIT { Bool($1) }

/* Parameter lists */
formals:
```

```
  | LPAREN formals_opt RPAREN  { $2 }
formals_opt:
  | { [] }
  | formals_list  { List.rev $1 }
formals_list:
  | vdecl  { [$1] }
  | formals_list COMMA vdecl  { $3 :: $1 }

/* Arguments */
actuals:
  | LPAREN actuals_opt RPAREN { $2 }
actuals_opt:
  | { [] }
  | actuals_list { List.rev $1 }
actuals_list:
  | expr { [$1] }
  | actuals_list COMMA expr { $3 :: $1}

/* Variable declaration */
vdecl:
  | vartype ID { ($1, $2) }
vartype:
  | TYPE          { $1 }
  | vartype ARRAY { $1 ^ "[]" }

/* Eat multiple semis */
semi:
  | SEMI  {}
  | semi SEMI  {}
```

Source 42: `parser.mly`

```
open Ast
open Util
open StringModules
open GlobalData

(** Approximates a class *)
(**
    From a class get the parent
    @param aklass is a class_def to get the parent of
    @return The name of the parent object
  *)
let klass_to_parent aklass = match aklass with
    | { klass = "Object" } -> raise(Invalid_argument("Cannot get parent of the root"))
    | { parent = None; _ } -> "Object"
    | { parent = Some(aklass); _ } -> aklass

(**
    Utility function -- place variables in left, methods (including init) in right
    @param mem A member_def value (VarMem, MethodMem, InitMem)
    @return Places the values held by VarMem in Left, values held by MethodMem or InitMem in Right
```

```
  *)
let member_split mem = match mem with
    | VarMem(v) -> Left(v)
    | MethodMem(m) -> Right(m)
    | InitMem(i) -> Right(i)

(**
    Stringify a section to be printed
    @param section A class_section value (Privates, Protects,
    Publics, Refines, or Mains)
    @return The stringification of the section for printing
  *)
let section_string section = match section with
    | Privates -> "private"
    | Protects -> "protected"
    | Publics -> "public"
    | Refines -> "refinement"
    | Mains -> "main"

(**
    Return the variables of the class
    @param aklass The class to explore
    @return A list of ordered pairs representing different
    sections,
    the first item of each pair is the type of the section, the
    second
    is a list of the variables defs (type, name). Note that this
     only
    returns pairs for Publics, Protects, and Privates as the
    others
    cannot have variables
  *)
let klass_to_variables aklass =
    let vars members = fst (either_split (List.map member_split
    members)) in
    let s = aklass.sections in
    [(Publics, vars s.publics); (Protects, vars s.protects); (
    Privates, vars s.privates)]

(**
    Return the methods of the class
    @param aklass The class to explore
    @return A list of ordered pairs representing different
    sections,
    the first item of each pair is the type of the section, the
    second
    is a list of the methods. Note that this only returns the
    methods
    in Publics, Protects, or Privates as the other sections don'
    t have
    `normal' methods in them
  *)
let klass_to_methods aklass =
    let funcs members = snd (either_split (List.map member_split
     members)) in
    let s = aklass.sections in
    [(Publics, funcs s.publics); (Protects, funcs s.protects); (
```

```
      Privates, funcs s.privates)]

(**
    Get anything that is invocable, not just instance methods
    @param aklass The class to explore
    @return The combined list of refinements, mains, and methods
  *)
let klass_to_functions aklass =
    let s = aklass.sections in
    (Refines, s.refines) :: (Mains, s.mains) :: klass_to_methods
     aklass

(**
    Return whether two function definitions have conflicting
    signatures
    @param func1 A func_def
    @param func2 A func_def
    @return Whether the functions have the same name and the
    same parameter type sequence
  *)
let conflicting_signatures func1 func2 =
    let same_type (t1, _) (t2, _) = (t1 = t2) in
    let same_name = (func1.name = func2.name) in
    let same_params = try List.for_all2 same_type func1.formals
    func2.formals with
        | Invalid_argument(_) -> false in
    same_name && same_params

(**
    Return a string that describes a function
    @param func A func_def
    @return A string showing the simple signature ([host.]name
    and arg types)
  *)
let signature_string func =
    let name = match func.host with
        | None -> func.name
        | Some(h) -> Format.sprintf "%s.%s" h func.name in
    Format.sprintf "%s(%s)" name (String.concat ", " (List.map
    fst func.formals))

(**
    Return a string representing the full signature of the
    function
    @param func A func_def
    @return A string showing the signature (section, [host.]name
    , arg types)
  *)
let full_signature_string func =
    let ret = match func.returns with
        | None -> "Void"
        | Some(t) -> t in
    Format.sprintf "%s %s %s" (section_string func.section) ret
    (signature_string func)

(**
    Given a class_data record, a class name, and a variable name
```

```
    , lookup the section and type
    info for that variable.
    @param data A class_data record
    @param klass_name The name of a class (string)
    @param var_name The name of a variable (string)
    @return Either None if the variable is not declared in the class or Some((section, type))
    where the variable is declared in section and has the given type.
  *)
let class_var_lookup data klass_name var_name =
    match map_lookup klass_name data.variables with
        | Some(var_map) -> map_lookup var_name var_map
        | _ -> None

(**
    Given a class_data record, a class_name, and a variable name, lookup the class in the hierarchy
    that provides access to that variable from within that class (i.e. private in that class or
    public / protected in an ancestor).
    @param data A class_data record.
    @param klass_name The name of a class (string)
    @param var_name The name of a variable (string).
    @return (class (string), type (string), class_section) option (None if not found).
  *)
let class_field_lookup data klass_name var_name =
    let var_lookup klass = class_var_lookup data klass var_name in
    let rec lookup klass sections = match var_lookup klass, klass with
        | Some((sect, vtype)), _ when List.mem sect sections -> Some((klass, vtype, sect))
        | _, "Object" -> None
        | _, _ -> lookup (StringMap.find klass data.parents) [Publics; Protects] in
    lookup klass_name [Publics; Protects; Privates]

(**
    Given a class_data record, a class name, a var_name, and whether the receiver of the field lookup
    is this, return the lookup of the field in the ancestry of the object. Note that this restricts
    things that should be kept protected (thus this thusly passed)
    @param data A class_data record
    @param klass_name The name of a class (string)
    @param var_name The name of a variable (string)
    @return Either the left of a triple (class found, type, section) or a Right of a boolean, which
    is true if the item was found but inaccessible and false otherwise.
  *)
let class_field_far_lookup data klass_name var_name this =
    match class_field_lookup data klass_name var_name with
        | Some((klass, vtyp, section)) when this || section =
```

```
    Publics -> Left((klass, vtyp, section))
        | Some(_) -> Right(true)
        | None -> Right(false)

(**
    Given a class_data record, a class name, and a method name, lookup all the methods in the
    given class with that name.
    @param data A class_data record
    @param klass_name The name of a class (string)
    @param func_name The name of a method (string)
    @return A list of methods in the class with that name or the empty list if no such method exists.
  *)
let class_method_lookup data klass_name func_name =
    match map_lookup klass_name data.methods with
        | Some(method_map) -> map_lookup_list func_name method_map
        | _ -> []

(**
    Given a class_data record, a class name, a method name, and whether the current context is
    `this' (i.e. if we want private / protected / etc), then return all methods in the ancestry
    of that class with that name (in the appropriate sections).
    @param data A class_data record value
    @param klass_name The name of a class.
    @param method_name The name of a method to look up
    @param this search mode -- true means public/protected/private and then public/protected,
    false is always public
    @return A list of methods with the given name.
  *)
let class_ancestor_method_lookup data klass_name method_name this =
    let (startsects, recsects) = if this then ([Publics; Protects; Privates], [Publics; Protects]) else ([Publics], [Publics]) in
    let rec find_methods found aklass sects =
        let accessible f = List.mem f.section sects in
        let funcs = List.filter accessible (class_method_lookup data aklass method_name) in
        let found = funcs @ found in
        if aklass = "Object" then found
        else if method_name = "init" then found
        else find_methods found (StringMap.find aklass data.parents) recsects in
    find_methods [] klass_name startsects

(**
    Given a class_data record, class name, method name, and refinement name, return the list of
    refinements in that class for that method with that name.
    @param data A class_data record value
    @param klass_name A class name
    @param method_name A method name
```

```
    @param refinement_name A refinement name
    @return A list of func_def values that match the given requirements. Note that this returns the
    functions defined IN class name, not the ones that could be used INSIDE class name (via a refine
    invocation). i.e. functions that may be invoked by the parent.
  *)
let refine_lookup data klass_name method_name refinement_name =
    match map_lookup klass_name data.refines with
        | Some(map) -> map_lookup_list (method_name ^ "." ^ refinement_name) map
        | _ -> []

(**
    Given a class_data record, a class name, a method name, and a refinement name, return the list
    of refinements across all subclasses for the method with that name.
    @param data A class_data record value
    @param klass_name A class name
    @param method_name A method name
    @param refinement_name A refinement name
    @return A list of func_def values that meet the criteria and may be invoked by this given method.
    i.e. these are all functions residing in SUBCLASSES of the named class.
  *)
let refinable_lookup data klass_name method_name refinement_name =
    let refines = match map_lookup klass_name data.refinable with
        | Some(map) -> map_lookup_list method_name map
        | None -> [] in
    List.filter (fun f -> f.name = refinement_name) refines

(**
    Given a class_data record and two classes, returns the distance between them. If one is a proper
    subtype of the other then Some(n) is returned where n is non-zero when the two classes are different
    and comparable (one is a subtype of the other), zero when they are the same, and None when they are
    incomparable (one is not a subtype of the other)
    @param data A class_data record
    @param klass1 A class to check the relation of to klass2
    @param klass2 A class to check the relation of to klass1
    @return An int option, None when the two classes are incomparable, Some(positive) when klass2 is an
    ancestor of klass1, Some(negative) when klass1 is an ancestor of klass2.
  *)
let get_distance data klass1 klass2 =
    (* We let these pop exceptions because that means bad programming on the compiler
     * writers part, not on the GAMMA programmer's part (when klass1, klass2 aren't found)
```

```
     *)
    let klass1_map = StringMap.find klass1 data.distance in
    let klass2_map = StringMap.find klass2 data.distance in
    match map_lookup klass2 klass1_map, map_lookup klass1 klass2_map with
        | None, None -> None
        | None, Some(n) -> Some(-n)
        | res, _ ->  res

(**
    Check if a type exists in the class data -- convenience function
    @param data A class_data record
    @param atype The name of a class (string)
    @return True if the atype is a known type, false otherwise.
  *)
let is_type data atype =
    let lookup = try String.sub atype 0 (String.index atype '[') with
        | Not_found -> atype in
    StringSet.mem lookup data.known

(**
    Check if a class is a subclass of another given a class_data record
    @param data A class_data record
    @param subtype A class name (string)
    @param supertype A class name (string)
    @return Whether subtype has supertype as an ancestor given data.
    Note that this is true when the two are equal (trivial ancestor).
  *)
let is_subtype data subtype supertype =
    let basetype s = try let n = String.index s '[' in String.sub s 0 n with Not_found -> s in
    match get_distance data (basetype subtype) (basetype supertype) with
        | Some(n) when n >= 0 -> true
        | _ -> false

(**
    Check if a class is a proper subclass of another given a class_data record
    @param data A class_data record
    @param subtype A class name (string)
    @param supertype A class name (string)
    @return Whether subtype has supertype as an ancestor given data.
    Note that this IS NOT true when the two are equal (trivial ancestor).
  *)
let is_proper_subtype data subtype supertype =
    match get_distance data subtype supertype with
        | Some(n) when n > 0 -> true
        | _ -> false
```

```
(**
    Return whether a list of actuals and a list of formals are compatible.
    For this to be true, each actual must be a (not-necessarily-proper) subtype
    of the formal at the same position. This requires that both be the same
    in quantity, obviously.
    @param data A class_data record (has type information)
    @param actuals A list of the types (and just the types) of the actual arguments
    @param formals A list of the types (and just the types) of the formal arguments
    @return Whether the actual arguments are compatible with the formal arguments.
  *)
let compatible_formals data actuals formals =
    let compatible formal actual = is_subtype data actual formal in
    try List.for_all2 compatible formals actuals with
        | Invalid_argument(_) -> false

(**
    Return whether a given func_def is compatible with a list of actual arguments.
    This means making sure that it has the right number of formal arguments and that
    each actual agument is a subtype of the corresponding formal argument.
    @param data A class_data record (has type information)
    @param actuals A list of the types (and just the types) of the actual arguments
    @param func A func_def from which to get formals
    @return Whether the given func_def is compatible with the actual arguments.
  *)
let compatible_function data actuals func =
    compatible_formals data actuals (List.map fst func.formals)

(**
    Return whether a function's return type is compatible with a desired return type.
    Note that if the desired return type is None then the function is compatible.
    Otherwise if it is not None and the function's is, then it is not compatible.
    Lastly, if the desired type is a supertype of the function's return type then the
    function is compatible.
    @param data A class_data record value
    @param ret_type The desired return type
    @param func A func_def to test.
    @return True if compatible, false if not.
  *)
let compatible_return data ret_type func =
    match ret_type, func.returns with
        | None, _ -> true
```

```
        | _, None -> false
        | Some(desired), Some(given) -> is_subtype data given desired

(**
    Return whether a function's signature is completely compatible with a return type
    and a set of actuals
    @param data A class_data record value
    @param ret_type The return type (string option)
    @param actuals The list of actual types
    @param func A func_def value
    @return True if compatible, false if not.
  *)
let compatible_signature data ret_type actuals func =
    compatible_return data ret_type func && compatible_function data actuals func

(**
    Filter a list of functions based on their section.
    @param funcs a list of functions
    @param sects a list of class_section values
    @return a list of functions in the given sections
    *)
let in_section sects funcs =
    List.filter (fun f -> List.mem f.section sects) funcs

(**
    Given a class_data record, a list of actual arguments, and a list of methods,
    find the best matches for the actuals. Note that if there are multiple best
    matches (i.e. ties) then a non-empty non-singleton list is returned.
    Raises an error if somehow our list of compatible methods becomes incompatible
    [i.e. there is a logic error in the compiler].
    @param data A class_data record
    @param actuals The list of types (and only types) for the actual arguments
    @param funcs The list of candidate functions
    @return The list of all best matching functions (should be at most one, we hope).
  *)
let best_matching_signature data actuals funcs =
    let funcs = List.filter (compatible_function data actuals) funcs in
    let distance_of actual formal = match get_distance data actual formal with
        | Some(n) when n >= 0 -> n
        | _ -> raise(Invalid_argument("Compatible methods somehow incompatible: " ^ actual ^ " vs. " ^ formal ^ ". Compiler error.")) in
    let to_distance func = List.map2 distance_of actuals (List.map fst func.formals) in
    let with_distances = List.map (fun func -> (func, to_distance func)) funcs in
```

```
    let lex_compare (_, lex1) (_, lex2) = lexical_compare lex1
    lex2 in
    List.map fst (find_all_min lex_compare with_distances)

(**
    Given a class_data record, method name, and list of actuals,
     and a list of sections to consider,
    get the best matching  method. Note that if there is more
    than one then an exception is raised
    as this should have been reported during collision detection
     [compiler error].
    @param data A class_data record
    @param method_name The name to lookup candidates for
    @param actuals The list of types (and only types) for the
    actual arguments
    @param sections The sections to filter on (only look in
    these sections)
    @return Either None if no function is found, Some(f) if one
    function is found, or an error is raised.
  *)
let best_method data klass_name method_name actuals sections =
    let methods = class_method_lookup data klass_name
    method_name in
    let methods = in_section sections methods in
    match best_matching_signature data actuals methods with
        | [] -> None
        | [func] -> Some(func)
        | _ -> raise(Invalid_argument("Multiple methods named "
    ^ method_name ^ " of the same signature in " ^ klass_name ^
    "; Compiler error."))

let best_inherited_method data klass_name method_name actuals
    this =
    let methods = class_ancestor_method_lookup data klass_name
    method_name this in
    match best_matching_signature data actuals methods with
        | [] -> None
        | [func] -> Some(func)
        | _ -> raise(Invalid_argument("Multiple methods named "
    ^ method_name ^ " of the same signature inherited in " ^
    klass_name ^ "; Compiler error."))

(**
    Given the name of a refinement to apply, the list of actual
    types,
    find the compatible refinements via the data / klass_name /
    method_name.
    Partition the refinements by their inklass value and then
    return a list
    of the best matches from each partition.
    @param data A class_data record value
    @param klass_name A class name
    @param method_name A method name
    @param refine_name A refinement name
    @param actuals The types of the actual arguments
    @return A list of functions to switch on based on the
    actuals.
```

```
  *)
let refine_on data klass_name method_name refine_name actuals ret_type =
    (* These are all the refinements available from subclasses *)
    let refines = refinable_lookup data klass_name method_name refine_name in

    (* Compatible functions *)
    let compat = List.filter (compatible_signature data ret_type actuals) refines in

    (* Organize by inklass *)
    let to_class map f = add_map_list f.inklass f map in
    let by_class = List.fold_left to_class StringMap.empty compat in

    (* Now make a map of only the best *)
    let best funcs = match best_matching_signature data actuals funcs with
        | [func] -> func
        | _ -> raise(Failure("Compiler error finding a unique best refinement.")) in
    let to_best klass funcs map = StringMap.add klass (best funcs) map in
    let best_map = StringMap.fold to_best by_class StringMap.empty in

    (* Now just return the bindings from the best *)
    List.map snd (StringMap.bindings best_map)

(**
    Get the names of the classes in level order (i.e. from root down).
    @param data A class_data record
    @return The list of known classes, from the root down.
  *)
let get_class_names data =
    let kids aklass = map_lookup_list aklass data.children in
    let rec append found = function
        | [] -> List.rev found
        | items -> let next = List.flatten (List.map kids items) in
            append (items@found) next in
    append [] ["Object"]


(**
    Get leaf classes
    @param data A class_data record
    @return A list of leaf classes
  *)
let get_leaves data =
    let is_leaf f = match map_lookup_list f data.children with
        | [] -> true
        | _ -> false in
    let leaves = StringSet.filter is_leaf data.known in
```

```
    StringSet.elements leaves
```

Source 43: `Klass.ml`

```
all: compile _tools _ray _doc

compile:
    #Generate the lexer and parser
    ocamllex scanner.mll
    ocamlyacc parser.mly

    ocamlc -c -g Ast.mli
    ocamlc -c -g UID.ml

    ocamlc -c -g parser.mli
    ocamlc -c -g scanner.ml
    ocamlc -c -g parser.ml

    ocamlc -c -g WhiteSpace.ml
    ocamlc -c -g Inspector.mli
    ocamlc -c -g Inspector.ml
    ocamlc -c -g Pretty.ml

    ocamlc -c -g Util.ml
    ocamlc -c -g StringModules.ml
    ocamlc -c -g GlobalData.mli
    ocamlc -c -g Klass.mli
    ocamlc -c -g KlassData.mli
    ocamlc -c -g BuiltIns.mli
    ocamlc -c -g BuiltIns.ml
    ocamlc -c -g Klass.ml
    ocamlc -c -g KlassData.ml
    ocamlc -c -g Variables.ml
    ocamlc -c -g Sast.mli
    ocamlc -c -g BuildSast.mli
    ocamlc -c -g BuildSast.ml
    ocamlc -c -g Unanonymous.mli
    ocamlc -c -g Unanonymous.ml
    ocamlc -c -g Cast.mli
    ocamlc -c -g GenCast.ml
    ocamlc -c -g GenC.ml
    ocamlc -c -g Debug.ml

    ocamlc -c -g classinfo.ml
    ocamlc -c -g inspect.ml
    ocamlc -c -g prettify.ml
    ocamlc -c -g streams.ml
    ocamlc -c -g canonical.ml
    ocamlc -c -g freevars.ml
    ocamlc -c -g ray.ml

_tools:
    #Make the tools
    ocamlc -g -o tools/prettify UID.cmo scanner.cmo parser.cmo
    Inspector.cmo Pretty.cmo WhiteSpace.cmo prettify.cmo
```

```
	ocamlc -g -o tools/inspect UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.cmo inspect.cmo
	ocamlc -g -o tools/streams UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.cmo streams.cmo
	ocamlc -g -o tools/canonical UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.cmo canonical.cmo
	ocamlc -g -o tools/freevars UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.cmo Util.cmo StringModules.cmo str.cma BuiltIns.cmo Klass.cmo KlassData.cmo Debug.cmo Variables.cmo freevars.cmo
	ocamlc -g -o tools/classinfo UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.cmo Util.cmo StringModules.cmo str.cma BuiltIns.cmo Klass.cmo KlassData.cmo classinfo.cmo

_ray:
	#Make ray
	mkdir -p bin
	ocamlc -g -o bin/ray UID.cmo scanner.cmo parser.cmo Inspector.cmo WhiteSpace.cmo Util.cmo StringModules.cmo str.cma BuiltIns.cmo Klass.cmo KlassData.cmo Debug.cmo Variables.cmo BuildSast.cmo Unanonymous.cmo GenCast.cmo GenC.cmo ray.cmo

nodoc: compile _tools _ray

docsources = Ast.mli BuildSast.ml BuildSast.mli BuiltIns.ml BuiltIns.mli Cast.mli Debug.ml GenCast.ml GenC.ml GlobalData.mli Inspector.ml Inspector.mli Klass.ml Klass.mli KlassData.ml KlassData.mli Pretty.ml Sast.mli StringModules.ml UID.ml Unanonymous.ml Unanonymous.mli Util.ml Variables.ml WhiteSpace.ml parser.ml parser.mli scanner.ml
docgen = ./doc/.docgen

_doc:
	#Generate the documentation
	mkdir -p doc
	ocamldoc -hide-warnings -dump $(docgen) -keep-code $(docsources)
	ocamldoc -hide-warnings -load $(docgen) -d doc -t "The Ray Compiler" -html -colorize-code -all-params
	ocamldoc -hide-warnings -load $(docgen) -dot -o "./doc/ray-modules.dot"
	ocamldoc -hide-warnings -load $(docgen) -dot -dot-types -o "./doc/ray-types.dot"

bleach:
	rm *.cmi *.cmo parser.ml parser.mli scanner.ml
	rm -r ./doc

clean:
	rm *.cmi *.cmo parser.ml parser.mli scanner.ml

cleantools:
	rm tools/{prettify,inspect,streams,canonical,freevars,classinfo}
```

Source 44: `Makefile`

```
val ast_to_sast_klass : GlobalData.class_data -> Ast.class_def
    -> Sast.class_def
val ast_to_sast : GlobalData.class_data -> Sast.class_def list
val update_refinements : GlobalData.class_data -> Sast.class_def
     list -> Sast.class_def list
```

Source 45: `BuildSast.mli`

```
/* N queens iterative solution */

class ChessBoard:
  public:
    init(Integer size):
      super()
      n := size
      solution_count := 0
      arrangement := new Integer[](n)
      Integer i := 0
      while(i < n):
        arrangement[i] := -1
        i += 1

    Boolean test_column(Integer row):
      Integer i := 0
      while(i < row):
        if(arrangement[i] = arrangement[row]):
          return false
        i += 1
      return true

    Boolean test_diag(Integer row):
      Integer i := 0
      while(i < row):
        if(((arrangement[row] - arrangement[i]) = row - i) or ((
    arrangement[row] - arrangement[i]) = i - row)):
          return false
        i += 1
      return true

    Boolean test(Integer row):
      if(test_column(row) and test_diag(row)):
        return true
      else:
        return false

    Integer print_board():
      system.out.printString("\nSolution # ")
      system.out.printInteger(solution_count)
```

```
      system.out.printString("\n")
      Integer r := 0
      while(r < n):
        Integer c := 0
        while(c < n):
          if(arrangement[r] = c):
            system.out.printString("Q ")
          else:
            system.out.printString("* ")
          c += 1
        system.out.printString("\n")
        r += 1
      return 0

    Integer get_solutions():
      arrangement[0] := -1
      Integer row := 0
      while(row >= 0):
        arrangement[row] += 1
        while(arrangement[row] < n and not test(row)):
          arrangement[row] += 1
        if(arrangement[row] < n):
          if(row = n - 1):
            solution_count += 1
            print_board()
          else:
            row += 1
            arrangement[row] := -1
        else:
          row -= 1
      return 0

  private:
    Integer n
    Integer solution_count
    Integer[] arrangement

  main(System system, String[] args):
    system.out.printString("Chess board size: ")
    Integer size := system.in.scanInteger()
    ChessBoard nqueens := new ChessBoard(size)
    nqueens.get_solutions()
```

Source 46: `demo/nqueens.gamma`

```
class HelloWorld:
  public:
    String greeting
    init():
      super()
      greeting := "Hello World!"

  main(System system, String[] args):
    HelloWorld hw := new HelloWorld()
    system.out.printString(hw.greeting)
```

```
    system.out.printString("\n")
```

Source 47: `demo/helloworld.gamma`

```
class Bank:
  public:
    init():
      super()
      id_counter := 0
      accounts := new Account[](100)

      /* Anonymous instantiation can 'get around' protected
    constructors */
      Account president := (new Account(id_counter, "Bank
    President") {
        Float apply_interest.rate() { return 0.10; }
      })
      accounts[id_counter] := president
      id_counter += 1

    Integer open_checking(String client_name):
      Account new_account := new Checking(id_counter,
    client_name)
      accounts[id_counter] := new_account
      id_counter += 1
      return id_counter-1

    Integer open_savings(String client_name):
      Account new_account := new Savings(id_counter, client_name
    )
      accounts[id_counter] := new_account
      id_counter += 1
      return id_counter-1

    Integer apply_interest(Integer id):
      if(id > id_counter or id < 0):
        return 1
      accounts[id].apply_interest()
      return 0

    Float get_balance(Integer id):
      if(id > id_counter):
        system.out.printString("Invalid account number.\n")
        return -1.0
      return accounts[id].get_balance()

    Integer deposit(Integer id, Float amount):
      if(id > id_counter):
        system.out.printString("Invalid account number.\n")
        return 1

      accounts[id].deposit(amount)
      return 0

    Integer withdraw(Integer id, Float amount):
```

```
      if(id > id_counter):
        system.out.printString("Invalid account number.\n")
        return 1
      if(amount > accounts[id].get_balance()):
        return 1

      accounts[id].withdraw(amount)
      return 0

    Integer transfer(Integer from_id, Integer to_id, Float
    amount):
      if(from_id > id_counter):
        system.out.printString("Invalid account number.\n")
        return 1
      if(accounts[from_id].get_balance() < amount):
        system.out.printString("Insufficient funds.\n")
        return 1
      accounts[from_id].withdraw(amount)
      accounts[to_id].deposit(amount)
      return 0

    Float get_balance(Integer id, Float amount):
      if(id > id_counter):
        return -1.0
      return accounts[id].get_balance()


  protected:
    Integer id_counter
    Account[] accounts

/* Subclasses can come before classes if you like */
class Checking extends Account:
  public:
    init(Integer id, String name):
      super(id, name)

  refinement:
    Float apply_interest.rate():
      return 0.005

class Savings extends Account:
  public:
    init(Integer id, String name):
      super(id, name)

  refinement:
    Float apply_interest.rate():
      return 0.02

class Account:
  protected:
    void apply_interest(Boolean check):
      if (not (refinable(rate))):
        system.out.printString("Account must have some interest
    rate.\n")
        system.exit(1)
```

```

    init(Integer new_id, String name):
      super()
      apply_interest(false)

      id := new_id
      client := name
      balance := 0.0
      transactions := new Float[](100)
      trans_len := 0

  public:
    Integer get_id():
      return id

    String get_client_name():
      return client

    Float get_balance():
      return balance

    void apply_interest():
      balance *= (1.0 + (refine rate() to Float))

    Integer deposit(Float amount):
      if(amount < 0.0):
        return 1
      balance += amount
      transactions[trans_len] := amount
      trans_len += 1
      return 0

    Integer withdraw(Float amount):
      if(amount < 0.0):
        system.out.printString("Invalid number entered.\n")
        return 1
      if(balance < amount):
        system.out.printString("Insufficient funds.\n")
        return 1
      balance -= amount
      return 0

  private:
    Integer id
    String client
    Float balance
    Float[] transactions
    Integer trans_len


class Main:
  public:
    init():
      super()

  main(System system, String[] args):
    Bank citibank := new Bank()
```

```
Integer menu_lvl := 0
Integer menu_num := 0
Integer selection := new Integer()
Integer account_id := -1

while(true):
  if(menu_lvl = 0):
    system.out.printString("Please Select:\n1.Open New Account\n2.Manage Existing Account\n3.I'm the President!\n-> ")
    selection := system.in.scanInteger()
    account_id := -1
    menu_lvl := 1

  if(menu_lvl = 1):
    if(selection = 1):
      system.out.printString("Your Name Please:")
      String name := new String()
      name := system.in.scanString()
      Integer checking_id := citibank.open_checking(name)
      Integer savings_id := citibank.open_savings(name)

      system.out.printString("\nDear ")
      system.out.printString(name)
      system.out.printString("\n")
      system.out.printString("Your new checking account number: ")
      system.out.printInteger(checking_id)
      system.out.printString("\n")
      system.out.printString("Your new savings account number: ")
      system.out.printInteger(savings_id)
      system.out.printString("\n")
      selection := 0
      menu_lvl := 0
    else:
      if(selection = 2):
        if(account_id < 0):
          system.out.printString("Your Account Number Please: ")
          account_id := system.in.scanInteger()

        citibank.apply_interest(account_id)
        system.out.printString("Please Select:\n1.Check Balance\n2.Deposit\n3.Withdraw\n4.Transfer\n5.Exit\n-> ")
        menu_lvl := 2
        selection := system.in.scanInteger()
        if(selection = 5):
          selection := 0
          menu_lvl := 0
      else:
        if(selection = 3):
          selection := 2
          account_id := 0
          menu_lvl := 1

   if(menu_lvl = 2):
```

```
          if(selection = 1):
            system.out.printString("Your current balance: ")
            system.out.printFloat(citibank.get_balance(account_id))
            system.out.printString("\n")
            menu_lvl := 1
            selection := 2
          else:
            if(selection = 2):
              system.out.printString("Please enter the amount you want to deposit: ")
              Float amount := system.in.scanFloat()
              citibank.deposit(account_id, amount)
              menu_lvl := 1
              selection := 2
            else:
              if(selection = 3):
                system.out.printString("Pleaser enter the amount you want to withdraw: ")
                Float amount := system.in.scanFloat()
                citibank.withdraw(account_id, amount)
                menu_lvl := 1
                selection := 2
              else:
                if(selection = 4):
                  system.out.printString("Please enter the account number you want to transfer to: ")
                  Integer to_account := system.in.scanInteger()
                  system.out.printString("Please enter the amount you want to transfer: ")
                  Float amount := system.in.scanFloat()
                  citibank.transfer(account_id, to_account, amount)
                  menu_lvl := 1
                  selection := 2
```

Source 48: `demo/bank.gamma`

```
open Parser

(** Convert a whitespace file into a brace file. *)

(**
    Gracefully tell the programmer that they done goofed
    @param msg The descriptive error message to convey to the programmer
  *)
let wsfail msg = raise(Failure(msg))

(**
    Only allow spacing that is at the start of a line
    @param program A program as a list of tokens
    @return a list of tokens where the only white space is indentation, newlines,
    and colons (which count as a newline as it must be followed
```

```
    by them)
  *)
let indenting_space program =
    let rec space_indenting rtokens = function
        | NEWLINE::SPACE(n)::rest -> space_indenting (SPACE(n)::NEWLINE::rtokens) rest
        | COLON::SPACE(n)::rest -> space_indenting (SPACE(n)::COLON::rtokens) rest
        | SPACE(n)::rest -> space_indenting rtokens rest
        | token::rest -> space_indenting (token::rtokens) rest
        | [] -> List.rev rtokens in
    match (space_indenting [] (NEWLINE::program)) with
        | NEWLINE::rest -> rest
        | _ -> wsfail "Indenting should have left a NEWLINE at the start of program; did not."

(**
    Between LBRACE and RBRACE we ignore spaces and newlines; colons are errors in this context.
    It's not necessary that this be done after the above, but it is recommended.
    @param program A program in the form of a list of tokens
    @return A slightly slimmer program
  *)
let despace_brace program =
    let rec brace_despace depth tokens rtokens last =
        if depth > 0 then
            match tokens with
                | SPACE(_)::rest -> brace_despace depth rest rtokens last
                | NEWLINE::rest -> brace_despace depth rest rtokens last
                | COLON::_ -> wsfail "Colon inside brace scoping."
                | LBRACE::rest -> brace_despace (depth+1) rest (LBRACE::rtokens) last
                | RBRACE::rest -> let rtokens = if depth = 1
                     then SPACE(last)::NEWLINE::RBRACE::rtokens
                     else RBRACE::rtokens in
                     brace_despace (depth-1) rest rtokens last
                | token::rest -> brace_despace depth rest (token::rtokens) last
                | [] -> List.rev rtokens
        else
            match tokens with
                | SPACE(n)::rest -> brace_despace depth rest (SPACE(n)::rtokens) n
                | LBRACE::rest -> brace_despace (depth+1) rest (LBRACE::rtokens) last
                | token::rest -> brace_despace depth rest (token::rtokens) last
                | [] -> List.rev rtokens in
    brace_despace 0 program [] 0

(**
    Remove empty indentation -- SPACE followed by COLON or NEWLINE
```

```
    @param program A program as a list of tokens
    @return A program without superfluous indentation
  *)
let trim_lines program =
    let rec lines_trim tokens rtokens =
        match tokens with
            | [] -> List.rev rtokens
            | SPACE(_)::NEWLINE::rest -> lines_trim rest (NEWLINE::rtokens)
            | SPACE(_)::COLON::rest -> lines_trim rest (COLON::rtokens)
            | token::rest -> lines_trim rest (token::rtokens) in
    lines_trim program []

(**
    Remove consecutive newlines
    @param program A program as a list of tokens
    @return A program without consecutive newlines
  *)
let squeeze_lines program =
    let rec lines_squeeze tokens rtokens =
        match tokens with
            | [] -> List.rev rtokens
            | NEWLINE::NEWLINE::rest -> lines_squeeze (NEWLINE::rest) rtokens
            | COLON::NEWLINE::rest -> lines_squeeze (COLON::rest) rtokens (* scanner handled this though *)
            | token::rest -> lines_squeeze rest (token::rtokens) in
    lines_squeeze program []

(**
    Remove the initial space from a line but semantically note it
    @return an ordered pair of the number of spaces at the beginning
    of the line and the tokens in the line
*)
let spacing = function
    | SPACE(n)::rest -> (n, rest)
    | list           -> (0, list)

(**
    Remove spaces, newlines, and colons but semantically note their presence.
    @param program A full program (transformed by the above pipeline)
    @return a list of triples, one for each line. Each triple's first item is
    the number of spaces at the beginning of the line; the second item is the
    tokens in the line; the third is whether the line ended in a colon.
  *)
let tokens_to_lines program =
    let rec lines_from_tokens rline rlines = function
        | NEWLINE::rest ->
```

```
            (match rline with
                | [] -> lines_from_tokens [] rlines rest
                | _ -> let (spacer, line) = spacing (List.rev rline) in
                                lines_from_tokens [] ((spacer, line, false)::rlines) rest)
        | COLON::rest ->
            (match rline with
                | [] -> lines_from_tokens [] rlines rest
                | _ -> let (spacer, line) = spacing (List.rev rline) in
                                lines_from_tokens [] ((spacer, line, true)::rlines) rest)
        | [] ->
            (match rline with
                | [] -> List.rev rlines
                | _ -> let (spacer, line) = spacing (List.rev rline) in
                                lines_from_tokens [] ((spacer, line, false)::rlines) [])
        | token::rest -> lines_from_tokens (token::rline) rlines rest in
    lines_from_tokens [] [] program

(**
    Merge line continuatons given output from tokens_to_lines.
    Line n+1 continues n if n does not end in a colon and n+1 is more
    indented than n (or if line n is a continuation and they are both
    equally indented).
    @param program_lines The individual lines of the program
    @return The lines of the program with whitespace collapsed
  *)
let merge_lines program_lines =
    let rec lines_merge rlines = function
        | ((n1, _, _) as line1)::((n2, _, _) as line2)::rest when n1 >= n2 -> lines_merge (line1::rlines) (line2::rest)
        | (n, line1, false)::(_, line2, colon)::rest -> lines_merge rlines ((n, line1@line2, colon)::rest)
        | ((_, _, true) as line)::rest -> lines_merge (line::rlines) rest
        | line::[] -> lines_merge (line::rlines) []
        | [] -> List.rev rlines in
    lines_merge [] program_lines

(**
    Check if a given line needs a semicolon at the end
  *)
let rec needs_semi = function
    | [] -> true              (* General base case *)
    | RBRACE::[] -> false     (* The end of bodies do not require semicolons *)
    | SEMI::[] -> false       (* A properly terminated line does not require an additional semicolon *)
    | _::rest -> needs_semi rest (* Go through *)

```

```
(**
    Build a block. Consecutive lines of the same indentation with only the last ending
    in a colon are a 'block'. Blocks are just 'lines' merged together but joined with
    a semi colon when necessary.
    @param lines The full set of lines
    @return A list of blocks
  *)
let block_merge lines =
    let add_semi = function
        | (n, toks, true) -> (n, toks, true, false)
        | (n, toks, false) -> (n, toks, false, needs_semi toks)
    in
    let lines = List.map add_semi lines in
    let rec merge_blocks rblocks = function
        | (n1, line1, false, s1)::(n2, line2, colon, s2)::rest when n1 = n2 ->
            let newline = line1 @ (if s1 then [SEMI] else []) @ line2 in
            merge_blocks rblocks ((n1, newline, colon, s2)::rest)
        | (n, line, colon, _)::rest -> merge_blocks ((n, line, colon)::rblocks) rest
        | [] -> List.rev rblocks in
    merge_blocks [] lines

(** Make sure every line is terminated with a semi-colon when necessary *)
let terminate_blocks blocks =
    let rec block_terminate rblocks = function
        | (n, toks, false)::rest ->
            let terminated = if (needs_semi toks) then toks@[SEMI] else toks in
            block_terminate ((n, terminated, false)::rblocks) rest
        | other::rest ->
            block_terminate (other::rblocks) rest
        | [] -> List.rev rblocks in
    block_terminate [] blocks

(** Pops the stack and adds rbraces when necessary *)
let rec arrange n stack rtokens =
    match stack with
        | top::rest when n <= top -> arrange n rest (RBRACE::rtokens)
        | _ -> (stack, rtokens)

(**
     Take results of pipeline and finally adds braces. If blocks are merged
    then either consecutive lines differ in scope or there are colons.
    so now everything should be easy peasy (lemon squeezy).
  *)
let space_to_brace = function
    | [] -> []
```

```
    | linelist -> let rec despace_enbrace stack rtokens = function
        | [] -> List.rev ((List.map (function _ -> RBRACE) stack) @ rtokens)
        | (n, line, colon)::rest ->
            let (stack, rtokens) = arrange n stack rtokens in
            let (lbrace, stack) = if colon then ([LBRACE], n::stack) else ([], stack) in
            despace_enbrace stack (lbrace@(List.rev line)@rtokens) rest
        in despace_enbrace [] [] linelist

(** Drop the EOF from a stream of tokens, failing if not possible *)
let drop_eof program =
    let rec eof_drop rtokens = function
        | EOF::[] -> List.rev rtokens
        | EOF::rest -> raise(Failure("Misplaced EOF"))
        | [] -> raise(Failure("No EOF available."))
        | tk::tks -> eof_drop (tk::rtokens) tks in
    eof_drop [] program

(** Append an eof token to a program *)
let append_eof program =
    let rec eof_add rtokens = function
        | [] -> List.rev (EOF::rtokens)
        | tk::tks -> eof_add (tk::rtokens) tks in
    eof_add [] program

(** Run the entire pipeline *)
let convert program =
    (* Get rid of the end of file *)
    let noeof = drop_eof program in
    (* Indent in response to blocks *)
    let indented = indenting_space noeof in
    (* Collapse whitespace around braces *)
    let despaced = despace_brace indented in
    (* Get rid of trailing whitespace *)
    let trimmed = trim_lines despaced in
    (* Remove consequetive newlines *)
    let squeezed = squeeze_lines trimmed in
    (* Turn tokens into semantics *)
    let lines = tokens_to_lines squeezed in
    (* Consolidate those semantics *)
    let merged = merge_lines lines in
    (* Turn the semantics into blocks *)
    let blocks = block_merge merged in
    (* Put in the semicolons *)
    let terminated = terminate_blocks blocks in
    (* Turn the blocks into braces *)
    let converted = space_to_brace terminated in
    (* Put the eof on *)
    append_eof converted

(** A function to act like a lexfun *)
let lextoks toks =
    let tokens = ref (convert toks) in
```

```
    function _ ->
        match !tokens with
            | [] -> raise(Failure("Not even EOF given."))
            | tk::tks -> tokens := tks; tk
```

Source 49: `WhiteSpace.ml`

```
open Cast
open StringModules

let c_indent = "  "

let dispatches = ref []
let dispatchon = ref []
let dispatcharr = ref []

let matches type1 type2 = String.trim (GenCast.get_tname type1) = String.trim type2

let lit_to_str lit = match lit with
    | Ast.Int(i) -> "LIT_INT("^(string_of_int i)^")"
    | Ast.Float(f) -> "LIT_FLOAT("^(string_of_float f)^")"
    | Ast.String(s) -> "LIT_STRING(\"" ^ s ^ "\")"  (* escapes were escaped during lexing *)
    | Ast.Bool(b) ->if b then "LIT_BOOL(1)" else "LIT_BOOL(0)"

let stringify_unop op rop rtype =
    let (is_int, is_flt, is_bool) = (matches "Integer", matches "Float", matches "Boolean") in
    let is_type = (is_int rtype, is_flt rtype, is_bool rtype) in
    let type_capital = match is_type with
        | (true, _, _) -> "INTEGER"
        | (_, true, _) -> "FLOAT"
        | (_, _, true) -> "BOOLEAN"
        | (_, _, _)    -> raise(Failure "Imcompatible type with unop") in
    match op with
    | Ast.Arithmetic(Ast.Neg) -> "NEG_"^type_capital^"( "^rop^")"
    | Ast.CombTest(Ast.Not)   -> "NOT_"^type_capital^"( "^rop^")"
    | _   -> raise (Failure "Unknown operator")

let stringify_arith op suffix =
    match op with
    | Ast.Add  -> "ADD_"^suffix
    | Ast.Sub  -> "SUB_"^suffix
    | Ast.Prod -> "PROD_"^suffix
    | Ast.Div  -> "DIV_"^suffix
    | Ast.Mod  -> "MOD_"^suffix
    | Ast.Neg  ->  raise(Failure "Unary operator")
    | Ast.Pow  -> "POW_"^suffix
  (*| Ast.Pow  -> Format.sprintf "pow(%s,%s)" lop rop*)

let stringify_numtest op suffix = match op with
```

```
    | Ast.Eq   -> "NTEST_EQ_"^suffix
    | Ast.Neq  -> "NTEST_NEQ_"^suffix
    | Ast.Less -> "NTEST_LESS_"^suffix
    | Ast.Grtr -> "NTEST_GRTR_"^suffix
    | Ast.Leq  -> "NTEST_LEQ_"^suffix
    | Ast.Geq  -> "NTEST_GEQ_"^suffix

let stringify_combtest op suffix = match op with
    | Ast.And  -> "CTEST_AND_"^suffix
    | Ast.Or   -> "CTEST_OR_"^suffix
    | Ast.Nand -> "CTEST_NAND_"^suffix
    | Ast.Nor  -> "CTEST_NOR_"^suffix
    | Ast.Xor  -> "CTEST_XOR_"^suffix
    | Ast.Not  -> raise(Failure "Unary operator")

let stringify_binop op lop rop types =
    let (is_int, is_flt, is_bool) = (matches "Integer", matches "Float", matches "Boolean") in
    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
    let prefix = match is_type with
        | (true, _, _, true, _, _) -> "INT_INT"
        | (_, true, _, _, true, _) -> "FLOAT_FLOAT"
        | (true, _, _, _, true, _) -> "INT_FLOAT"
        | (_, true, _, true, _, _) -> "FLOAT_INT"
        | (_, _, true, _, _, true) -> "BOOL_BOOL"
        | (_, _, _, _, _, _)       -> raise(Failure(Format.sprintf "Binary operator applied to %s, %s" (fst types) (snd types))) in
    let suffix = prefix^"( "^lop^" , "^rop^" )" in
    match op with
    | Ast.Arithmetic(arith)  -> stringify_arith arith suffix
    | Ast.NumTest(numtest)   -> stringify_numtest numtest suffix
    | Ast.CombTest(combtest) -> stringify_combtest combtest suffix

let stringify_list stmtlist = String.concat "\n" stmtlist

let rec expr_to_cstr (exptype, expr_detail) = exprdetail_to_cstr expr_detail

and exprdetail_to_cstr castexpr_detail =
    let generate_deref obj index =
        let arrtype = fst obj in
        Format.sprintf "((struct %s*)(%s))[INTEGER_OF((%s))]" arrtype (expr_to_cstr obj) (expr_to_cstr index) in

    let generate_field obj field =
        let exptype = fst obj in
        Format.sprintf "(%s)->%s.%s" (expr_to_cstr obj) (GenCast.from_tname exptype) field in

    let generate_invocation recvr fname args =
        let this = Format.sprintf "((struct %s*)(%s))" (fst recvr) (expr_to_cstr recvr) in
        let vals = List.map expr_to_cstr args in
```

```
        Format.sprintf "%s(%s)" fname (String.concat ", " (this :: vals)) in

    let generate_vreference vname = function
        | Sast.Local -> vname
        | Sast.Instance(klass) -> Format.sprintf "(this->%s).%s" klass vname in

    let generate_allocation klass fname args =
        let vals = List.map expr_to_cstr args in
        let alloc = Format.sprintf "MAKE_NEW(%s)" klass in
        Format.sprintf "%s(%s)" fname (String.concat ", " (alloc :: vals)) in

    let generate_array_alloc _ fname args =
        let vals = List.map expr_to_cstr args in
        Format.sprintf "%s(%s)" fname (String.concat ", " vals) in

    let generate_refine args ret = function
        | Sast.Switch(_, _, dispatch) ->
          let vals = List.map expr_to_cstr args in
          Format.sprintf "%s(%s)" dispatch (String.concat ", " ("this"::vals))
        | _ -> raise(Failure("Wrong switch applied to refine -- compiler error.")) in

    let generate_refinable = function
        | Sast.Test(_, _, dispatchby) -> Format.sprintf "%s(this)" dispatchby
        | _ -> raise(Failure("Wrong switch applied to refinable -- compiler error.")) in

    match castexpr_detail with
    | This                               -> "this" (* There is no way this is right with implicit object passing *)
    | Null                               -> "NULL"
    | Id(vname, varkind)                 -> generate_vreference vname varkind
    | NewObj(classname, fname, args)     -> generate_allocation classname fname args
    | NewArr(arrtype, fname, args)       -> generate_array_alloc arrtype fname args
    | Literal(lit)                       -> lit_to_str lit
    | Assign((vtype, _) as memory, data) -> Format.sprintf "%s = ((struct %s*)(%s))" (expr_to_cstr memory) vtype (expr_to_cstr data)
    | Deref(carray, index)               -> generate_deref carray index
    | Field(obj, fieldname)              -> generate_field obj fieldname
    | Invoc(recvr, fname, args)          -> generate_invocation recvr fname args
    | Unop(op, expr)                     -> stringify_unop op (expr_to_cstr expr) (fst expr)
    | Binop(lop, op, rop)                -> stringify_binop op (expr_to_cstr lop) (expr_to_cstr rop) ((fst lop), (fst rop))
```

```
    | Refine(args, ret, switch)          -> generate_refine args ret switch
    | Refinable(switch)                  -> generate_refinable switch

and vdecl_to_cstr (vtype, vname) = Format.sprintf "struct %s*%s" vtype vname


let rec collect_dispatches_exprs exprs = List.iter collect_dispatches_expr exprs
and collect_dispatches_stmts stmts = List.iter collect_dispatches_stmt stmts
and collect_dispatches_expr (_, detail) = match detail with
    | This -> ()
    | Null -> ()
    | Id(_,_) -> ()
    | NewObj(_, _, args) -> collect_dispatches_exprs args
    | NewArr(arrtype, fname, args) -> collect_dispatch_arr arrtype fname args
    | Literal(_) -> ()
    | Assign(mem, data) -> collect_dispatches_exprs [mem; data]
    | Deref(arr, idx) -> collect_dispatches_exprs [arr; idx]
    | Field(obj, _) -> collect_dispatches_expr obj
    | Invoc(recvr, _, args) -> collect_dispatches_exprs (recvr::args)
    | Unop(_, expr) -> collect_dispatches_expr expr
    | Binop(l, _, r) -> collect_dispatches_exprs [l; r]
    | Refine(args, ret, switch) -> collect_dispatch args ret switch
    | Refinable(switch) -> collect_dispatch_on switch
and collect_dispatches_stmt = function
    | Decl(_, Some(expr), _) -> collect_dispatches_expr expr
    | Decl(_, None, _) -> ()
    | If(iflist, env) -> collect_dispatches_clauses iflist
    | While(pred, body, _) -> collect_dispatches_expr pred; collect_dispatches_stmts body
    | Expr(expr, _) -> collect_dispatches_expr expr
    | Return(Some(expr), _) -> collect_dispatches_expr expr
    | Super(_, _, args) -> collect_dispatches_exprs args
    | Return(None, _) -> ()
and collect_dispatches_clauses pieces =
    let (preds, bodies) = List.split pieces in
    collect_dispatches_exprs (Util.filter_option preds);
    collect_dispatches_stmts (List.flatten bodies)
and collect_dispatch args ret = function
    | Sast.Switch(klass, cases, dispatch) -> dispatches := (klass, ret, (List.map fst args), dispatch, cases)::(!dispatches);
    | Sast.Test(_, _, _) -> raise(Failure("Impossible (wrong switch -- compiler error)"))
and collect_dispatch_on = function
    | Sast.Test(klass, klasses, dispatchby) -> dispatchon := (klass, klasses, dispatchby)::(!dispatchon);
    | Sast.Switch(_, _, _) -> raise(Failure("Impossible (wrong switch -- compiler error)"))
and collect_dispatch_func func = collect_dispatches_stmts func.
```

```
    body
and collect_dispatch_arr arrtype fname args =
    dispatcharr := (arrtype, fname, args)::(!dispatcharr)

(**
    Takes an element from the dispatchon list and generates the test function for refinable.
    @param  klasses - list of klasses in which the refinable method is defined for the method
             fuid - unique function name for the test function.
    @return true or false
    Checks if the object on which refinable was invoked has an associated refinable method
    dispatched via this function that's being generated in one of the classes.
**)


let generate_testsw (klass, klasses, fuid) =
    let test klass = Format.sprintf "\tif ( IS_CLASS(this, \"%s\") ) return LIT_BOOL(1);" (String.trim klass) in
    let cases = String.concat "\n" (List.map test klasses) in
    let body = Format.sprintf "%s\n\treturn LIT_BOOL(0);" cases in
    Format.sprintf "struct t_Boolean *%s( struct %s*this )\n{\n%s\n}\n\n" fuid klass body

(**
     Takes a dispatch element of the global dispatches list
     And generates the dispatch function - dispatcher which dispatches
     calls to refinable methods based on the RTTI of the this.
     @param ret - return type of the function
            args - arguments to the dispatcher and the dispatched method
            dispatch uid - unique function name for the dispatcher
            cases - list of classes and their corresponding uid of the invokable refinable methods.
**)

let generate_refinesw (klass, ret, args, dispatchuid, cases) =
    let rettype = match ret with
        | None -> "void "
        | Some(atype) -> Format.sprintf "struct %s*" atype in
    let this = (Format.sprintf "struct %s*" klass, "this") in
    let formals = List.mapi (fun i t -> (Format.sprintf "struct %s*" t, Format.sprintf "varg_%d" i)) args in
    let signature = String.concat ", " (List.map (fun (t, v) -> t ^ v) (this::formals)) in
    let actuals = List.map snd formals in
    let withthis kname = String.concat ", " ((Format.sprintf "(struct %s*) this" kname)::actuals) in
    let invoc fuid kname = Format.sprintf "%s(%s)" fuid (withthis kname) in
    let execute fuid kname = match ret with
        | None -> Format.sprintf "%s; return;" (invoc fuid kname
```

```
    )
        | Some(atype) -> Format.sprintf "return ((struct %s*)(%s));" (String.trim atype) (invoc fuid kname) in
    let unroll_case (kname, fuid) =
        Format.sprintf "\tif( IS_CLASS( this, \"%s\") )\n\t\t{ %s }\n" (String.trim kname) (execute fuid kname) in
    let generated = List.map unroll_case cases in
    let fail = Format.sprintf "REFINE_FAIL(\"%s\")" (String.trim klass) in
    Format.sprintf "%s%s(%s)\n{\n%s\n\t%s\n}\n\n" rettype dispatchuid signature (String.concat "" generated) fail

let generate_arrayalloc (arrtype, fname, args) =
    let params = List.mapi (fun i _ -> Format.sprintf "struct %s*v_dim%d" (GenCast.get_tname "Integer") i) args in
    match List.length params with
        | 1 -> Format.sprintf "struct %s*%s(%s) {\n\treturn ONE_DIM_ALLOC(struct %s, INTEGER_OF(v_dim0));\n}\n" arrtype fname (String.concat ", " params) arrtype
        | _ -> raise(Failure("Only one dimensional arrays currently supported."))

(**
    Take a list of cast_stmts and return a body of c statements
    @param stmtlist A list of statements
    @return A body of c statements
*)
let rec cast_to_c_stmt indent cast =
    let indents = String.make indent '\t' in
    let stmts = cast_to_c_stmtlist (indent+1) in

    let cstmt = match cast with
        | Decl((vtype, _) as vdecl, Some(expr), env) -> Format.sprintf "%s = ((struct %s*)(%s));" (vdecl_to_cstr vdecl) vtype (expr_to_cstr expr)
        | Decl(vdecl, None, env) -> Format.sprintf "%s;" (vdecl_to_cstr vdecl)
        | If(iflist, env) -> cast_to_c_if_chain indent iflist
        | While(pred, [], env) -> Format.sprintf "while ( BOOL_OF( %s ) ) { }" (expr_to_cstr pred)
        | While(pred, body, env) -> Format.sprintf "while ( BOOL_OF( %s ) ) {\n%s\n%s}" (expr_to_cstr pred) (stmts body) indents
        | Expr(expr, env) -> Format.sprintf "( %s );" (expr_to_cstr expr)
        | Return(Some(expr), env) -> Format.sprintf "return ( %s );" (expr_to_cstr expr)
        | Return(_, env) -> "return;"
        | Super(klass, fuid, []) -> Format.sprintf "%s((struct %s*)(this));" fuid (GenCast.get_tname klass)
        | Super(klass, fuid, args) -> Format.sprintf "%s((struct %s*)(this), %s);" fuid (GenCast.get_tname klass) (String.concat ", " (List.map expr_to_cstr args)) in
    indents ^ cstmt

and cast_to_c_stmtlist indent stmts =
    String.concat "\n" (List.map (cast_to_c_stmt indent) stmts)
```

```
and cast_to_c_if_pred = function
    | None -> ""
    | Some(ifpred) -> Format.sprintf "if ( BOOL_OF( %s ) )" (expr_to_cstr ifpred)

and cast_to_c_if_chain indent pieces =
    let indents = String.make indent '\t' in
    let stmts = cast_to_c_stmtlist (indent + 1) in
    let combine (pred, body) = Format.sprintf "%s {\n%s\n%s}" (cast_to_c_if_pred pred) (stmts body) indents in
    String.concat " else " (List.map combine pieces)


let cast_to_c_class_struct klass_name ancestors =
    let ancestor_var (vtype, vname) = Format.sprintf "struct %s *%s;" vtype vname in
    let ancestor_vars vars = String.concat "\n\t\t" (List.map ancestor_var vars) in
    let internal_struct (ancestor, vars) = match vars with
        | [] -> Format.sprintf "struct { BYTE empty_vars; } %s;" ancestor
        | _ -> Format.sprintf "struct {\n\t\t%s\n\t} %s;\n" (ancestor_vars vars) ancestor in
    let internals = String.concat "\n\n\t" (List.map internal_struct ancestors) in
    let meta = "\tClassInfo *meta;" in
    Format.sprintf "struct %s {\n%s\n\n\t%s\n};\n\n" (String.trim klass_name) meta internals

let cast_to_c_func cfunc =
    let ret_type = match cfunc.returns with
        | None -> "void "
        | Some(atype) -> Format.sprintf "struct %s*" atype in
    let body = match cfunc.body with
        | [] -> " { }"
        | body -> Format.sprintf "\n{\n%s\n}" (cast_to_c_stmtlist 1 body) in
    let params = if cfunc.static = false then (GenCast.get_tname cfunc.inklass, "this")::cfunc.formals
                else cfunc.formals in
    let signature = String.concat ", " (List.map (fun (t,v) -> "struct " ^ t ^ "*" ^ v) params) in
    if cfunc.builtin then Format.sprintf "/* Place-holder for %s%s(%s) */" ret_type cfunc.name signature
    else Format.sprintf "\n%s%s(%s)%s\n" ret_type cfunc.name signature body

let cast_to_c_proto cfunc =
    let ret_type = match cfunc.returns with
        | None -> "void "
        | Some(atype) -> Format.sprintf "struct %s*" atype in
    let first = if cfunc.static then [] else [(GenCast.get_tname cfunc.inklass, "this")] in
    let params = first@cfunc.formals in
    let types = String.concat ", " (List.map (fun (t,v) -> "struct " ^ t ^ "*") params) in
```

```
    let signature = Format.sprintf "%s%s(%s);" ret_type cfunc.name types in
    if cfunc.builtin then Format.sprintf "" else signature

let cast_to_c_proto_dispatch_arr (arrtype, fname, args) =
    let int = Format.sprintf "struct %s*" (GenCast.get_tname "Integer") in
    let params = List.map (fun _ -> int) args in
    Format.sprintf "struct %s*%s(%s);" arrtype fname (String.concat ", " params)

let cast_to_c_proto_dispatch_on (klass, _, uid) =
    Format.sprintf "struct t_Boolean *%s(struct %s *);" uid klass

let cast_to_c_proto_dispatch (klass, ret, args, uid, _) =
    let types = List.map (fun t -> "struct " ^ t ^ "*") (klass::args) in
    let proto rtype = Format.sprintf "struct %s*%s(%s);" rtype uid (String.concat ", " types) in
    match ret with
        | None -> proto "void"
        | Some(t) -> proto t

let cast_to_c_main mains =
    let main_fmt = ""^^"\tif (!strncmp(gmain, \"%s\", %d)) { %s(&global_system, str_args); return 0; }" in
    let for_main (klass, uid) = Format.sprintf main_fmt klass (String.length klass + 1) uid in
    let switch = String.concat "\n" (List.map for_main mains) in
    let cases = Format.sprintf "\"%s\"" (String.concat ", " (List.map fst mains)) in
    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

let commalines input n =
    let newline string = String.length string >= n in
    let rec line_builder line rlines = function
        | [] -> List.map String.trim (List.rev (line::rlines))
        | str::rest ->
            let comma = match rest with [] -> false | _ -> true in
            let str = if comma then str ^ ", " else str in
            if newline line then line_builder str (line::rlines) rest
            else line_builder (line ^ str) rlines rest in
    match input with
        | [] -> []
        | [one] -> [one]
        | str::rest -> line_builder (str ^ ", ") [] rest

let print_class_strings = function
    | [] -> raise(Failure("Not even built in classes?"))
    | classes -> commalines (List.map (fun k -> "\"" ^ k ^ "\"") classes) 75
```

```
let print_class_enums = function
    | [] -> raise(Failure("Not even built in classes?"))
    | first::rest ->
        let first = first ^ " = 0" in
        commalines (List.map String.uppercase (first::rest)) 75

let setup_meta klass =
    Format.sprintf "ClassInfo M_%s;" klass

let meta_init bindings =
    let to_ptr klass = Format.sprintf "m_classes[%s]" (String.trim (String.uppercase (GenCast.get_tname klass))) in
    let init (klass, ancestors) =
        let ancestors_strings = String.concat ", " (List.map to_ptr ancestors) in
        Format.sprintf "class_info_init(&M_%s, %d, %s);" klass (List.length ancestors) ancestors_strings in
    let bindings = List.filter (fun (k, _) -> not (StringSet.mem (GenCast.get_tname k) GenCast.built_in_names)) bindings in
    let inits = List.map init bindings in
    let inits = List.map (Format.sprintf "\t%s") inits in
    let built_in_init = "\tinit_built_in_infos();" in
    Format.sprintf "void init_class_infos() {\n%s\n}\n" (String.concat "\n" (built_in_init::inits))

let cast_to_c ((cdefs, funcs, mains, ancestry) : Cast.program) channel =
    let out string = Printf.fprintf channel "%s\n" string in
    let noblanks = function
        | "" -> ()
        | string -> Printf.fprintf channel "%s\n" string in
    let incl file = out (Format.sprintf "#include \"%s.h\"\n" file) in

    let comment string =
        let comments = Str.split (Str.regexp "\n") string in
        let commented = List.map (Format.sprintf " * %s") comments in
        out (Format.sprintf "\n\n/*\n%s\n */" (String.concat "\n" commented)) in

    let func_compare f g =
       let strcmp = Pervasives.compare f.name g.name in
       if f.builtin = g.builtin then strcmp else if f.builtin then -1 else 1 in
    let funcs = List.sort func_compare funcs in

    comment "Passing over code to find dispatch data.";
    List.iter collect_dispatch_func funcs;

    comment "Gamma preamble -- macros and such needed by various things";
    incl "gamma-preamble";

    comment "Ancestry meta-info to link to later.";
    let classes = List.map (fun (kls, _) -> String.trim (GenCast.get_tname kls)) (StringMap.bindings ancestry) in
```

```
    let class_strs = List.map (Format.sprintf "\t%s") (print_class_strings classes) in
    out (Format.sprintf "char *m_classes[] = {\n%s\n};" (String.concat "\n" class_strs));

    comment "Enums used to reference into ancestry meta-info strings.";
    let class_enums = List.map (Format.sprintf "\t%s") (print_class_enums classes) in
    out (Format.sprintf "enum m_class_idx {\n%s\n};" (String.concat "\n" class_enums));

    comment "Header file containing meta information for built in classes.";
    incl "gamma-builtin-meta";

    comment "Meta structures for each class.";
    let print_meta (klass, ancestors) =
        if StringSet.mem (GenCast.get_tname klass) GenCast.built_in_names then ()
        else out (setup_meta klass) in
    List.iter print_meta (StringMap.bindings ancestry);
    out "";
    out (meta_init (StringMap.bindings ancestry));

    comment "Header file containing structure information for built in classes.";
    incl "gamma-builtin-struct";

    comment "Structures for each of the objects.";
    let print_class klass data =
        if StringSet.mem klass GenCast.built_in_names then ()
        else out (cast_to_c_class_struct klass data) in
    StringMap.iter print_class cdefs;

    comment "Header file containing information regarding built in functions.";
    incl "gamma-builtin-functions";

    comment "All of the function prototypes we need to do magic.";
    List.iter (fun func -> noblanks (cast_to_c_proto func)) funcs;

    comment "All the dispatching functions we need to continue the magic.";
    List.iter (fun d -> out (cast_to_c_proto_dispatch_on d)) (!dispatchon);
    List.iter (fun d -> out (cast_to_c_proto_dispatch d)) (!dispatches);

    comment "Array allocators also do magic.";
    List.iter (fun d -> out (cast_to_c_proto_dispatch_arr d)) (!dispatcharr);

    comment "All of the functions we need to run the program.";
    List.iter (fun func -> out (cast_to_c_func func)) funcs;
```

```

    comment "Dispatch looks like this.";
    List.iter (fun d -> out (generate_testsw d)) (!dispatchon);
    List.iter (fun d -> out (generate_refinesw d)) (!dispatches);

    comment "Array allocators.";
    List.iter (fun d -> out (generate_arrayalloc d)) (!dispatcharr);

    comment "The main.";
    out (cast_to_c_main mains);
```

Source 50: `GenC.ml`

```
open Ast
open Variables
open StringModules

let rec get_vars_formals = function
    | [] -> StringSet.empty
    | [(_,var)] -> StringSet.singleton var
    | (_,var)::tl -> StringSet.add var (get_vars_formals tl)

let _ =
    let func = List.hd (Debug.get_example_longest_body "Multi" "Collection") in
    let stmts = func.body in
    let prebound = get_vars_formals func.formals in
    let free_variables = free_vars prebound stmts in
    StringSet.iter (Printf.printf "%s\n") free_variables
```

Source 51: `freevars.ml`

```
let debug_print tokens =
    let ptoken header tokens =
        Inspector.pprint_token_list header tokens;
        print_newline () in
    let plines header lines =
        Inspector.pprint_token_lines header lines;
        print_newline () in
    begin
        ptoken "Input:      " tokens;
        let tokens = WhiteSpace.drop_eof tokens in
        ptoken "No EOF      " tokens;
        let tokens = WhiteSpace.indenting_space tokens in
        ptoken "Indented:   " tokens;
        let tokens = WhiteSpace.despace_brace tokens in
        ptoken "In-Brace:   " tokens;
        let tokens = WhiteSpace.trim_lines tokens in
        ptoken "Trimmed:    " tokens;
        let tokens = WhiteSpace.squeeze_lines tokens in
        ptoken "Squeezed:   " tokens;
```

```
        let lines = WhiteSpace.tokens_to_lines tokens in
        plines "Lines:      " lines;
        let lines = WhiteSpace.merge_lines lines in
        plines "Merged:     " lines;
        let lines = WhiteSpace.block_merge lines in
        plines "Blocks:     " lines;
        let tokens = WhiteSpace.space_to_brace lines in
        ptoken "Converted:  " tokens;
        let tokens = WhiteSpace.append_eof tokens in
        ptoken "With EOF:   " tokens
    end

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

Source 52: `streams.ml`

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

Source 53: `BuiltIns.mli`

```
open Parser

let descan = Inspector.descan

let rec indenter depth indent =
    for i = 1 to depth do print_string indent done

(* Unscan a sequence of tokens. Requires sanitized stream *)
let rec clean_unscan depth indent = function
    (* ARRAY / LBRACKET RBRACKET ambiguity... *)
    | LBRACKET::RBRACKET::rest ->
        print_string ((descan LBRACKET) ^ " " ^ (descan RBRACKET));
        clean_unscan depth indent rest
    | LBRACE::rest ->
        print_string (descan LBRACE);
        print_newline ();
        indenter (depth+1) indent;
        clean_unscan (depth+1) indent rest
    | SEMI::RBRACE::rest ->
        print_string (descan SEMI);
        clean_unscan depth indent (RBRACE::rest)
    | RBRACE::RBRACE::rest ->
        print_newline ();
        indenter (max (depth-1) 0) indent;
        print_string (descan RBRACE);
        clean_unscan (max (depth-1) 0) indent (RBRACE::rest)
```

```
    | RBRACE::rest ->
        print_newline ();
        indenter (depth-1) indent;
        print_string (descan RBRACE);
        print_newline ();
        indenter (depth-1) indent;
        clean_unscan (max (depth-1) 0) indent rest
    | SEMI::rest ->
        print_string (descan SEMI);
        print_newline ();
        indenter depth indent;
        clean_unscan depth indent rest
    | EOF::[] ->
        print_newline ()
    | EOF::_ ->
        raise(Failure("Premature end of file."))
    | token::rest ->
        print_string (descan token);
        print_string " ";
        clean_unscan depth indent rest
    | [] ->
        print_newline ()

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

Source 54: `canonical.ml`

```
open Ast
open StringModules

(** Module to contain global class hierarchy type declarations
    *)

(** A full class record table as a type *)
type class_data = {
    known : StringSet.t; (** Set of known class names *)
    classes : class_def lookup_map; (** class name -> class def
    map *)
    parents : string lookup_map; (** class name -> parent name
    map *)
    children : (string list) lookup_map; (** class name ->
    children list map *)
    variables : (class_section * string) lookup_table; (** class
     name -> var name -> (section, type) map *)
    methods : (func_def list) lookup_table; (** class name ->
    method name -> func_def list map *)
    refines : (func_def list) lookup_table; (** class name ->
    host.refinement -> func_def list map *)
    mains : func_def lookup_map; (** class name -> main map *)
    ancestors : (string list) lookup_map; (** class name ->
    ancestor list (given to Object) *)
    distance : int lookup_table; (** subtype -> supertype -> #
    hops map *)
```

```
    refinable : (func_def list) lookup_table (** class -> host -> refinements (in subclasses) *)
}

(**
    All the different types of non-compiler errors that can occur (programmer errors)
  *)
type class_data_error
    = HierarchyIssue of string
    | DuplicateClasses of string list
    | DuplicateVariables of (string * string list) list
    | DuplicateFields of (string * (string * string) list) list
    | UnknownTypes of (string * (string * string) list) list
    | ConflictingMethods of (string * (string * string list) list) list
    | ConflictingInherited of (string * (string * string list) list) list
    | PoorlyTypedSigs of (string * (string * string option * (string * string) list) list) list
    | Uninstantiable of string list
    | ConflictingRefinements of (string * (string * string list) list) list
    | MultipleMains of string list
```

Source 55: `GlobalData.mli`

```
{
  open Parser

  (** The general lexographic scanner for Gamma *)

  (**
      Build a string from a list of characters
      from: http://caml.inria.fr/mantis/view.php?id=5367
      @param l The list to be glued
      @return A string of the characters in the list glued together
  *)
  let implode l =
    let res = String.create (List.length l) in
    let rec imp i = function
    | [] -> res
    | c :: l -> res.[i] <- c; imp (i + 1) l in
    imp 0 l

  (**
      Explode a string into a list of characters
      @param s The string to be exploded
      @return A list of the characters in the string in order
  *)
  let explode s =
    let rec exploder idx l =
      if idx < 0
        then l
```

```
        else exploder (idx-1) (s.[idx] :: l) in
    exploder (String.length s - 1) []

  (**
      A generic function to count the character-spaces of a
    character. (I.e. weight tabs more heavily)
  *)
  let spacecounter = function
    | '\t' -> 8
    | _    -> 1

  (**
      Count the space width of a string using the spacecounter
    function
      @param s The string to be evaluated
      @return The effective width of the string when rendered
  *)
  let spacecount s =
    let spaces = List.map spacecounter (explode s) in
    List.fold_left (+) 0 spaces

  (**/**)
  let line_number = ref 1
  (**/**)

  (**
    Count the lines in a series of vertical spacing characters.
    Please note that as of now, it is not intelligent enough to
    understand
    that \n\r should be counted as one. It seems like an
    oversized-amount
    of work for something we will never effectively need.
    @param v The vertical spacing series string
  *)
  let count_lines v = (line_number := !line_number + String.
    length v)

  (**
      Gracefully tell the programmer that they done goofed
      @param msg The descriptive error message to convey to the
    programmer
  *)
  let lexfail msg =
    raise (Failure("Line " ^ string_of_int !line_number ^ ": " ^
     msg))
}

let digit = ['0'-'9']
let lower = ['a'-'z']
let upper = ['A'-'Z']
let alpha = lower | upper
let ualphanum = '_' | alpha | digit

(* horizontal spacing: space & tab *)
let hspace = [' ' '\t']

(* vertical spaces: newline (line feed), carriage return,
```

```
    vertical tab, form feed *)
let vspace = ['\n' '\r' '\011' '\012']


rule token = parse
  (* Handling whitespace mode *)
  | hspace+ as s                { SPACE(spacecount s) }
  | ':' hspace* (vspace+ as v)   { count_lines v; COLON }
  | vspace+ as v                { count_lines v; NEWLINE }

  (* Comments *)
  | "/*"                        { comment 0 lexbuf }

  (* Boolean Tests & Values *)
  | "refinable"                 { REFINABLE }
  | "and"                       { AND }
  | "or"                        { OR }
  | "xor"                       { XOR }
  | "nand"                      { NAND }
  | "nor"                       { NOR }
  | "not"                       { NOT }
  | "true"                      { BLIT(true) }
  | "false"                     { BLIT(false) }
  | "="                         { EQ }
  | "<>"                        { NEQ }
  | "=/="                       { NEQ }
  | '<'                         { LT }
  | "<="                        { LEQ }
  | ">"                         { GT }
  | ">="                        { GEQ }

  (* Grouping [args, arrays, code, etc] *)
  | "[]"                        { ARRAY }
  | '['                         { LBRACKET }
  | ']'                         { RBRACKET }
  | '('                         { LPAREN }
  | ')'                         { RPAREN }
  | '{'                         { LBRACE }
  | '}'                         { RBRACE }

  (* Punctuation for the sytnax *)
  | ';'                         { SEMI }
  | ','                         { COMMA }

  (* Arithmetic operations *)
  | '+'                         { PLUS }
  | '-'                         { MINUS }
  | '*'                         { TIMES }
  | '/'                         { DIVIDE }
  | '%'                         { MOD }
  | '^'                         { POWER }

  (* Arithmetic assignment *)
  | "+="                        { PLUSA }
  | "-="                        { MINUSA }
  | "*="                        { TIMESA }
  | "/="                        { DIVIDEA }
```

```
    | "%="                           { MODA }
    | "^="                           { POWERA }

    (* Control flow *)
    | "if"                           { IF }
    | "else"                         { ELSE }
    | "elsif"                        { ELSIF }
    | "while"                        { WHILE }
    | "return"                       { RETURN }

    (* OOP Stuff *)
    | "class"                        { CLASS }
    | "extends"                      { EXTEND }
    | "super"                        { SUPER }
    | "init"                         { INIT }

    (* Pre defined types / values *)
    | "null"                         { NULL }
    | "void"                         { VOID }
    | "this"                         { THIS }

    (* Refinement / specialization related *)
    | "refine"                       { REFINE }
    | "refinement"                   { REFINES }
    | "to"                           { TO }

    (* Access *)
    | "private"                      { PRIVATE }
    | "public"                       { PUBLIC }
    | "protected"                    { PROTECTED }

    (* Miscellaneous *)
    | '.'                            { DOT }
    | "main"                         { MAIN }
    | "new"                          { NEW }
    | ":="                           { ASSIGN }

    (* Variable and Type IDs *)
    | '_'? lower ualphanum* as vid        { ID(vid) }
    | upper ualphanum* as tid             { TYPE(tid) }

    (* Literals *)
    | digit+ as inum                 { ILIT(int_of_string inum) }
    | digit+ '.' digit+ as fnum      { FLIT(float_of_string fnum) }
    | '"'                            { stringlit [] lexbuf }

    (* Some type of end, for sure *)
    | eof                            { EOF }
    | _ as char { lexfail("Illegal character " ^ Char.escaped char ) }

and comment level = parse
    (* Comments can be nested *)
    | "/*"            { comment (level+1) lexbuf }
    | "*/"            { if level = 0 then token lexbuf else comment (level-1) lexbuf }
    | eof             { lexfail("File ended inside comment.") }
```

```
  | vspace+ as v  { count_lines v; comment level lexbuf }
  | _             { comment level lexbuf }

and stringlit chars = parse
  (* Accept valid C string literals as that is what we will output directly *)
  | '\\'           { escapechar chars lexbuf }
  | eof            { lexfail("File ended inside string literal") }
  | vspace as char { lexfail("Line ended inside string literal (" ^ Char.escaped char ^ " used): " ^ implode(List.rev chars)) }
  | '"'            { SLIT(implode(List.rev chars)) }
  | _ as char      { stringlit (char::chars) lexbuf }

and escapechar chars = parse
  (* Accept valid C escape sequences *)
  | ['a' 'b' 'f' 'n' 'r' 't' 'v' '\\' '"' '0'] as char {
      stringlit (char :: '\\' :: chars) lexbuf
    }
  | eof       { lexfail("File ended while seeking escape character") }
  | _ as char { lexfail("Illegal escape character:  \\" ^ Char.escaped(char)) }
```

Source 56: `scanner.mll`

```
open Ast
open Sast
open Klass
open StringModules
open Util
open GlobalData

(** Module to take an AST and build the sAST out of it. *)

(**
    Update an environment to have a variable
    @param mode The mode the variable is in (instance, local)
    @param vtype The type of the variable
    @param vname The name of the variable
    @return A function that will update an environment passed to it.
  *)
let env_update mode (vtype, vname) env = match map_lookup vname env, mode with
    | None, _ -> StringMap.add vname (vtype, mode) env
    | Some((otype, Local)), Local -> raise(Failure("Local variable " ^ vname ^ " loaded twice, once with type " ^ otype ^ " and then with type " ^ vtype ^ "."))
    | _, Local -> StringMap.add vname (vtype, mode) env
    | _, _ -> raise(Failure("Instance variable declared twice in ancestry chain -- this should have been detected earlier; compiler error."))
let env_updates mode = List.fold_left (fun env vdef ->
```

```
    env_update mode vdef env)
let add_ivars klass env level =
    let sects = match level with
        | Publics -> [Publics]
        | Protects -> [Publics; Protects]
        | Privates -> [Publics; Protects; Privates]
        | _ -> raise(Failure("Inappropriate class section -
    access level.")) in
    let filter (s, _) = List.mem s sects in
    let vars = Klass.klass_to_variables klass in
    let eligible = List.flatten (List.map snd (List.filter
    filter vars)) in
    env_updates (Instance(klass.klass)) env eligible

(** Marker for being in the current class -- ADT next time *)
let current_class = "_CurrentClassMarker_"

(** Marker for the null type -- ADT next time *)
let null_class = "_Null_"

(** Empty environment *)
let empty_environment = StringMap.empty

(** Return whether an expression is a valid lvalue or not *)
let is_lvalue (expr : Ast.expr) = match expr with
    | Ast.Id(_) -> true
    | Ast.Field(_, _) -> true
    | Ast.Deref(_, _) -> true
    | _ -> false

(**
    Map a literal value to its type
    @param litparam a literal
    @return A string representing the type.
  *)
let getLiteralType litparam = match litparam with
    | Ast.Int(i) -> "Integer"
    | Ast.Float(f) -> "Float"
    | Ast.String(s) -> "String"
    | Ast.Bool(b) -> "Boolean"

(**
    Map a return type string option to a return type string
    @param ret_type The return type.
    @return The return type -- Void or its listed type.
  *)
let getRetType ret_type = match ret_type with
    | Some(retval) -> retval
    | None -> "Void"

(**
    Update a refinement switch based on updated data.
  *)
let rec update_refinements_stmts klass_data kname mname = List.
    map (update_refinements_stmt klass_data kname mname)
and update_refinements_exprs klass_data kname mname = List.map (
    update_refinements_expr klass_data kname mname)
```

```
and update_refinements_expr klass_data kname mname (atype, expr) =
    let doexp = update_refinements_expr klass_data kname mname in
    let doexps = update_refinements_exprs klass_data kname mname in

    let get_refine rname arglist desired uid =
        let argtypes = List.map fst arglist in
        let refines = Klass.refine_on klass_data kname mname rname argtypes desired in
        let switch = List.map (fun (f : Ast.func_def) -> (f.inklass, f.uid)) refines in
        (getRetType desired, Sast.Refine(rname, arglist, desired, Switch(kname, switch, uid))) in

    let get_refinable rname uid =
        let refines = Klass.refinable_lookup klass_data kname mname rname in
        let klasses = List.map (fun (f : Ast.func_def) -> f.inklass) refines in
        ("Boolean", Sast.Refinable(rname, Test(kname, klasses, uid))) in

    match expr with
        | Sast.Refine(rname, args, desired, Switch(_, _, uid)) -> get_refine rname args desired uid
        | Sast.Refine(_, _, _, _) -> raise(Failure("Test in switch."))
        | Sast.Refinable(rname, Test(_, _, uid)) -> get_refinable rname uid
        | Sast.Refinable(_, _) -> raise(Failure("Switch in test."))

        | Sast.Anonymous(_, _, _) -> raise(Failure("Anonymous detected during reswitching."))

        | Sast.This -> (atype, Sast.This)
        | Sast.Null -> (atype, Sast.Null)
        | Sast.Id(id) -> (atype, Sast.Id(id))
        | Sast.NewObj(klass, args, uid) -> (atype, Sast.NewObj(klass, doexps args, uid))
        | Sast.Literal(lit) -> (atype, Sast.Literal(lit))
        | Sast.Assign(l, r) -> (atype, Sast.Assign(doexp l, doexp r))
        | Sast.Deref(l, r) -> (atype, Sast.Deref(doexp l, doexp r))
        | Sast.Field(e, m) -> (atype, Sast.Field(doexp e, m))
        | Sast.Invoc(r, m, args, uid) -> (atype, Sast.Invoc(doexp r, m, doexps args, uid))
        | Sast.Unop(op, e) -> (atype, Sast.Unop(op, doexp e))
        | Sast.Binop(l, op, r) -> (atype, Sast.Binop(doexp l, op, doexp r))
and update_refinements_stmt klass_data kname mname stmt =
    let doexp = update_refinements_expr klass_data kname mname in
    let doexps = update_refinements_exprs klass_data kname mname
```

```
         in
    let dostmts = update_refinements_stmts klass_data kname mname in
    let docls = update_refinements_clauses klass_data kname mname in

    match stmt with
        | Sast.Decl(_, None, _) as d -> d
        | Sast.Decl(vdef, Some(e), env) -> Sast.Decl(vdef, Some(doexp e), env)
        | Sast.If(pieces, env) -> Sast.If(docls pieces, env)
        | Sast.While(pred, body, env) -> Sast.While(doexp pred, dostmts body, env)
        | Sast.Expr(expr, env) -> Sast.Expr(doexp expr, env)
        | Sast.Return(None, _) as r -> r
        | Sast.Return(Some(e), env) -> Sast.Return(Some(doexp e), env)
        | Sast.Super(args, uid, super, env) -> Sast.Super(doexps args, uid, super, env)
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 =
    let dobody = update_refinements_stmts klass_data kname mname in
    let dopred = update_refinements_expr klass_data kname mname in

    let mapping = function
        | (None, body) -> (None, dobody body)
        | (Some(e), body) -> (Some(dopred e), dobody body) in
    List.map mapping pieces

let update_refinements_func klass_data (func : Sast.func_def) =
    { func with body = update_refinements_stmts klass_data func.inklass func.name func.body }

let update_refinements_member klass_data = function
    | Sast.InitMem(i) -> Sast.InitMem(update_refinements_func klass_data i)
    | Sast.MethodMem(m) -> Sast.MethodMem(update_refinements_func klass_data m)
    | v -> v

let update_refinements_klass klass_data (klass : Sast.class_def) =
    let mems = List.map (update_refinements_member klass_data) in
    let funs = List.map (update_refinements_func klass_data) in
    let s = klass.sections in
    let sects =
        { publics = mems s.publics;
          protects = mems s.protects;
          privates = mems s.privates;
          mains = funs s.mains;
          refines = funs s.refines } in
    { klass with sections = sects }
```

```ocaml

let update_refinements klass_data (klasses : Sast.class_def list) =
    List.map (update_refinements_klass klass_data) klasses

(**
    Given a class_data record, a class name, an environment, and an Ast.expr expression,
    return a Sast.expr expression.
    @param klass_data A class_data record
    @param kname The name of of the current class
    @param env The local environment (instance and local variables so far declared)
    @param exp An expression to eval to a Sast.expr value
    @return A Sast.expr expression, failing when there are issues.
  *)
let rec eval klass_data kname mname isstatic env exp =
    let eval' expr = eval klass_data kname mname isstatic env expr in
    let eval_exprlist elist = List.map eval' elist in

    let get_field expr mbr =
        let (recvr_type, _) as recvr = eval' expr in
        let this = (recvr_type = current_class) in
        let recvr_type = if this then kname else recvr_type in
        let field_type = match Klass.class_field_far_lookup klass_data recvr_type mbr this with
            | Left((_, vtyp, _)) -> vtyp
            | Right(true) -> raise(Failure("Field " ^ mbr ^ " is not accessible in " ^ recvr_type ^ " from " ^ kname ^ "."))
            | Right(false) -> raise(Failure("Unknown field " ^ mbr ^ " in the ancestry of " ^ recvr_type ^ ".")) in
        (field_type, Sast.Field(recvr, mbr)) in

    let cast_to klass (_, v) = (klass, v) in

    let get_invoc expr methd elist =
        let (recvr_type, _) as recvr = eval' expr in
        let arglist = eval_exprlist elist in
        let this = (recvr_type = current_class) in
        let _ = if (this && isstatic)
            then raise(Failure(Format.sprintf "Cannot invoke %s on %s in %s for %s is static." methd mname kname mname))
            else () in
        let recvr_type = if this then kname else recvr_type in
        let argtypes = List.map fst arglist in
        let mfdef = match Klass.best_inherited_method klass_data recvr_type methd argtypes this with
            | None when this -> raise(Failure(Format.sprintf "Method %s not found ancestrally in %s (this=%b)" methd recvr_type this))
            | None -> raise(Failure("Method " ^ methd ^ " not found (publically) in the ancestry of " ^ recvr_type ^ "."))
            | Some(fdef) -> fdef in
        let mfid = if mfdef.builtin then BuiltIn mfdef.uid else FuncId mfdef.uid in
```

```
        (getRetType mfdef.returns, Sast.Invoc(cast_to (mfdef.inklass) recvr, methd, arglist, mfid)) in

    let get_init class_name exprlist =
        let arglist = eval_exprlist exprlist in
        let argtypes = List.map fst arglist in
        let mfdef = match best_method klass_data class_name "init" argtypes [Ast.Publics] with
            | None        -> raise(Failure "Constructor not found")
            | Some(fdef) -> fdef in
        let mfid = if mfdef.builtin then BuiltIn mfdef.uid else FuncId mfdef.uid in
        (class_name, Sast.NewObj(class_name, arglist, mfid)) in

    let get_assign e1 e2 =
        let (t1, t2) = (eval' e1, eval' e2) in
        let (type1, type2) = (fst t1, fst t2) in
        match is_subtype klass_data type2 type1, is_lvalue e1 with
            | _, false -> raise(Failure "Assigning to non-lvalue")
            | false, _ -> raise(Failure "Assigning to incompatible types")
            | _ -> (type1, Sast.Assign(t1, t2)) in

    let get_binop e1 op e2 =
        let isCompatible typ1 typ2 =
            if is_subtype klass_data typ1 typ2 then typ2
            else if is_subtype klass_data typ2 typ1 then typ1
            else raise (Failure (Format.sprintf "Binop takes incompatible types: %s %s" typ1 typ2)) in
        let (t1, t2) = (eval' e1, eval' e2) in
        let gettype op (typ1,_) (typ2,_) = match op with
            | Ast.Arithmetic(Neg) -> raise(Failure("Negation is not a binary operation!"))
            | Ast.CombTest(Not) -> raise(Failure("Boolean negation is not a binary operation!"))
            | Ast.Arithmetic(_) -> isCompatible typ1 typ2
            | Ast.NumTest(_)
            | Ast.CombTest(_) -> ignore(isCompatible typ1 typ2); "Boolean" in
        (gettype op t1 t2, Sast.Binop(t1,op,t2)) in

    let get_refine rname elist desired =
        let arglist = eval_exprlist elist in
        let argtypes = List.map fst arglist in
        let refines = Klass.refine_on klass_data kname mname rname argtypes desired in
        let switch = List.map (fun (f : Ast.func_def) -> (f.inklass, f.uid)) refines in
        (getRetType desired, Sast.Refine(rname, arglist, desired, Switch(kname, switch, UID.uid_counter ()))) in

    let get_refinable rname =
        let refines = Klass.refinable_lookup klass_data kname mname rname in
```

```
        let klasses = List.map (fun (f : Ast.func_def) -> f.inklass) refines in
        ("Boolean", Sast.Refinable(rname, Test(kname, klasses, UID.uid_counter ()))) in

    let get_deref e1 e2 =
        let expectArray typename = match Str.last_chars typename 2 with
            | "[]" -> Str.first_chars typename (String.length typename - 2)
            | _  -> raise (Failure "Not an array type") in
        let (t1, t2) = (eval' e1, eval' e2) in
        let getArrayType (typ1, _) (typ2, _) = match typ2 with
            | "Integer" -> expectArray typ1
            | _ -> raise(Failure "Dereferencing invalid") in
        (getArrayType t1 t2, Sast.Deref(t1, t2)) in
    let get_unop op expr = match op with
        | Ast.Arithmetic(Neg) -> let (typ, _) as evaled = eval' expr in (typ, Sast.Unop(op, evaled))
        | Ast.CombTest(Not) -> ("Boolean", Sast.Unop(op, eval' expr))
        | _ -> raise(Failure("Unknown binary operator " ^ Inspector.inspect_ast_op op ^ " given.")) in

    let lookup_type id = match map_lookup id env with
        | None -> raise(Failure("Unknown id " ^ id ^ " in environment built around " ^ kname ^ ", " ^ mname ^ "."))
        | Some((vtype, _)) -> vtype in

    let get_new_arr atype args =
        let arglist = eval_exprlist args in
        if List.exists (fun (t, _) -> t <> "Integer") arglist
           then raise(Failure "Size of an array dimensions does not correspond to an integer.")
           else (atype, Sast.NewObj(atype, arglist, ArrayAlloc(UID.uid_counter ()))) in

    let get_new_obj atype args = try
        let index = String.index atype '[' in
        let dimensions = (String.length atype - index) / 2 in
        match List.length args with
            | n when n > dimensions -> raise(Failure("Cannot allocate array, too many dimensions given."))
            | n when n < dimensions -> raise(Failure("Cannot allocate array, too few dimensions given."))
            | 0 -> (null_class, Sast.Null)
            | _ -> get_new_arr atype args
        with Not_found -> get_init atype args in

    match exp with
        | Ast.This -> (current_class, Sast.This)
        | Ast.Null -> (null_class, Sast.Null)
        | Ast.Id(vname) -> (lookup_type vname, Sast.Id(vname))
        | Ast.Literal(lit) -> (getLiteralType lit, Sast.Literal(lit))
        | Ast.NewObj(s1, elist) -> get_new_obj s1 elist
        | Ast.Field(expr, mbr) -> get_field expr mbr
```

```
        | Ast.Invoc(expr, methd, elist) -> get_invoc expr methd elist
        | Ast.Assign(e1, e2) -> get_assign e1 e2
        | Ast.Binop(e1,op,e2) -> get_binop e1 op e2
        | Ast.Refine(s1, elist, soption) -> get_refine s1 elist soption
        | Ast.Deref(e1, e2) -> get_deref e1 e2
        | Ast.Refinable(s1) -> get_refinable s1
        | Ast.Unop(op, expr) -> get_unop op expr
        | Ast.Anonymous(atype, args, body) -> (atype, Sast.Anonymous(atype, eval_exprlist args, body)) (* Delay evaluation *)

(**
    Given a class_data record, the name of the current class, a list of AST statements,
    and an initial environment, enumerate the statements and attach the environment at
    each step to that statement, yielding Sast statements. Note that when there is an
    issue the function will raise Failure.
    @param klass_data A class_data record
    @param kname The name of the class that is the current context.
    @param stmts A list of Ast statements
    @param initial_env An initial environment
    @return A list of Sast statements
  *)
let rec attach_bindings klass_data kname mname meth_ret isstatic stmts initial_env =
    (* Calls that go easy on the eyes *)
    let eval' = eval klass_data kname mname isstatic in
    let attach' = attach_bindings klass_data kname mname meth_ret isstatic in
    let eval_exprlist env elist = List.map (eval' env) elist in

    let rec get_superinit kname arglist =
        let parent = StringMap.find kname klass_data.parents in
        let argtypes = List.map fst arglist in
        match best_method klass_data parent "init" argtypes [Ast.Publics; Ast.Protects] with
            | None       -> raise(Failure "Cannot find super init")
            | Some(fdef) -> fdef in

    (* Helper function for building a predicate expression *)
    let build_predicate pred_env exp = match eval' pred_env exp with
        | ("Boolean", _) as evaled -> evaled
        | _ -> raise (Failure "Predicates must be boolean") in

    (* Helper function for building an optional expression *)
    let opt_eval opt_expr opt_env = match opt_expr with
        | None -> None
        | Some(exp) -> Some(eval' opt_env exp) in

    (* For each kind of statement, build the associated Sast
```

```
    statment *)
    let build_ifstmt iflist if_env =
        let build_block if_env (exp, slist) =
            let exprtyp = match exp with
                | None -> None
                | Some exp -> Some(build_predicate if_env exp) in
            (exprtyp, attach' slist if_env) in
        Sast.If(List.map (build_block if_env) iflist, if_env) in

    let build_whilestmt expr slist while_env =
        let exprtyp = build_predicate while_env expr in
        let stmts = attach' slist while_env in
        Sast.While(exprtyp, stmts, while_env) in

    let build_declstmt ((vtype, vname) as vdef) opt_expr decl_env =
        if not (Klass.is_type klass_data vtype) then raise(Failure(Format.sprintf "%s in %s.%s has unknown type %s." vname kname mname vtype))
        else match opt_eval opt_expr decl_env with
            | Some((atype, _)) as evaled -> if not (Klass.is_subtype klass_data atype vtype)
                then raise(Failure(Format.sprintf "%s in %s.%s is type %s but is assigned a value of type %s." vname kname mname vtype atype))
                else Sast.Decl(vdef, evaled, decl_env)
            | None -> Sast.Decl(vdef, None, decl_env) in

    let check_ret_type ret_type = match ret_type, meth_ret with
        | None, Some(_) -> raise(Failure("Void return from non-void function " ^ mname ^ " in klass " ^ kname ^ "."))
        | Some(_), None -> raise(Failure("Non-void return from void function " ^ mname ^ " in klass " ^ kname ^ "."))
        | 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))
        | _, _ -> () in

    let build_returnstmt opt_expr ret_env =
        let ret_val = opt_eval opt_expr ret_env in
        let ret_type = match ret_val with Some(t, _) -> Some(t) | _ -> None in
        check_ret_type ret_type;
        Sast.Return(ret_val, ret_env) in
    let build_exprstmt expr expr_env = Sast.Expr(eval' expr_env expr, expr_env) in
    let build_superstmt expr_list super_env =
        let arglist = eval_exprlist super_env expr_list in
        let init = get_superinit kname arglist in
        match map_lookup kname klass_data.parents with
            | None -> raise(Failure("Error -- getting parent for object without parent: " ^ kname))
            | Some(parent) -> Sast.Super(arglist, init.uid, parent, super_env) in

```

```
    (* Ast statement -> (Sast.Statement, Environment Update Option) *)
    let updater in_env = function
        | Ast.While(expr, slist)   -> (build_whilestmt expr slist in_env, None)
        | Ast.If(iflist)           -> (build_ifstmt iflist in_env, None)
        | Ast.Decl(vdef, opt_expr) -> (build_declstmt vdef opt_expr in_env, Some(vdef))
        | Ast.Expr(expr)           -> (build_exprstmt expr in_env, None)
        | Ast.Return(opt_expr)     -> (build_returnstmt opt_expr in_env, None)
        | Ast.Super(exprs)         -> (build_superstmt exprs in_env, None) in

    (* Function to fold a statement into a growing reverse list of Sast statements *)
    let build_env (output, acc_env) stmt =
        let (node, update) = updater acc_env stmt in
        let updated_env = match update with
            | None -> acc_env
            | Some(vdef) -> env_update Local vdef acc_env in
        (node::output, updated_env) in

    List.rev (fst(List.fold_left build_env ([], initial_env) stmts))

(**
    Given a list of statements, return whether every execution path therein returns
    @param stmts A bunch of Ast.stmts
    @return true or false based on whether everything returns a value.
  *)
let rec does_return_stmts (stmts : Ast.stmt list) = match stmts with
    | [] -> false
    | Return(None)::_ -> false
    | Return(_)::_ -> true
    | If(pieces)::rest -> does_return_clauses pieces || does_return_stmts rest
    | _::rest -> does_return_stmts rest
(**
    Given a collection of if clauses, return whether they represent a return from the function.
    @param pieces If clauses (option expr, stmt list)
    @return whether or not it can be determined that a return is guaranteed here.
  *)
and does_return_clauses pieces =
    let (preds, bodies) = List.split pieces in
    List.mem None preds && List.for_all does_return_stmts bodies

(**
    Change inits so that they return this
  *)
```

```
let init_returns (func : Sast.func_def) =
    let body = if func.builtin then [] else func.body @ [Sast.
    Return(None, empty_environment)] in
    let this_val = (current_class, Sast.This) in
    let return_this (stmt : Sast.sstmt) = match stmt with
        | Return(None, env) -> Return(Some(this_val), env)
        | _ -> stmt in
    { func with
        returns = Some(func.inklass);
        body = List.map return_this body }

let rec update_current_ref_stmts (kname : string) (stmts : Sast.
    sstmt list) : Sast.sstmt list = List.map (
    update_current_ref_stmt kname) stmts
and update_current_ref_exprs (kname : string) (exprs : Sast.expr
     list) = List.map (update_current_ref_expr kname) exprs
and update_current_ref_stmt (kname : string) (stmt : Sast.sstmt)
     = match stmt with
    | Sast.Decl(vdef, None, env) -> Sast.Decl(vdef, None, env)
    | Sast.Decl(vdef, Some(expr), env) -> Sast.Decl(vdef, Some(
    update_current_ref_expr kname expr), env)
    | Sast.Expr(expr, env) -> Sast.Expr(update_current_ref_expr
    kname expr, env)
    | Sast.If(pieces, env) -> Sast.If(update_current_ref_clauses
     kname pieces, env)
    | Sast.While(expr, body, env) -> Sast.While(
    update_current_ref_expr kname expr, update_current_ref_stmts
     kname body, env)
    | Sast.Return(None, env) -> Sast.Return(None, env)
    | Sast.Return(Some(expr), env) -> Sast.Return(Some(
    update_current_ref_expr kname expr), env)
    | Sast.Super(args, uid, parent, env) -> Sast.Super(
    update_current_ref_exprs kname args, uid, parent, env)
and update_current_ref_expr (kname : string) ((atype, detail) :
    string * Sast.expr_detail) : string * Sast.expr_detail =
    let cleaned = match detail with
        | Sast.This -> Sast.This
        | Sast.Null -> Sast.Null
        | Sast.Id(i) -> Sast.Id(i)
        | Sast.NewObj(klass, args, uid) -> Sast.NewObj(klass,
    update_current_ref_exprs kname args, uid)
        | Sast.Anonymous(klass, args, refs) -> Sast.Anonymous(
    klass, args, refs)
        | Sast.Literal(lit) -> Sast.Literal(lit)
        | Sast.Assign(mem, data) -> Sast.Assign(
    update_current_ref_expr kname mem, update_current_ref_expr
    kname data)
        | Sast.Deref(arr, idx) -> Sast.Deref(
    update_current_ref_expr kname arr, update_current_ref_expr
    kname idx)
        | Sast.Field(expr, member) -> Sast.Field(
    update_current_ref_expr kname expr, member)
        | Sast.Invoc(expr, meth, args, id) -> Sast.Invoc(
    update_current_ref_expr kname expr, meth,
    update_current_ref_exprs kname args, id)
        | Sast.Unop(op, expr) -> Sast.Unop(op,
    update_current_ref_expr kname expr)
```

```
        | Sast.Binop(l, op, r) -> Sast.Binop(update_current_ref_expr kname l, op, update_current_ref_expr kname r)
        | Sast.Refine(refine, args, ret, switch) -> Sast.Refine(refine, update_current_ref_exprs kname args, ret, switch)
       | Sast.Refinable(refine, switch) -> Sast.Refinable(refine, switch) in
    let realtype : string = if current_class = atype then kname else atype in
    (realtype, cleaned)
and update_current_ref_clauses (kname : string) pieces =
    let (preds, bodies) = List.split pieces in
    let preds = List.map (function None -> None | Some(expr) -> Some(update_current_ref_expr kname expr)) preds in
    let bodies = List.map (update_current_ref_stmts kname) bodies in
    List.map2 (fun a b -> (a, b)) preds bodies

(**
    Given a class_data record, an Ast.func_def, an an initial environment,
    convert the func_def to a Sast.func_def. Can raise failure when there
    are issues with the statements / expressions in the function.
    @param klass_data A class_data record
    @param func An Ast.func_def to transform
    @param initial_env The initial environment
    @return A Sast.func_def value
  *)
let ast_func_to_sast_func klass_data (func : Ast.func_def) initial_env isinit =
    let with_params = List.fold_left (fun env vdef -> env_update Local vdef env) initial_env func.formals in
    let checked : Sast.sstmt list = attach_bindings klass_data func.inklass func.name func.returns func.static func.body with_params in
    let cleaned = update_current_ref_stmts func.inklass checked in
    let sast_func : Sast.func_def =
        {   returns = func.returns;
            host = func.host;
            name = func.name;
            formals = func.formals;
            static = func.static;
            body = cleaned;
            section = func.section;
            inklass = func.inklass;
            uid = func.uid;
            builtin = func.builtin } in
    let isvoid = match func.returns with None -> true | _ -> false in
    if not func.builtin && not isvoid && not (does_return_stmts func.body)
        then raise(Failure(Format.sprintf "The function %s in %s does not return on all execution paths" (full_signature_string func) func.inklass))
```

```
        else if isinit then init_returns sast_func else sast_func

(**
    Given a class_data record, an Ast.member_def, and an initial environment,
    convert the member into an Sast.member_def. May raise failure when there
    are issues in the statements / expressions in the member.
    @param klass_data A class_data record.
    @param mem An Ast.member_def value
    @param initial_env An environment of variables
    @return A Sast.member_def
  *)
let ast_mem_to_sast_mem klass_data (mem : Ast.member_def) initial_env =
    let change isinit func = ast_func_to_sast_func klass_data func initial_env isinit in
    let transformed : Sast.member_def = match mem with
        | Ast.VarMem(v) -> Sast.VarMem(v)
        | Ast.MethodMem(m) -> Sast.MethodMem(change false m)
        | Ast.InitMem(m) -> Sast.InitMem(change true m) in
    transformed

let init_calls_super (aklass : Sast.class_def) =
    let validate_init func_def = match func_def.builtin, func_def.body with
        | true, _ -> true
        | _, (Super(_,_,_,_)::_) -> true
        | _, _ -> false in
    let grab_init = function
        | InitMem(m) -> Some(m)
        | _ -> None in
    let get_inits mems = Util.filter_option (List.map grab_init mems) in
    let s = aklass.sections in
    let inits = List.flatten (List.map get_inits [s.publics; s.protects; s.privates]) in
    List.for_all validate_init inits

let check_main (func : Ast.func_def) = match func.formals with
    | [("System", _); ("String[]", _)] -> func
    | _ -> 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))

(**
    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.
    @param klass_data A class_data record value
    @param ast_klass A class to transform
    @return The transformed class.
  *)
let ast_to_sast_klass klass_data (ast_klass : Ast.class_def) =
    let s : Ast.class_sections_def = ast_klass.sections in
```

```
    let rec update_env env sect (klass : Ast.class_def) =
        let env = add_ivars klass env sect in
        match klass.klass with
            | "Object" -> env
            | _ -> let parent = Klass.klass_to_parent klass in
                   let pclass = StringMap.find parent klass_data.classes in
                   update_env env Protects pclass in
    let env = update_env empty_environment Privates ast_klass in

    let mems = List.map (fun m -> ast_mem_to_sast_mem klass_data m env) in
    let funs = List.map (fun f -> ast_func_to_sast_func klass_data f env false) in

    let sections : Sast.class_sections_def =
        {   publics = mems s.publics;
            protects = mems s.protects;
            privates = mems s.privates;
            refines = funs s.refines;
            mains = funs (List.map check_main s.mains) } in

    let sast_klass : Sast.class_def =
        {   klass = ast_klass.klass;
            parent = ast_klass.parent;
            sections = sections } in

    if init_calls_super sast_klass then sast_klass
    else raise(Failure(Format.sprintf "%s's inits don't always call super as their first statement (maybe empty body, maybe something else)." sast_klass.klass))

(**
    @param ast An ast program
    @return A sast program
*)
let ast_to_sast klass_data =
    let klasses = StringMap.bindings klass_data.classes in
    let to_sast (_, klass) = ast_to_sast_klass klass_data klass in
    List.map to_sast klasses
```

Source 57: `BuildSast.ml`

```
(**
    The abstract syntax tree for Gamma
*)

(**
    The four literal classes of Gamma:
    - Int - Integer
    - Float - Floating-point number
    - String - A sequence of characters
    - Bool - a boolean value of either true or false
*)
```

```
type lit =
    Int of int
  | Float of float
  | String of string
  | Bool of bool

(** The binary arithmatic operators *)
type arith = Add | Sub | Prod | Div | Mod | Neg | Pow

(** The binary comparison operators *)
type numtest = Eq | Neq | Less | Grtr | Leq | Geq

(** The binary boolean operators *)
type combtest = And | Or | Nand | Nor | Xor | Not

(** All three sets of binary operators *)
type op = Arithmetic of arith | NumTest of numtest | CombTest of combtest

(** The various types of expressions we can have. *)
type expr =
    This
  | Null
  | Id of string
  | NewObj of string * expr list
  | Anonymous of string * expr list * func_def list
  | Literal of lit
  | Assign of expr * expr  (* memory := data -- whether memory is good is a semantic issue *)
  | Deref of expr * expr (* road[pavement] *)
  | Field of expr * string (* road.pavement *)
  | Invoc of expr * string * expr list (* receiver.method(args) *)
  | Unop of op * expr (* !x *)
  | Binop of expr * op * expr (* x + y *)
  | Refine of string * expr list * string option
  | Refinable of string (* refinable *)
(** The basic variable definition, a type and an id*)
and var_def = string * string  (* Oh typing, you pain in the ass, add a int for array *)
(** The basic statements: Variable declarations, control statements, assignments, return statements, and super class expressions *)
and stmt =
    Decl of var_def * expr option
  | If of (expr option * stmt list) list
  | While of expr * stmt list
  | Expr of expr
  | Return of expr option
  | Super of expr list

(** Three access levels, the refinements, and the main function *)
and class_section = Publics | Protects | Privates | Refines | Mains

(** We have four different kinds of callable code blocks: main,
```

```
    init, refine, method. *)
and func_def = {
  returns : string option;  (** A return type (method/refine) *)
  host    : string option;  (** A host class (refine) *)
  name    : string;         (** The function name (all) *)
  static  : bool;           (** If the function is static (main) *)
  formals : var_def list;   (** A list of all formal parameters of the function (all) *)
  body    : stmt list;      (** A list of statements that form the function body (all) *)
  section : class_section;  (** A sementic tag of the class section in which the function lives (all) *)
  inklass : string;         (** A semantic tag of the class in which the function lives (all) *)
  uid     : string;         (** A string for referencing this -- should be maintained in transformations to later ASTs *)
  builtin : bool;           (** Whether or not the function is built in (uid should have _ in it then) *)
}

(** A member is either a variable or some sort of function *)
type member_def = VarMem of var_def | MethodMem of func_def | InitMem of func_def

(** Things that can go in a class *)
type class_sections_def = {
  privates : member_def list;
  protects : member_def list;
  publics  : member_def list;
  refines  : func_def list;
  mains    : func_def list;
}

(* Just pop init and main in there? *)
(** The basic class definition *)
type class_def = {
  klass    : string; (** A name string *)
  parent   : string option; (** The parent class name *)
  sections : class_sections_def; (** The five sections *)
}

(** A program, right and proper  *)
type program = class_def list
```

Source 58: `Ast.mli`

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

Source 59: `prettify.ml`

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

Source 60: `Unanonymous.mli`

```

/* GLOBAL DATA */
struct t_System global_system;
int object_counter;
int global_argc;

/* Prototypes */
struct t_Object *allocate_for(size_t, ClassInfo *);
void *array_allocator(size_t, int);
struct t_Integer *integer_value(int);
struct t_Float *float_value(double);
struct t_Boolean *bool_value(unsigned char);
struct t_String *string_value(char *);
struct t_Boolean *boolean_init(struct t_Boolean *);
struct t_Integer *integer_init(struct t_Integer *);
struct t_Float *float_init(struct t_Float *);
struct t_Object *object_init(struct t_Object *);
struct t_String *string_init(struct t_String *);
struct t_Printer *printer_init(struct t_Printer *, struct
    t_Boolean *);
struct t_Scanner *scanner_init(struct t_Scanner *);
struct t_Integer *float_to_i(struct t_Float *);
struct t_Float *integer_to_f(struct t_Integer *);
struct t_Float *scanner_scan_float(struct t_Scanner *);
struct t_Integer *scanner_scan_integer(struct t_Scanner *);
struct t_String *scanner_scan_string(struct t_Scanner *);
void printer_print_float(struct t_Printer *, struct t_Float *);
void printer_print_integer(struct t_Printer *, struct t_Integer
    *);
void printer_print_string(struct t_Printer *, struct t_String *)
    ;
struct t_String **get_gamma_args(char **argv, int argc);


char *stack_overflow_getline(FILE *);

/* Functions! */

/* Magic allocator. DO NOT INVOKE THIS, USE MAKE_NEW(TYPE)
 * where type is not prefixed (i.e. MAKE_NEW(Integer) not
 * MAKE_NEW(t_Integer))
 */
struct t_Object *allocate_for(size_t s, ClassInfo *meta) {
```

```
    struct t_Object *this = (struct t_Object *)(malloc(s));
    if (!this) {
        fprintf(stderr, "Could not even allocate memory. Exiting.\n");
        exit(1);
    }
    this->meta = meta;
    return this;
}

void *array_allocator(size_t size, int n) {
    void *mem = malloc(size * n);
    if (!mem) {
        fprintf(stderr, "Failure allocating for array.  Exiting.\n");
        exit(1);
    }
    memset(mem, 0, size * n);
    return mem;
}

/* Make basic objects with the given values. */
struct t_Integer *integer_value(int in_i) {
    struct t_Integer *i = MAKENEW(Integer);
    i = integer_init(i);
    i->Integer.value = in_i;
    return i;
}

struct t_Float *float_value(double in_f) {
    struct t_Float *f = MAKENEW(Float);
    f = float_init(f);
    f->Float.value = in_f;
    return f;
}

struct t_Boolean *bool_value(unsigned char in_b) {
    struct t_Boolean *b = MAKENEW(Boolean);
    b = boolean_init(b);
    b->Boolean.value = in_b;
    return b;
}

struct t_String *string_value(char *s_in) {
    size_t length = 0;
    char *dup = NULL;
    length = strlen(s_in) + 1;

    struct t_String *s = MAKENEW(String);
    s = string_init(s);
    dup = malloc(sizeof(char) * length);
    if (!dup) {
        fprintf(stderr, "Out of memory in string_value.\n");
        exit(1);
    }
    s->String.value = strcpy(dup, s_in);
    return s;
```

```
}

struct t_Boolean *boolean_init(struct t_Boolean *this){
    object_init((struct t_Object *)(this));
    this->Boolean.value = 0;
    return this;
}

struct t_Integer *integer_init(struct t_Integer *this){
    object_init((struct t_Object *)(this));
    this->Integer.value = 0;
    return this;
}

struct t_Float *float_init(struct t_Float *this){
    object_init((struct t_Object *)(this));
    this->Float.value = 0.0;
    return this;
}

struct t_Object *object_init(struct t_Object *this){
    this->Object.v_system = &global_system;
    return this;
}

struct t_String *string_init(struct t_String *this)
{
    object_init((struct t_Object *)(this));
    this->String.value = NULL;
    return this;
}

struct t_System *system_init(struct t_System *this)
{
    this->System.v_err = MAKENEW(Printer);
    this->System.v_in = MAKENEW(Scanner);
    this->System.v_out = MAKENEW(Printer);
    this->System.v_argc = MAKENEW(Integer);

    this->System.v_err->Printer.target = stderr;
    this->System.v_in->Scanner.source = stdin;
    this->System.v_out->Printer.target = stdout;
    this->System.v_argc->Integer.value = global_argc;
    this->Object.v_system =
        this->System.v_err->Object.v_system =
        this->System.v_in->Object.v_system =
        this->System.v_out->Object.v_system =
        this->System.v_argc->Object.v_system = this;
    return this;
};

struct t_Printer *printer_init(struct t_Printer *this, struct
    t_Boolean *v_stdout)
{
    object_init((struct t_Object *)(this));
    this->Printer.target = v_stdout->Boolean.value ? stdout :
    stderr;
```

```
    return this;
}

struct t_Scanner *scanner_init(struct t_Scanner *this)
{
    object_init((struct t_Object *)(this));
    this->Scanner.source = stdin;
}

struct t_Integer *float_to_i(struct t_Float *this){
    return integer_value((int)(this->Float.value));
}

struct t_Float *integer_to_f(struct t_Integer *this){
    return float_value((double)(this->Integer.value));
}

void toendl(FILE *in) {
    int c = 0;
    while (1) {
      c = fgetc(in);
      if (c == '\n' || c == '\r' || c == EOF) break;
    }
}

struct t_Float *scanner_scan_float(struct t_Scanner *this)
{
    double dval;
    fscanf(this->Scanner.source, "%lf", &dval);
    toendl(this->Scanner.source);

    return float_value(dval);
}

struct t_Integer *scanner_scan_integer(struct t_Scanner *this)
{
    int ival;
    fscanf(this->Scanner.source, "%d", &ival);
    toendl(this->Scanner.source);
    return integer_value(ival);
}

struct t_String *scanner_scan_string(struct t_Scanner *this)
{
    char *inpstr = NULL;
    struct t_String *astring = NULL;

    inpstr = stack_overflow_getline(this->Scanner.source);
    astring = string_value(inpstr);

    free(inpstr);
    return astring;
}

void printer_print_float(struct t_Printer *this, struct t_Float
    *v_arg)
{
```

```c
    fprintf(this->Printer.target, "%lf", v_arg->Float.value);
}

void printer_print_integer(struct t_Printer *this, struct t_Integer *v_arg)
{
    fprintf(this->Printer.target, "%d", v_arg->Integer.value);
}

void printer_print_string(struct t_Printer *this, struct t_String *v_arg)
{
    fprintf(this->Printer.target, "%s", v_arg->String.value);
}

void system_exit(struct t_System *this, struct t_Integer *v_code) {
    exit(INTEGER_OF(v_code));
}


struct t_String **get_gamma_args(char **argv, int argc) {
    struct t_String **args = NULL;
    int i = 0;

    if (!argc) return NULL;
    args = ONE_DIM_ALLOC(struct t_String *, argc);
    for (i = 0; i < argc; ++i)
        args[i] = string_value(argv[i]);
    args[i] = NULL;

    return args;
}



char *stack_overflow_getline(FILE *in) {
    char * line = malloc(100), * linep = line;
    size_t lenmax = 100, len = lenmax;
    int c;

    if(line == NULL)
        return NULL;

    for(;;) {
        c = fgetc(in);
        if(c == EOF)
            break;

        if(--len == 0) {
            len = lenmax;
            char * linen = realloc(linep, lenmax *= 2);

            if(linen == NULL) {
                free(linep);
                return NULL;
            }
```

```c
                line = linen + (line - linep);
                linep = linen;
            }

            if((*line++ = c) == '\n')
                break;
        }
        *line = '\0';
        return linep;
}
```

Source 61: `headers/gamma-builtin-functions.h`

```c
#include <stdarg.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {
        int generation;
        char* class;
        char** ancestors;
} ClassInfo;


ClassInfo M_Boolean;
ClassInfo M_Float;
ClassInfo M_Integer;
ClassInfo M_Object;
ClassInfo M_Printer;
ClassInfo M_Scanner;
ClassInfo M_String;
ClassInfo M_System;


/*
        Initializes the given ClassInfo
*/
void class_info_init(ClassInfo* meta, int num_args, ...) {

        int i;
        va_list objtypes;
        va_start(objtypes, num_args);

        meta->ancestors = malloc(sizeof(char *) * num_args);

        if (meta->ancestors == NULL) {
                printf("\nMemory error - class_info_init failed\n");
                exit(0);
        }
        for(i = 0; i < num_args; i++) {
                meta->ancestors[i] = va_arg(objtypes, char * );
        }
        meta->generation = num_args - 1;
        meta->class = meta->ancestors[meta->generation];
```

```
        va_end(objtypes);
}


void init_built_in_infos() {
    class_info_init(&M_Boolean, 2, m_classes[T_OBJECT],
    m_classes[T_BOOLEAN]);
    class_info_init(&M_Float, 2, m_classes[T_OBJECT], m_classes[
    T_FLOAT]);
    class_info_init(&M_Integer, 2, m_classes[T_OBJECT],m_classes
    [T_INTEGER]);
    class_info_init(&M_Object, 1, m_classes[T_OBJECT]);
    class_info_init(&M_Printer, 2, m_classes[T_OBJECT],m_classes
    [T_PRINTER]);
    class_info_init(&M_Scanner, 2, m_classes[T_OBJECT],m_classes
    [T_SCANNER]);
    class_info_init(&M_String, 2, m_classes[T_OBJECT],m_classes[
    T_STRING]);
    class_info_init(&M_System, 2, m_classes[T_OBJECT],m_classes[
    T_SYSTEM]);
}
```

Source 62: `headers/gamma-builtin-meta.h`

```


/*
 * Structures for each of the objects.
 */
struct t_Boolean;
struct t_Float;
struct t_Integer;
struct t_Object;
struct t_Printer;
struct t_Scanner;
struct t_String;
struct t_System;


struct t_Boolean {
    ClassInfo *meta;

    struct {
        struct t_System *v_system;
    } Object;


    struct { unsigned char value; } Boolean;
};


struct t_Float {
    ClassInfo *meta;

    struct {
```

```
        struct t_System *v_system;
    } Object;


    struct { double value; } Float;
};


struct t_Integer {
    ClassInfo *meta;

    struct {
        struct t_System *v_system;
    } Object;


    struct { int value; } Integer;
};


struct t_Object {
    ClassInfo *meta;

    struct {
        struct t_System *v_system;
    } Object;
};


struct t_Printer {
    ClassInfo *meta;

    struct {
        struct t_System *v_system;
    } Object;


    struct { FILE *target; } Printer;
};


struct t_Scanner {
    ClassInfo *meta;

    struct {
        struct t_System *v_system;
    } Object;


    struct { FILE *source; } Scanner;
};


struct t_String {
    ClassInfo *meta;

    struct {
```

```
        struct t_System *v_system;
    } Object;


    struct { char *value; } String;
};


struct t_System {
    ClassInfo *meta;

    struct {
        struct t_System *v_system;
    } Object;


    struct {
        struct t_Printer *v_err;
        struct t_Scanner *v_in;
        struct t_Printer *v_out;
        struct t_Integer *v_argc;
    } System;
};
```

Source 63: `headers/gamma-builtin-struct.h`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define BYTE unsigned char

#define PROMOTE_INTEGER(ival)   integer_value((ival))
#define PROMOTE_FLOAT(fval)     float_value((fval))
#define PROMOTE_STRING(sval)    string_value((sval))
#define PROMOTE_BOOL(bval)      bool_value((bval))

#define LIT_INT(lit_int)        PROMOTE_INTEGER(lit_int)
#define LIT_FLOAT(lit_flt)      PROMOTE_FLOAT(lit_flt)
#define LIT_STRING(lit_str)     PROMOTE_STRING(lit_str)
#define LIT_BOOL(lit_bool)      PROMOTE_BOOL(lit_bool)

#define ADD_INT_INT(l, r)       PROMOTE_INTEGER(INTEGER_OF(l) + INTEGER_OF(r))
#define ADD_FLOAT_FLOAT(l, r)   PROMOTE_FLOAT(FLOAT_OF(l) + FLOAT_OF(r))
#define SUB_INT_INT(l, r)       PROMOTE_INTEGER(INTEGER_OF(l) - INTEGER_OF(r))
#define SUB_FLOAT_FLOAT(l, r)   PROMOTE_FLOAT(FLOAT_OF(l) - FLOAT_OF(r))
#define PROD_INT_INT(l, r)      PROMOTE_INTEGER(INTEGER_OF(l) * INTEGER_OF(r))
#define PROD_FLOAT_FLOAT(l, r)  PROMOTE_FLOAT(FLOAT_OF(l) * FLOAT_OF(r))
```

```c
#define DIV_INT_INT(l, r)          PROMOTE_INTEGER(INTEGER_OF(l) / INTEGER_OF(r))
#define DIV_FLOAT_FLOAT(l, r)      PROMOTE_FLOAT(FLOAT_OF(l) / FLOAT_OF(r))
#define MOD_INT_INT(l, r)          PROMOTE_INTEGER(INTEGER_OF(l) % INTEGER_OF(r))
#define POW_INT_INT(l, r)          PROMOTE_INTEGER(( (int)pow(INTEGER_OF(l), INTEGER_OF(r)) ))
#define POW_FLOAT_FLOAT(l, r)      PROMOTE_FLOAT( pow(FLOAT_OF(l), FLOAT_OF(r)) )

#define MAKE_NEW2(type, meta) ((struct type *)(allocate_for(sizeof(struct type), &meta)))
#define MAKE_NEW(t_name) MAKE_NEW2(t_##t_name, M_##t_name)

#define CAST(type, v) ( (struct t_##type *)(v) )
#define VAL_OF(type, v) ( CAST(type, v)->type.value )
#define BOOL_OF(b)     VAL_OF(Boolean, b)
#define FLOAT_OF(f)    VAL_OF(Float, f)
#define INTEGER_OF(i) VAL_OF(Integer, i)
#define STRING_OF(s)   VAL_OF(String, s)

#define NEG_INTEGER(i)              PROMOTE_INTEGER(-INTEGER_OF(i))
#define NEG_FLOAT(f)                PROMOTE_FLOAT(-FLOAT_OF(f))
#define NOT_BOOLEAN(b)              PROMOTE_BOOL(!BOOL_OF(b))

#define BINOP(type, op, l, r)       ( VAL_OF(type, l) op VAL_OF(type, r) )
#define PBINOP(type, op, l, r)      PROMOTE_BOOL(BINOP(type, op, l, r))
#define IBINOP(op, l, r)            PBINOP(Integer, op, l, r)
#define FBINOP(op, l, r)            PBINOP(Float, op, l, r)
#define BBINOP(op, l, r)            PBINOP(Boolean, op, l, r)

#define NTEST_EQ_INT_INT(l, r)     IBINOP(==, l, r)
#define NTEST_NEQ_INT_INT(l, r)    IBINOP(!=, l, r)
#define NTEST_LESS_INT_INT(l, r)   IBINOP(<, l, r)
#define NTEST_GRTR_INT_INT(l, r)   IBINOP(>, l, r)
#define NTEST_LEQ_INT_INT(l, r)    IBINOP(<=, l, r)
#define NTEST_GEQ_INT_INT(l, r)    IBINOP(>=, l, r)

#define NTEST_EQ_FLOAT_FLOAT(l, r)     FBINOP(==, l, r)
#define NTEST_NEQ_FLOAT_FLOAT(l, r)    FBINOP(!=, l, r)
#define NTEST_LESS_FLOAT_FLOAT(l, r)   FBINOP(<, l, r)
#define NTEST_GRTR_FLOAT_FLOAT(l, r)   FBINOP(>, l, r)
#define NTEST_LEQ_FLOAT_FLOAT(l, r)    FBINOP(<=, l, r)
#define NTEST_GEQ_FLOAT_FLOAT(l, r)    FBINOP(>=, l, r)

#define CTEST_AND_BOOL_BOOL(l, r)    BBINOP(&&, l, r)
#define CTEST_OR_BOOL_BOOL(l, r)     BBINOP(||, l, r)
#define CTEST_NAND_BOOL_BOOL(l, r)  PROMOTE_BOOL(( !(BOOL_OF(l) && BOOL_OF(r)) ))
#define CTEST_NOR_BOOL_BOOL(l, r)   PROMOTE_BOOL(( !(BOOL_OF(l) || BOOL_OF(r)) ))
#define CTEST_XOR_BOOL_BOOL(l, r)   PROMOTE_BOOL((!BOOL_OF(l) != !BOOL_OF(r)))
```

```

#define IS_CLASS(obj, kname) ( strcmp((obj)->meta->ancestors[obj->meta->generation], (kname)) == 0 )

#define ONE_DIM_ALLOC(type, len) ((type *) array_allocator(sizeof(type), (len)))

#define INIT_MAIN(options) \
struct t_String **str_args = NULL; \
char *gmain = NULL; \
--argc; ++argv; \
if (!argc) { \
    fprintf(stderr, "Please select a main to use.  Available options: " options "\n"); \
    exit(1); \
} \
gmain = *argv; ++argv; --argc; \
init_class_infos(); \
global_argc = argc; \
system_init(&global_system); \
str_args = get_gamma_args(argv, argc);


#define FAIL_MAIN(options) \
fprintf(stderr, "None of the available options were selected. Options were: " options "\n"); \
exit(1);

#define REFINE_FAIL(parent) \
    fprintf(stderr, "Refinement fail: " parent "\n"); \
    exit(1);
```

Source 64: `headers/gamma-preamble.h`

```

(** Types for the semantic abstract syntax tree *)

(** A switch for refinement or refinable checks *)
type refine_switch =
    | Switch of string * (string * string) list * string (* host class, class/best-uid list, switch uid *)
    | Test of string * string list * string  (* host class, class list, uid of switch *)

(** The type of a variable in the environment *)
type varkind = Instance of string | Local

(** The environment at any given statement. *)
type environment = (string * varkind) Map.Make(String).t

(** The ID can be built in (and so won't get mangled) or an array allocator. *)
type funcid = BuiltIn of string | FuncId of string | ArrayAlloc of string

```

```
(** An expression value -- like in AST *)
type expr_detail =
    | This
    | Null
    | Id of string
    | NewObj of string * expr list * funcid
    | Anonymous of string * expr list * Ast.func_def list (*
    Evaluation is delayed *)
    | Literal of Ast.lit
    | Assign of expr * expr  (* memory := data -- whether memory
     is good is a semantic issue *)
    | Deref of expr * expr (* road[pavement] *)
    | Field of expr * string (* road.pavement *)
    | Invoc of expr * string * expr list * funcid (* receiver.
    method(args) * bestmethod_uid  *)
    | Unop of Ast.op * expr (* !x *)
    | Binop of expr * Ast.op * expr (* x + y *)
    | Refine of string * expr list * string option *
    refine_switch (* refinement, arg list, opt ret type, switch
    *)
    | Refinable of string * refine_switch (* desired refinement,
     list of classes supporting refinement *)

(** An expression with a type tag *)
and expr = string * expr_detail

(** A statement tagged with an environment *)
and sstmt =
    | Decl of Ast.var_def * expr option * environment
    | If of (expr option * sstmt list) list * environment
    | While of expr * sstmt list * environment
    | Expr of expr * environment
    | Return of expr option * environment
    | Super of expr list * string * string * environment (**
    arglist, uidof super init, superclass, env**)

(** A function definition *)
and func_def = {
    returns : string option;
    host    : string option;
    name    : string;
    static  : bool;
    formals : Ast.var_def list;
    body    : sstmt list;
    section : Ast.class_section;  (* Makes things easier later
    *)
    inklass : string;
    uid     : string;
    builtin : bool;
}

(* A member is either a variable or some sort of function *)
type member_def = VarMem of Ast.var_def | MethodMem of func_def
    | InitMem of func_def

(* Things that can go in a class *)
type class_sections_def = {
```

```
    privates : member_def list;
    protects : member_def list;
    publics  : member_def list;
    refines  : func_def list;
    mains    : func_def list;
}

(* Just pop init and main in there? *)
type class_def = {
    klass    : string;
    parent   : string option;
    sections : class_sections_def;
}

type program = class_def list
```

Source 65: `Sast.mli`

```
open StringModules

(* The detail of an expression *)
type cexpr_detail =
    | This
    | Null
    | Id of string * Sast.varkind (* name, local/instance *)
    | NewObj of string * string * cexpr list (* ctype * fname *
    args *)
    | NewArr of string * string * cexpr list (* type (with []'s)
     * fname * args (sizes) *)
    | Literal of Ast.lit
    | Assign of cexpr * cexpr  (* memory := data -- whether
    memory is good is a semantic issue *)
    | Deref of cexpr * cexpr (* road[pavement] *)
    | Field of cexpr * string (* road.pavement *)
    | Invoc of cexpr * string * cexpr list (*Invoc(receiver,
    functionname, args) *)
    | Unop of Ast.op * cexpr (* !x *)
    | Binop of cexpr * Ast.op * cexpr (* x + y *)
    | Refine of cexpr list * string option * Sast.refine_switch
    (* arg list, opt ret type, switch list (class, uids) *)
    | Refinable of Sast.refine_switch (* list of classes
    supporting refinement *)

(* The expression and its type *)
and cexpr = string * cexpr_detail

(* A statement which has cexpr detail *)
and cstmt =
    | Decl of Ast.var_def * cexpr option * Sast.environment
    | If of (cexpr option * cstmt list) list * Sast.environment
    | While of cexpr * cstmt list * Sast.environment
    | Expr of cexpr * Sast.environment
    | Super of string * string * cexpr list (* class, fuid, args
     *)
    | Return of cexpr option * Sast.environment
```

```

(* A c func is a simplified function (no host, etc) *)
and cfunc = {
    returns : string option;
    name    : string;  (* Combine uid and name into this *)
    formals : Ast.var_def list;
    body    : cstmt list;
    builtin : bool;
    inklass : string; (* needed for THIS *)
    static  : bool;
}

(* The bare minimum for a struct represention *)
type class_struct = (string * Ast.var_def list) list (* All the data for this object from the root (first item) down, paired with class name *)

(* A main is a class name and a function name for that main *)
type main_func = (string * string)

(* We actually need all the ancestry information, cause we're gonna do it the right way [lists should go from object down] *)
type ancestry_info = (string list) lookup_map

(* A program is a map from all classes to their struct's, a list of all functions, and a list of mainfuncs, and ancestor information *)
type program = class_struct lookup_map * cfunc list * main_func list * ancestry_info
```

Source 66: `Cast.mli`

```
#!/bin/bash

function errwith {
  echo "$1" >&2
  exit 1
}

function run_file {
  test "$#" -lt 1 && errwith "Please give a file to test"
  file=$1

  test -e "$file" || errwith "File $file does not exist."
  test -f "$file" || errwith "File $file is not a file."

  echo "=========================================================================="
  echo "=========================================================================="
  echo "$file"
  cat "$file"
```

```
  echo "================================================================"
  echo "================================================================"
  ./bin/ray "$file" > ctest/test.c && ( cd ctest && ./compile && ./a.out Test )
}

for afile in "${@}" ; do
  run_file "$afile"
done
```

Source 67: `run-compiler-test.sh`

```
open Ast

(** Various utility functions *)

(* Types *)
(**
    Paramaterized variable typing for building binary ASTs
    @see <http://caml.inria.fr/pub/docs/oreilly-book/html/book-ora016.html#toc19> For more details on paramterized typing
*)
type ('a, 'b) either = Left of 'a | Right of 'b

(** Split a list of 'a 'b either values into a pair of 'a list and 'b list *)
let either_split eithers =
    let rec split_eithers (left, right) = function
        | [] -> (List.rev left, List.rev right)
        | (Left(a))::rest -> split_eithers (a::left, right) rest
        | (Right(b))::rest -> split_eithers (left, b::right) rest in
    split_eithers ([], []) eithers

(** Reduce a list of options to the values in the Some constructors *)
let filter_option list =
    let rec do_filter rlist = function
        | [] -> List.rev rlist
        | None::tl -> do_filter rlist tl
        | (Some(v))::tl -> do_filter (v::rlist) tl in
    do_filter [] list

let option_as_list = function
    | Some(v) -> [v]
    | _ -> []

let decide_option x = function
    | true -> Some(x)
    | _ -> None

```

```
(** Lexically compare two lists of comparable items *)
let rec lexical_compare list1 list2 = match list1, list2 with
    | [], [] -> 0
    | [], _  -> -1
    | _, []  -> 1
    | (x::xs), (y::ys) -> if x < y then -1 else if x > y then 1 else lexical_compare xs ys

(**
    Loop through a list and find all the items that are minimum with respect to the total
    ordering cmp. (If an item is found to be a minimum, any item that is found to
    be equal to the item is in the returned list.) Note can return any size list.
    @param cmp A comparator function
    @param alist A list of items
    @return A list of one or more items deemed to be the minimum by cmp.
*)
let find_all_min cmp alist =
    let rec min_find found items = match found, items with
        | _, [] -> List.rev found (* Return in the same order at least *)
        | [], i::is -> min_find [i] is
        | (f::fs), (i::is) -> let result = cmp i f in
            if result = 0 then min_find (i::found) is
            else if result < 0 then min_find [i] is
            else min_find found is in
    min_find [] alist

(**
    Either monad stuffage
    @param value A monad
    @param func A function to run on a monad
    @return The result of func if we're on the left side, or the error if we're on the right
*)
let (|->) value func =
    match value with
        | Left(v) -> func(v)
        | Right(problem) -> Right(problem)

(** Sequence a bunch of monadic actions together, piping results together *)
let rec seq init actions = match init, actions with
    | Right(issue), _ -> Right(issue)
    | Left(data), [] -> Left(data)
    | Left(data), act::ions -> seq (act data) ions

(**
    Return the length of a block -- i.e. the total number of statements (recursively) in it
    @param stmt_list A list of stmt type objects
    @return An int encoding the length of a block
*)
let get_statement_count stmt_list =
```

```
    let rec do_count stmts blocks counts = match stmts, blocks with
        | [], [] -> counts
        | [], _ -> do_count blocks [] counts
        | (stmt::rest), _ -> match stmt with
            | Decl(_) -> do_count rest blocks (counts + 1)
            | Expr(_) -> do_count rest blocks (counts + 1)
            | Return(_) -> do_count rest blocks (counts + 1)
            | Super(_) -> do_count rest blocks (counts + 1)
            | While(_, block) -> do_count rest (block @ blocks) (counts + 1)
            | If(parts) ->
                let ifblocks = List.map snd parts in
                let ifstmts = List.flatten ifblocks in
                do_count rest (ifstmts @ blocks) (counts + 1) in
    do_count stmt_list [] 0
```

Source 68: `Util.ml`

```
open Parser
open Ast

(** Provides functionality for examining values used in the compilation pipeline. *)

(* TOKEN stuff *)
(** Convert a given token to a string representation for output *)
let token_to_string = function
    | SPACE(n) -> "SPACE(" ^ string_of_int n ^ ")"
    | COLON -> "COLON"
    | NEWLINE -> "NEWLINE"
    | THIS -> "THIS"
    | ARRAY -> "ARRAY"
    | REFINABLE -> "REFINABLE"
    | AND -> "AND"
    | OR -> "OR"
    | XOR -> "XOR"
    | NAND -> "NAND"
    | NOR -> "NOR"
    | NOT -> "NOT"
    | EQ -> "EQ"
    | NEQ -> "NEQ"
    | LT -> "LT"
    | LEQ -> "LEQ"
    | GT -> "GT"
    | GEQ -> "GEQ"
    | LBRACKET -> "LBRACKET"
    | RBRACKET -> "RBRACKET"
    | LPAREN -> "LPAREN"
    | RPAREN -> "RPAREN"
    | LBRACE -> "LBRACE"
    | RBRACE -> "RBRACE"
    | SEMI -> "SEMI"
    | COMMA -> "COMMA"
```

```
    | PLUS -> "PLUS"
    | MINUS -> "MINUS"
    | TIMES -> "TIMES"
    | DIVIDE -> "DIVIDE"
    | MOD -> "MOD"
    | POWER -> "POWER"
    | PLUSA -> "PLUSA"
    | MINUSA -> "MINUSA"
    | TIMESA -> "TIMESA"
    | DIVIDEA -> "DIVIDEA"
    | MODA -> "MODA"
    | POWERA -> "POWERA"
    | IF -> "IF"
    | ELSE -> "ELSE"
    | ELSIF -> "ELSIF"
    | WHILE -> "WHILE"
    | RETURN -> "RETURN"
    | CLASS -> "CLASS"
    | EXTEND -> "EXTEND"
    | SUPER -> "SUPER"
    | INIT -> "INIT"
    | NULL -> "NULL"
    | VOID -> "VOID"
    | REFINE -> "REFINE"
    | REFINES -> "REFINES"
    | TO -> "TO"
    | PRIVATE -> "PRIVATE"
    | PUBLIC -> "PUBLIC"
    | PROTECTED -> "PROTECTED"
    | DOT -> "DOT"
    | MAIN -> "MAIN"
    | NEW -> "NEW"
    | ASSIGN -> "ASSIGN"
    | ID(vid) -> Printf.sprintf "ID(%s)" vid
    | TYPE(tid) -> Printf.sprintf "TYPE(%s)" tid
    | BLIT(bool) -> Printf.sprintf "BLIT(%B)" bool
    | ILIT(inum) -> Printf.sprintf "ILIT(%d)" inum
    | FLIT(fnum) -> Printf.sprintf "FLIT(%f)" fnum
    | SLIT(str) -> Printf.sprintf "SLIT(\"%s\")" (str)
    | EOF -> "EOF"

(** Convert token to its (assumed) lexographical source *)
let descan = function
    | COLON -> ":"
    | NEWLINE -> "\n"
    | SPACE(n) -> String.make n ' '
    | REFINABLE -> "refinable"
    | AND -> "and"
    | OR -> "or"
    | XOR -> "xor"
    | NAND -> "nand"
    | NOR -> "nor"
    | NOT -> "not"
    | EQ -> "="
    | NEQ -> "=/="
    | LT -> "<"
    | LEQ -> "<="
```

```
    | GT -> ">"
    | GEQ -> ">="
    | ARRAY -> "[]"
    | LBRACKET -> "["
    | RBRACKET -> "]"
    | LPAREN -> "("
    | RPAREN -> ")"
    | LBRACE -> "{"
    | RBRACE -> "}"
    | SEMI -> ";"
    | COMMA -> ","
    | PLUS -> "+"
    | MINUS -> "-"
    | TIMES -> "*"
    | DIVIDE -> "/"
    | MOD -> "%"
    | POWER -> "^"
    | PLUSA -> "+="
    | MINUSA -> "-="
    | TIMESA -> "*="
    | DIVIDEA -> "/="
    | MODA -> "%="
    | POWERA -> "^="
    | IF -> "if"
    | ELSE -> "else"
    | ELSIF -> "elsif"
    | WHILE -> "while"
    | RETURN -> "return"
    | CLASS -> "class"
    | EXTEND -> "extends"
    | SUPER -> "super"
    | INIT -> "init"
    | NULL -> "null"
    | VOID -> "void"
    | THIS -> "this"
    | REFINE -> "refine"
    | REFINES -> "refinement"
    | TO -> "to"
    | PRIVATE -> "private"
    | PUBLIC -> "public"
    | PROTECTED -> "protected"
    | DOT -> "."
    | MAIN -> "main"
    | NEW -> "new"
    | ASSIGN -> ":="
    | ID(var) -> var
    | TYPE(typ) -> typ
    | BLIT(b) -> if b then "true" else "false"
    | ILIT(i) -> string_of_int(i)
    | FLIT(f) -> string_of_float(f)
    | SLIT(s) -> Format.sprintf "\"%s\"" s
    | EOF -> "eof"

(**
    Given a lexing function and a lexing buffer, consume tokesn until
    the end of file is reached. Return the generated tokens.
```

```
    @param lexfun A function that takes a lexbuf and returns a token
    @param lexbuf A lexographical buffer from Lexing
    @return A list of scanned tokens
  *)
let token_list (lexfun : Lexing.lexbuf -> token) (lexbuf : Lexing.lexbuf) =
    let rec list_tokens rtokens =
        match (lexfun lexbuf) with
            | EOF -> List.rev (EOF::rtokens)
            | tk -> list_tokens (tk::rtokens) in
    list_tokens []

(**
    Scan a list of tokens from an input file.
    @param source A channel to get tokens from
    @return A list of tokens taken from a source
  *)
let from_channel source = token_list Scanner.token (Lexing.from_channel source)

(**
    Print a list of tokens to stdout.
    @param tokens A list of tokens
    @return Only returns a unit
  *)
let print_token_list tokens = print_string (String.concat " " (List.map token_to_string tokens))

(**
    Used to print out de-whitespacing lines which consist of a number (indentation), a list
    of tokens (the line), and whether there is a colon at the end of the line.
    @return Only returns a unit
  *)
let print_token_line = function
    | (space, toks, colon) ->
        print_string ("(" ^ string_of_int space ^ "," ^ string_of_bool colon ^ ") ");
        print_token_list toks

(**
    Print out a list of tokens with a specific header and some extra margins
    @param header A nonsemantic string to preface our list
    @param toks A list of tokens
    @return Only returns a unit
  *)
let pprint_token_list header toks = print_string header ; print_token_list toks ; print_newline ()

(**
    Print out de-whitespacing lines (see print_token_line) for various lines, but with a header.
    @param header A nonsemantic string to preface our list
    @param lines A list of line representations (number of
```

```
    spaces, if it ends in a colon, a list of tokens)
    @return Only returns a unit
  *)
let pprint_token_lines header lines =
    let spaces = String.make (String.length header) ' ' in
    let rec lines_printer prefix = function
        | line::rest ->
            print_string prefix;
            print_token_line line;
            print_newline ();
            lines_printer spaces rest
        | [] -> () in
    lines_printer header lines

(** The majority of the following functions are relatively direct AST to string operations *)

(* Useful for both sAST and AST *)
let _id x = x
let inspect_str_list stringer a_list = Printf.sprintf "[%s]" (String.concat ", " (List.map stringer a_list))
let inspect_opt stringer = function
    | None -> "None"
    | Some(v) -> Printf.sprintf "Some(%s)" (stringer v)

(* AST Parser Stuff *)
let inspect_ast_lit (lit : Ast.lit) = match lit with
    | Int(i)    -> Printf.sprintf "Int(%d)" i
    | Float(f)  -> Printf.sprintf "Float(%f)" f
    | String(s) -> Printf.sprintf "String(\"%s\")" s
    | Bool(b)   -> Printf.sprintf "Bool(%B)" b

let inspect_ast_arith (op : Ast.arith) = match op with
    | Add  -> "Add"
    | Sub  -> "Sub"
    | Prod -> "Prod"
    | Div  -> "Div"
    | Mod  -> "Mod"
    | Neg  -> "Neg"
    | Pow  -> "Pow"

let inspect_ast_numtest (op : Ast.numtest) = match op with
    | Eq   -> "Eq"
    | Neq  -> "Neq"
    | Less -> "Less"
    | Grtr -> "Grtr"
    | Leq  -> "Leq"
    | Geq  -> "Geq"

let inspect_ast_combtest (op : Ast.combtest) = match op with
    | And  -> "And"
    | Or   -> "Or"
    | Nand -> "Nand"
    | Nor  -> "Nor"
    | Xor  -> "Xor"
    | Not  -> "Not"
```

```
let inspect_ast_op (op : Ast.op) = match op with
    | Arithmetic(an_op) -> Printf.sprintf "Arithmetic(%s)" (inspect_ast_arith an_op)
    | NumTest(an_op)    -> Printf.sprintf "NumTest(%s)" (inspect_ast_numtest an_op)
    | CombTest(an_op)   -> Printf.sprintf "CombTest(%s)" (inspect_ast_combtest an_op)

let rec inspect_ast_expr (expr : Ast.expr) = match expr with
    | Id(id) -> Printf.sprintf "Id(%s)" id
    | This -> "This"
    | Null -> "Null"
    | NewObj(the_type, args) -> Printf.sprintf("NewObj(%s, %s)") the_type (inspect_str_list inspect_ast_expr args)
    | 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)
    | Literal(l) -> Printf.sprintf "Literal(%s)" (inspect_ast_lit l)
    | Invoc(receiver, meth, args) -> Printf.sprintf "Invocation(%s, %s, %s)" (inspect_ast_expr receiver) meth (inspect_str_list inspect_ast_expr args)
    | Field(receiver, field) -> Printf.sprintf "Field(%s, %s)" (inspect_ast_expr receiver) field
    | Deref(var, index) -> Printf.sprintf "Deref(%s, %s)" (inspect_ast_expr var) (inspect_ast_expr var)
    | Unop(an_op, exp) -> Printf.sprintf "Unop(%s, %s)" (inspect_ast_op an_op) (inspect_ast_expr exp)
    | Binop(left, an_op, right) -> Printf.sprintf "Binop(%s, %s, %s)" (inspect_ast_op an_op) (inspect_ast_expr left) (inspect_ast_expr right)
    | Refine(fname, args, totype) -> Printf.sprintf "Refine(%s,%s,%s)" fname (inspect_str_list inspect_ast_expr args) (inspect_opt _id totype)
    | Assign(the_var, the_expr) -> Printf.sprintf "Assign(%s, %s)" (inspect_ast_expr the_var) (inspect_ast_expr the_expr)
    | Refinable(the_var) -> Printf.sprintf "Refinable(%s)" the_var
and inspect_ast_var_def (var : Ast.var_def) = match var with
    | (the_type, the_var) -> Printf.sprintf "(%s, %s)" the_type the_var
and inspect_ast_stmt (stmt : Ast.stmt) = match stmt with
    | Decl(the_def, the_expr) -> Printf.sprintf "Decl(%s, %s)" (inspect_ast_var_def the_def) (inspect_opt inspect_ast_expr the_expr)
    | If(clauses) -> Printf.sprintf "If(%s)" (inspect_str_list inspect_ast_clause clauses)
    | While(pred, body) -> Printf.sprintf "While(%s, %s)" (inspect_ast_expr pred) (inspect_str_list inspect_ast_stmt body)
    | Expr(the_expr) -> Printf.sprintf "Expr(%s)" (inspect_ast_expr the_expr)
    | Return(the_expr) -> Printf.sprintf "Return(%s)" (inspect_opt inspect_ast_expr the_expr)
    | Super(args) -> Printf.sprintf "Super(%s)" (inspect_str_list inspect_ast_expr args)
```

```
and inspect_ast_clause ((opt_expr, body) : Ast.expr option * Ast.stmt list) =
    Printf.sprintf "(%s, %s)" (inspect_opt inspect_ast_expr opt_expr) (inspect_str_list inspect_ast_stmt body)
and inspect_ast_class_section (sect : Ast.class_section) = match sect with
    | Publics  -> "Publics"
    | Protects -> "Protects"
    | Privates -> "Privates"
    | Refines  -> "Refines"
    | Mains    -> "Mains"
and inspect_ast_func_def (func : Ast.func_def) =
    Printf.sprintf "{ returns = %s, host = %s, name = %s, static = %B, formals = %s, body = %s, section = %s, inklass = %s, uid = %s }"
    (inspect_opt _id func.returns)
    (inspect_opt _id func.host)
    func.name
    func.static
    (inspect_str_list inspect_ast_var_def func.formals)
    (inspect_str_list inspect_ast_stmt func.body)
    (inspect_ast_class_section func.section)
    func.inklass
    func.uid

let inspect_ast_member_def (mem : Ast.member_def) = match mem with
    | VarMem(vmem) -> Printf.sprintf "VarMem(%s)" (inspect_ast_var_def vmem)
    | MethodMem(mmem) -> Printf.sprintf "MethodMem(%s)" (inspect_ast_func_def mmem)
    | InitMem(imem) -> Printf.sprintf "InitMem(%s)" (inspect_ast_func_def imem)

let inspect_ast_class_sections (sections : Ast.class_sections_def) =
    Printf.sprintf "{ privates = %s, protects = %s, publics = %s, refines = %s, mains = %s }"
    (inspect_str_list inspect_ast_member_def sections.privates)
    (inspect_str_list inspect_ast_member_def sections.protects)
    (inspect_str_list inspect_ast_member_def sections.publics)
    (inspect_str_list inspect_ast_func_def sections.refines)
    (inspect_str_list inspect_ast_func_def sections.mains)

let inspect_ast_class_def (the_klass : Ast.class_def) =
    Printf.sprintf "{ klass = %s, parent = %s, sections = %s }"
    the_klass.klass
    (inspect_opt _id the_klass.parent)
    (inspect_ast_class_sections the_klass.sections)
```

Source 69: `Inspector.ml`

```
open Util

module StringSet = Set.Make(String)
```

```
module StringMap = Map.Make(String)

(** A place for StringSet and StringMap to live. *)

(**
    Convenience type to make reading table types easier. A lookup_table
    is a primary key -> second key -> value map (i.e. the values of the
    first StringMap are themselves StringMap maps...
  *)
type 'a lookup_table = 'a StringMap.t StringMap.t

(**
    Convenience type to make reading string maps easier. A lookup_map
    is just a StringMap map.
  *)
type 'a lookup_map = 'a StringMap.t


(** Print the contents of a lookup_map *)
let print_lookup_map map stringer =
    let print_item (secondary, item) =
        print_string (stringer secondary item) in
    List.iter print_item (StringMap.bindings map)

(** Print the contents of a lookup_table *)
let print_lookup_table table stringer =
    let print_lookup_map (primary, table) =
        print_lookup_map table (stringer primary) in
    List.iter print_lookup_map (StringMap.bindings table)


(**
    To put it into symbols, we have builder : (StringMap, errorList) -> item -> (StringMap', errorList')
    @param builder A function that accepts a StringMap/(error list) pair and a new item
    and returns a new pair with either and updated map or updated error list
    @param alist The list of data to build the map out of.
  *)
let build_map_track_errors builder alist =
    match List.fold_left builder (StringMap.empty, []) alist with
        | (value, []) -> Left(value)
        | (_, errors) -> Right(errors)

(**
    Look a value up in a map
    @param key The key to look up
    @param map The map to search in
    @return Some(value) or None
  *)
let map_lookup key map = if StringMap.mem key map
    then Some(StringMap.find key map)
```

```
      else None

(**
     Look a list up in a map
     @param key The key to look up
     @param map The map to search in
     @return a list or None
   *)
let map_lookup_list key map = if StringMap.mem key map
     then StringMap.find key map
     else []

(** Updating a string map that has list of possible values *)
let add_map_list key value map =
     let old = map_lookup_list key map in
     StringMap.add key (value::old) map

(** Updating a string map that has a list of possible values
     with a bunch of new values *)
let concat_map_list key values map =
     let old = map_lookup_list key map in
     StringMap.add key (values@old) map

(** Update a map but keep track of collisions *)
let add_map_unique key value (map, collisions) =
     if StringMap.mem key map
          then (map, key::collisions)
          else (StringMap.add key value map, collisions)
```

Source 70: `StringModules.ml`

```
val token_to_string : Parser.token -> string
val descan : Parser.token -> string
val token_list : (Lexing.lexbuf -> Parser.token) -> Lexing.
     lexbuf -> Parser.token list
val from_channel : Pervasives.in_channel -> Parser.token list
val pprint_token_list : string -> Parser.token list -> unit
val pprint_token_lines : string -> (int * Parser.token list *
     bool) list -> unit
val inspect_ast_lit : Ast.lit -> string
val inspect_ast_arith : Ast.arith -> string
val inspect_ast_numtest : Ast.numtest -> string
val inspect_ast_combtest : Ast.combtest -> string
val inspect_ast_op : Ast.op -> string
val inspect_ast_expr : Ast.expr -> string
val inspect_ast_var_def : Ast.var_def -> string
val inspect_ast_stmt : Ast.stmt -> string
val inspect_ast_clause : Ast.expr option * Ast.stmt list ->
     string
val inspect_ast_class_section : Ast.class_section -> string
val inspect_ast_func_def : Ast.func_def -> string
val inspect_ast_member_def : Ast.member_def -> string
val inspect_ast_class_sections : Ast.class_sections_def ->
     string
val inspect_ast_class_def : Ast.class_def -> string
```

Source 71: `Inspector.mli`

```
let _ =
    let tokens = Inspector.from_channel stdin in
    let classes = Parser.cdecls (WhiteSpace.lextoks tokens) (
    Lexing.from_string "") in
    let inspect_classes = List.map Inspector.
    inspect_ast_class_def classes in
    print_string (String.concat "\n\n" inspect_classes);
    print_newline ()
```

Source 72: `inspect.ml`

```
open Parser
open Ast

(**
    A collection of pretty printing functions.
    I don't believe it actually needs the Parser dependency.
    Should probably absorb a fair margin from other files like
    Inspector.ml
  *)

let indent level = String.make (level*2) ' '
let _id x = x

let pp_lit = function
    | Int(i)    -> Printf.sprintf "Int(%d)" i
    | Float(f)  -> Printf.sprintf "Float(%f)" f
    | String(s) -> Printf.sprintf "String(%s)" s
    | Bool(b)   -> Printf.sprintf "Bool(%B)" b

let pp_arith = function
    | Add  -> "Add"
    | Sub  -> "Sub"
    | Prod -> "Prod"
    | Div  -> "Div"
    | Mod  -> "Mod"
    | Neg  -> "Neg"
    | Pow  -> "Pow"

let pp_numtest = function
    | Eq   -> "Eq"
    | Neq  -> "Neq"
    | Less -> "Less"
    | Grtr -> "Grtr"
    | Leq  -> "Leq"
    | Geq  -> "Geq"

let pp_combtest = function
    | And  -> "And"
```

```
    | Or   -> "Or"
    | Nand -> "Nand"
    | Nor  -> "Nor"
    | Xor  -> "Xor"
    | Not  -> "Not"

let pp_op = function
    | Arithmetic(an_op) -> Printf.sprintf "Arithmetic(%s)" (pp_arith an_op)
    | NumTest(an_op)    -> Printf.sprintf "NumTest(%s)" (pp_numtest an_op)
    | CombTest(an_op)   -> Printf.sprintf "CombTest(%s)" (pp_combtest an_op)

let pp_str_list stringer a_list depth = Printf.sprintf "[ %s ]" (String.concat ", " (List.map stringer a_list))
let pp_opt stringer = function
    | None -> "None"
    | Some(v) -> Printf.sprintf "Some(%s)" (stringer v)

let rec pp_expr depth = function
    | Id(id) -> Printf.sprintf "Id(%s)" id
    | This -> "This"
    | Null -> "Null"
    | NewObj(the_type, args) -> Printf.sprintf("\n%sNewObj(%s, %s)") (indent depth) the_type (pp_str_list (pp_expr depth) args depth)
    | 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)
    | Literal(l) -> Printf.sprintf "\n%sLiteral(%s)" (indent depth) (pp_lit l)
    | 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)
    | Field(receiver, field) -> Printf.sprintf "\n%sField(%s, %s)" (indent depth) ((pp_expr depth) receiver) field
    | Deref(var, index) -> Printf.sprintf "\n%sDeref(%s, %s)" (indent depth) ((pp_expr depth) var) ((pp_expr depth) var)
    | Unop(an_op, exp) -> Printf.sprintf "\n%sUnop(%s, %s)" (indent depth) (pp_op an_op) ((pp_expr depth) exp)
    | 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)
    | Refine(fname, args, totype) -> Printf.sprintf "Refine(%s, %s, %s)" fname (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) ((pp_expr (depth+1)) the_var) ((pp_expr (depth+1)) the_expr)
    | Refinable(the_var) -> Printf.sprintf "\n%sRefinable(%s)" (indent depth) the_var
and pp_var_def depth (the_type, the_var) = Printf.sprintf "\n%s(%s, %s)" (indent depth) the_type the_var
and pp_stmt depth = function
    | 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)
    | If(clauses) -> Printf.sprintf "\n%sIf(%s)" (indent depth) (pp_str_list (inspect_clause depth) clauses depth)
    | While(pred, body) -> Printf.sprintf "\n%sWhile(%s, %s)" (indent depth) ((pp_expr depth) pred) (pp_str_list (pp_stmt (depth+1)) body depth)
    | Expr(the_expr) -> Printf.sprintf "\n%sExpr(%s)" (indent depth) ((pp_expr (depth+1)) the_expr)
    | Return(the_expr) -> Printf.sprintf "\n%sReturn(%s)" (indent depth) (pp_opt (pp_expr depth) the_expr)
    | Super(args) -> Printf.sprintf "\n%sSuper(%s)" (indent depth) (pp_str_list (pp_expr depth) args depth)
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)
and class_section = function
    | Publics  -> "Publics"
    | Protects -> "Protects"
    | Privates -> "Privates"
    | Refines  -> "Refines"
    | Mains    -> "Mains"
and pp_func_def depth func = Printf.sprintf "\n%s{\n%sreturns = %s,\n%shost = %s,\n%sname = %s,\n%sstatic = %B,\n%sformals = %s,\n%sbody = %s,\n%ssection = %s,\n%sinklass = %s,\n%suid = %s\n%s}"
    (indent (depth-1))
    (indent depth)
    (pp_opt _id func.returns)
    (indent depth)
    (pp_opt _id func.host)
    (indent depth)
    func.name
    (indent depth)
    func.static
    (indent depth)
    (pp_str_list (pp_var_def (depth+1)) func.formals depth)
    (indent depth)
    (pp_str_list (pp_stmt (depth+1)) func.body depth)
    (indent depth)
    (class_section func.section)
    (indent depth)
    func.inklass
    (indent depth)
    func.uid
    (indent (depth-1))

let pp_member_def depth = function
    | VarMem(vmem) -> Printf.sprintf "\n%sVarMem(%s)" (indent depth) (pp_var_def (depth+1) vmem)
    | MethodMem(mmem) -> Printf.sprintf "\n%sMethodMem(%s)" (indent depth) (pp_func_def (depth+1) mmem)
    | InitMem(imem) -> (*let fmt = "@[<v " ^^ (string_of_int depth) ^^ ">@,InitMem(%s)@]" in*)
        Format.sprintf "\n%sInitMem(%s)@]"
        (indent depth) (pp_func_def (depth+1) imem)
        (*Format.sprintf fmt
```

```
        (pp_func_def (depth+1) imem)*)

let pp_class_sections sections depth =
    Format.sprintf "@[<v 3>@,{@[<v 2>@,privates = %s,@,protects = %s,@,publics = %s,@,refines = %s,@,mains = %s@]@,}@]"
    (pp_str_list (pp_member_def (depth+1)) sections.privates depth)
    (pp_str_list (pp_member_def (depth+1)) sections.protects depth)
    (pp_str_list (pp_member_def (depth+1)) sections.publics depth)
    (pp_str_list (pp_func_def (depth+1)) sections.refines depth)
    (pp_str_list (pp_func_def (depth+1)) sections.mains depth)

let pp_class_def the_klass =
    Format.sprintf "@[<v>@,{@[<v 2>@,klass = %s,@,parent = %s,@,sections = %s@]@,}@]"
    the_klass.klass
    (pp_opt _id the_klass.parent)
    (pp_class_sections the_klass.sections 3)
```

Source 73: `Pretty.ml`

```
(** A global UID generator *)

(** The number of digits in a UID [error after rollover] *)
let uid_digits = 8

(**
    A function to return the a fresh UID. Note that UIDs are copies,
    so they need not be copied on their own
  *)
let uid_counter =
    let counter = String.make uid_digits '0' in
    let inc () =
        let i = ref (uid_digits - 1) in
        while (!i >= 0) && (String.get counter (!i) = 'z') do
            String.set counter (!i) '0' ;
            i := !i - 1
        done ;
        String.set counter (!i) (match String.get counter (!i) with
            | '9' -> 'A'
            | 'Z' -> 'a'
            | c -> char_of_int (int_of_char c + 1));
        String.copy counter in
    inc
```

Source 74: `UID.ml`

```

if [ "${#@}" -eq 0 ] ; then
```

```
  # Read from stdin when there are no arguments (runtool)
  cat
  exit 0
fi

dir="$1"
file="$2"
shift 2

type="Brace"
if [ ${#@} -ne 0 ] ; then
  case "$1" in
    -b) type="Brace"
        ;;
    -s) type="Space"
        ;;
    -m1) type="Mixed1"
        ;;
    *)  echo "Unknown meta-directory $1" >&2
        exit 1
        ;;
  esac
fi

cat "test/tests/${type}/${dir}/${file}"
```

Source 75: `tools/show-example`

```

program="$( basename "$0" )"
if [ ${#@} -lt 3 ] ; then
  echo "Usage: $program dir file tool [-s|-b|-m1]" >&2
  exit 1
fi

dir="$1"
file="$2"
tool="$3"
shift 3

type="Brace"
if [ ${#@} -ne 0 ] ; then
  case "$1" in
    -b) type="Brace"
        ;;
    -s) type="Space"
        ;;
    -m1) type="Mixed1"
        ;;
    *)  echo "Unknown meta-directory $1" >&2
        exit 1
        ;;
  esac
fi

```

```
tool="$( basename "$tool" )"
if [ ! -e "tools/${tool}" ] ; then
  echo "Cannot find tool '${tool}' to execute." >&2
  exit 1
fi

test -e "tools/${tool}"
cat "test/tests/${type}/${dir}/${file}" | "tools/${tool}" "$@"
```

Source 76: `tools/runtool`

```
open Ast
open Sast
open Cast
open Klass
open StringModules
open GlobalData

let to_fname fuid fname = Format.sprintf "f_%s_%s" fuid fname
let to_aname fuid fname = Format.sprintf "a_%s_%s" fuid fname
let to_rname fuid fhost fname = Format.sprintf "f_%s_%s_%s" fuid fhost fname
let to_dispatch fuid fhost fname = Format.sprintf "d_%s_%s_%s" fuid fhost fname

let get_fname (f : Sast.func_def) = to_fname f.uid f.name
let get_rname (f : Sast.func_def) = match f.host with
    | None -> raise(Failure("Generating refine name for non-refinement " ^ f.name ^ " in class " ^ f.inklass ^ "."))
    | Some(host) -> to_rname f.uid host f.name
let get_vname vname = "v_" ^ vname
let get_pointer typ = ("t_"^(Str.global_replace (Str.regexp "\\[\\]") "*" typ));;

let get_tname tname =
    let fixtypes str = try
        let splitter n = (String.sub str 0 n, String.sub str n (String.length str - n)) in
        let (before, after) = splitter (String.index str '*') in (String.trim before) ^ " " ^ (String.trim after)
    with Not_found -> str ^ " " in
fixtypes (get_pointer tname)


let from_tname tname = String.sub tname 2 (String.length tname - 3)
let opt_tname = function
    | None -> None
    | Some(atype) -> Some(get_tname atype)
let get_vdef (vtype, vname) = (get_tname vtype, get_vname vname)

let cast_switch meth refine =
    let update_klass klass = get_tname klass in
    let update_dispatch (klass, uid) = (get_tname klass, to_rname uid meth refine) in
```

```
    let update_test klass = get_tname klass in
    function
        | Switch(klass, cases, uid) -> Switch(update_klass klass, List.map update_dispatch cases, to_dispatch uid meth refine)
        | Test(klass, klasses, uid) -> Test(update_klass klass, List.map update_test klasses, to_dispatch uid meth refine)

(*Convert the sast expr to cast expr*)
let rec sast_to_castexpr mname env (typetag, sastexpr) = (get_tname typetag, c_expr_detail mname sastexpr env)
and sast_to_castexprlist mname env explist = List.map (sast_to_castexpr mname env) explist

(* Convert the sast expr_detail to cast_expr detail; convert names / types / etc *)
and c_expr_detail mname sastexp env = match sastexp with
    | Sast.This                                  -> Cast.This
    | Sast.Null                                  -> Cast.Null
    | Sast.Id(vname)                             -> Cast.Id(get_vname vname, snd (StringMap.find vname env))
    | Sast.NewObj(klass, args, BuiltIn(fuid))    -> Cast.NewObj(klass, fuid, sast_to_castexprlist mname env args)
    | Sast.NewObj(klass, args, FuncId(fuid))     -> Cast.NewObj(klass, to_fname fuid "init", sast_to_castexprlist mname env args)
    | Sast.NewObj(klass, args, ArrayAlloc(fuid)) -> Cast.NewArr(get_tname klass, to_aname fuid "array_alloc", sast_to_castexprlist mname env args)
    | Sast.Literal(lit)                          -> Cast.Literal(lit)
    | Sast.Assign(e1, e2)                        -> Cast.Assign(sast_to_castexpr mname env e1, sast_to_castexpr mname env e2)
    | Sast.Deref(e1, e2)                         -> Cast.Deref(sast_to_castexpr mname env e1, sast_to_castexpr mname env e2)
    | Sast.Field(e1, field)                      -> Cast.Field(sast_to_castexpr mname env e1, get_vname field)
    | Sast.Invoc(recv, fname, args, BuiltIn(fuid)) -> Cast.Invoc(sast_to_castexpr mname env recv, fuid, sast_to_castexprlist mname env args)
    | 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)
    | Sast.Invoc(_, _, _, ArrayAlloc(_))         -> raise(Failure "Cannot allocate an array in an invocation, that is nonsensical.")
    | Sast.Unop(op, expr)                        -> Cast.Unop(op, sast_to_castexpr mname env expr)
    | Sast.Binop(e1, op, e2)                     -> Cast.Binop(sast_to_castexpr mname env e1, op, sast_to_castexpr mname env e2)
    | Sast.Refine(name, args, rtype, switch)     -> Cast.Refine(sast_to_castexprlist mname env args, opt_tname rtype, cast_switch mname name switch)
    | Sast.Refinable(name, switch)               -> Cast.
```

```
    Refinable(cast_switch mname name switch)
    | Anonymous(_, _, _)                         -> raise(Failure("Anonymous objects should have been deanonymized."))

(*Convert the statement list by invoking cstmt on each of the sast stmt*)
let rec cstmtlist mname slist =  List.map (cstmt mname) slist

(* Prepend suffixes *)
and cdef vdef = get_vdef vdef

(*convert sast statement to c statements*)
and cstmt mname sstmt =
    let getoptexpr env = function
        | Some exp -> Some(sast_to_castexpr mname env exp)
        | None     -> None in

    let rec getiflist env = function
        | []                  -> []
        | [(optexpr, slist)]   -> [(getoptexpr env optexpr, cstmtlist mname slist)]
        | (optexpr, slist)::tl -> (getoptexpr env optexpr, cstmtlist mname slist)::(getiflist env tl) in

    let getsuper args fuid parent env =
        let init = if BuiltIns.is_built_in parent then fuid else to_fname fuid "init" in
        let cargs = sast_to_castexprlist mname env args in
        Cast.Super(parent, init, cargs) in

    match sstmt with
        | Sast.Decl(var_def, optexpr, env)      -> Cast.Decl(cdef var_def, getoptexpr env optexpr, env)
        | Sast.If(iflist, env)                  -> Cast.If(getiflist env iflist, env)
        | Sast.While(expr, sstmtlist, env)      -> Cast.While(sast_to_castexpr mname env expr, cstmtlist mname sstmtlist, env)
        | Sast.Expr(exp, env)                   -> Cast.Expr(sast_to_castexpr mname env exp, env)
        | Sast.Return(optexpr, env)             -> Cast.Return(getoptexpr env optexpr, env)
        | Sast.Super(args, fuid, parent, env)   -> getsuper args fuid parent env

(**
    Trim up the sast func_def to the cast cfunc_def
    @param func It's a sast func_def. Woo.
    @return It's a cast cfunc_def. Woo.
*)
let sast_to_cast_func (func : Sast.func_def) : cfunc =
    let name = match func.host, func.builtin with
        | _, true -> func.uid
        | None, _ -> get_fname func
        | Some(host), _ -> get_rname func in
    {   returns = opt_tname func.returns;
        name = name;
```

```
        formals = List.map get_vdef func.formals;
        body = cstmtlist func.name func.body;
        builtin = func.builtin;
        inklass = func.inklass;
        static  = func.static;
    }

let build_class_struct_map klass_data (sast_classes : Sast.class_def list) =
    (* Extract the ancestry and variables from a class into a cdef *)
    let klass_to_struct klass_name (aklass : Ast.class_def) =
        let compare (_, n1) (_, n2) = Pervasives.compare n1 n2 in
        let ivars = List.flatten (List.map snd (Klass.klass_to_variables aklass)) in
        let renamed = List.map get_vdef ivars in
        [(klass_name, List.sort compare renamed)] in

    (* Map each individial class to a basic class_struct *)
    let struct_map = StringMap.mapi klass_to_struct klass_data.classes in

    (* Now, assuming we get parents before children, update the maps appropriately *)
    let folder map = function
        | "Object" -> StringMap.add (get_tname "Object") (StringMap.find "Object" struct_map) map
        | aklass ->
            let parent = StringMap.find aklass klass_data.parents in
            let ancestors = StringMap.find (get_tname parent) map in
            let this = StringMap.find aklass struct_map in
            StringMap.add (get_tname aklass) (this @ ancestors) map in

    (* Update the map so that each child has information from parents *)
    let struct_map = List.fold_left folder StringMap.empty (Klass.get_class_names klass_data) in

    (* Reverse the values so that they start from the root *)
    StringMap.map List.rev struct_map

let sast_functions (klasses : Sast.class_def list) =
    (* Map a Sast class to its functions *)
    let get_functions (klass : Sast.class_def) =
        let s = klass.sections in
        let funcs = function
            | Sast.MethodMem(m) -> Some(m)
            | Sast.InitMem(i) -> Some(i)
            | _ -> None in
        let get_funcs mems = Util.filter_option (List.map funcs mems) in
        List.flatten [ get_funcs s.publics ; get_funcs s.protects ; get_funcs s.privates ; s.refines ; s.mains ] in
```

```

    let all_functions = List.flatten (List.map get_functions klasses) in
    let all_mains = List.flatten (List.map (fun k -> k.sections.mains) klasses) in

    (all_functions, all_mains)

let leaf_ancestors klass_data =
    let leaves = get_leaves klass_data in
    let mangled l = List.map get_tname (map_lookup_list l klass_data.ancestors) in
    let ancestors l = (l, List.rev (mangled l)) in
    List.map ancestors leaves

let sast_to_cast klass_data (klasses : Sast.class_def list) : Cast.program =
    let (funcs, mains) = sast_functions klasses in
    let main_case (f : Sast.func_def) = (f.inklass, get_fname f) in
    let cfuncs = List.map sast_to_cast_func funcs in
    let main_switch = List.map main_case mains in
    let struct_map = build_class_struct_map klass_data klasses in
    let ancestor_data = klass_data.ancestors in

    (struct_map, cfuncs, main_switch, StringMap.map List.rev ancestor_data)

let built_in_names =
    let klass_names = List.map (fun (f : Ast.class_def) -> get_tname f.klass) BuiltIns.built_in_classes in
    List.fold_left (fun set i -> StringSet.add i set) StringSet.empty klass_names
```

Source 77: `GenCast.ml`

```
open Util

val klass_to_parent : Ast.class_def -> string
val section_string : Ast.class_section -> string
val klass_to_variables : Ast.class_def -> (Ast.class_section * Ast.var_def list) list
val klass_to_methods : Ast.class_def -> (Ast.class_section * Ast.func_def list) list
val klass_to_functions : Ast.class_def -> (Ast.class_section * Ast.func_def list) list
val conflicting_signatures : Ast.func_def -> Ast.func_def -> bool
val signature_string : Ast.func_def -> string
val full_signature_string : Ast.func_def -> string
val class_var_lookup : GlobalData.class_data -> string -> string -> (Ast.class_section * string) option
val class_field_lookup : GlobalData.class_data -> string -> string -> (string * string * Ast.class_section) option
```

```
val class_field_far_lookup : GlobalData.class_data -> string ->
    string -> bool -> ((string * string * Ast.class_section),
    bool) either
val class_method_lookup : GlobalData.class_data -> string ->
    string -> Ast.func_def list
val class_ancestor_method_lookup : GlobalData.class_data ->
    string -> string -> bool -> Ast.func_def list
val refine_lookup : GlobalData.class_data -> string -> string ->
     string -> Ast.func_def list
val refinable_lookup : GlobalData.class_data -> string -> string
     -> string -> Ast.func_def list
val get_distance : GlobalData.class_data -> string -> string ->
    int option
val is_type : GlobalData.class_data -> string -> bool
val is_subtype : GlobalData.class_data -> string -> string ->
    bool
val is_proper_subtype : GlobalData.class_data -> string ->
    string -> bool
val compatible_formals : GlobalData.class_data -> string list ->
     string list -> bool
val compatible_function : GlobalData.class_data -> string list
    -> Ast.func_def -> bool
val compatible_return : GlobalData.class_data -> string option
    -> Ast.func_def -> bool
val compatible_signature : GlobalData.class_data -> string
    option -> string list -> Ast.func_def -> bool
val best_matching_signature : GlobalData.class_data -> string
    list -> Ast.func_def list -> Ast.func_def list
val best_method : GlobalData.class_data -> string -> string ->
    string list -> Ast.class_section list -> Ast.func_def option
val best_inherited_method : GlobalData.class_data -> string ->
    string -> string list -> bool -> Ast.func_def option
val refine_on : GlobalData.class_data -> string -> string ->
    string -> string list -> string option -> Ast.func_def list
val get_class_names : GlobalData.class_data -> string list
val get_leaves : GlobalData.class_data -> string list
```

Source 78: `Klass.mli`

```
open Ast
open Str

(** Built in classes *)

let built_in cname : Ast.func_def = match Str.split (regexp "_")
  cname with
    | [] -> raise(Failure "Bad cname -- empty.")
    | [klass] -> raise(Failure("Bad cname -- just class: " ^
    klass))
    | klass::func ->
        let methname = match func with
            | [] -> raise(Failure("Impossible!"))
            | func::rest -> func ^ (String.concat "" (List.map
    String.capitalize rest)) in
        { returns = None;
```

```
            host = None;
            name = methname;
            static = false;
            formals = [];
            body = [];
            section = Publics;
            inklass = String.capitalize klass;
            uid = cname;
            builtin = true }
let breturns cname atype = { (built_in cname) with returns =
    Some(atype) }
let btakes cname formals = { (built_in cname) with formals =
    formals }

let sections : Ast.class_sections_def =
    { publics = [];
      protects = [];
      privates = [];
      refines = [];
      mains = [] }

let func f = if f.name = "init" then InitMem(f) else MethodMem(f
    )
let var v = VarMem(v)
let variables = List.map var
let functions = List.map func
let members f v = (functions f) @ (variables v)

let class_object : Ast.class_def =
    let name = "Object" in

    let init_obj : Ast.func_def = { (built_in "object_init")
    with section = Protects } in
    let system = ("System", "system") in

    let sections : Ast.class_sections_def =
        { sections with
          publics = [];
          protects = [func init_obj; var system] } in

    { klass = name; parent = None; sections = sections }

let class_scanner : Ast.class_def =
    let name = "Scanner" in

    let scan_line : Ast.func_def = breturns "scanner_scan_string
    " "String" in
    let scan_int : Ast.func_def = breturns "scanner_scan_integer
    " "Integer" in
    let scan_float : Ast.func_def = breturns "scanner_scan_float
    " "Float" in
    let scan_init : Ast.func_def = built_in "scanner_init" in

    let sections : Ast.class_sections_def =
        { sections with
          publics = functions [scan_line; scan_int; scan_float;
    scan_init] } in
```

```
    { klass = name; parent = None; sections = sections }

let class_printer : Ast.class_def =
    let name = "Printer" in

    let print_string : Ast.func_def = btakes "
    printer_print_string" [("String", "arg")] in
    let print_int : Ast.func_def = btakes "printer_print_integer
    " [("Integer", "arg")] in
    let print_float : Ast.func_def = btakes "printer_print_float
    " [("Float", "arg")] in
    let print_init : Ast.func_def = btakes "printer_init" [("
    Boolean", "stdout")] in

    let sections : Ast.class_sections_def =
        { sections with
          publics = functions [print_string; print_int;
    print_float; print_init] } in

    { klass = name; parent = None; sections = sections }

let class_string : Ast.class_def =
    let name = "String" in

    let string_init : Ast.func_def = built_in "string_init" in

    let sections : Ast.class_sections_def =
        { sections with
          protects = [func string_init] } in

    { klass = name; parent = None; sections = sections }


let class_boolean : Ast.class_def =
    let name = "Boolean" in

    let boolean_init : Ast.func_def = built_in "boolean_init" in

    let sections : Ast.class_sections_def =
        { sections with
          protects = [func boolean_init] } in

    { klass = name; parent = None; sections = sections }

let class_integer : Ast.class_def =
    let name = "Integer" in

    let integer_init : Ast.func_def = built_in "integer_init" in
    let integer_float : Ast.func_def = breturns "integer_to_f" "
    Float" in

    let sections : Ast.class_sections_def =
        { sections with
          publics = [func integer_float];
          protects = [func integer_init] } in

```

```
    { klass = name; parent = None; sections = sections }

let class_float : Ast.class_def =
    let name = "Float" in

    let float_init : Ast.func_def = built_in "float_init" in
    let float_integer : Ast.func_def = breturns "float_to_i" "
    Integer" in

    let sections : Ast.class_sections_def =
        { sections with
          publics = [func float_integer];
          protects = [func float_init] } in

    { klass = name; parent = None; sections = sections }

let class_system : Ast.class_def =
    let name = "System" in

    let system_init : Ast.func_def = built_in "system_init" in
    let system_exit : Ast.func_def = btakes "system_exit" [("
    Integer", "code")] in

    let system_out = ("Printer", "out") in
    let system_err = ("Printer", "err") in
    let system_in = ("Scanner", "in") in
    let system_argc = ("Integer", "argc") in

    let sections : Ast.class_sections_def =
        { sections with
          publics = members [system_init; system_exit] [
    system_out; system_err; system_in; system_argc]; } in

    { klass = name; parent = None; sections = sections }

(** The list of built in classes and their methods *)
let built_in_classes =
  [ class_object; class_string; class_boolean; class_integer;
    class_float; class_printer; class_scanner; class_system ]

(** Return whether a class is built in or not *)
let is_built_in name =
  List.exists (fun klass -> klass.klass = name) built_in_classes
```

Source 79: `BuiltIns.ml`

```
open Ast
open Util
open StringModules

(** Module for getting sets of variables *)

(** Get the formal variables of a function *)
let formal_vars func =
    let add_param set (_, v) = StringSet.add v set in
```

```
    List.fold_left add_param StringSet.empty func.formals

(** Get the free variables of a list of statements *)
let free_vars bound stmts =
    let rec get_free_vars free = function
        | [] -> free
        | (bound, Left(stmts))::todo -> get_free_stmts free bound todo stmts
        | (bound, Right(exprs))::todo -> get_free_exprs free bound todo exprs
    and get_free_stmts free bound todo = function
        | [] -> get_free_vars free todo
        | stmt::rest ->
            let (expr_block_list, stmt_block_list, decl) = match stmt with
                | Decl(((_, var), e)) -> ([option_as_list e], [], Some(var))
                | Expr(e)             -> ([[e]], [], None)
                | Return(e)           -> ([option_as_list e], [], None)
                | Super(es)           -> ([es], [], None)
                | While(e, body)      -> ([[e]], [body], None)
                | If(parts)           -> let (es, ts) = List.split parts in
                                          ([filter_option es], ts, None) in
            let expressions = List.map (function exprs -> (bound, Right(exprs))) expr_block_list in
            let statements  = List.map (function stmts -> (bound, Left(stmts))) stmt_block_list in
            let bound = match decl with
                | Some(var) -> StringSet.add var bound
                | _ -> bound in
            get_free_stmts free bound (expressions @ statements @ todo) rest
    and get_free_exprs free bound todo = function
        | [] -> get_free_vars free todo
        | expr::rest ->
            let func_to_task bound func =
                (StringSet.union (formal_vars func) bound, Left(func.body)) in

            let (exprs, tasks, id) = match expr with
                | NewObj(_, args)           -> (args, [], None)
                | Assign(l, r)              -> ([l; r], [], None)
                | Deref(v, i)               -> ([v; i], [], None)
                | Field(e, _)               -> ([e], [], None)
                | Invoc(e, _, args)         -> (e::args, [], None)
                | Unop(_, e)                -> ([e], [], None)
                | Binop(l, _, r)            -> ([l; r], [], None)
                | Refine(_, args, _)        -> (args, [], None)
                | This                      -> ([], [], None)
                | Null                      -> ([], [], None)
```

```
                     | Refinable(_)              -> ([], [], None)
                     | Literal(_)                -> ([], [], None)
                     | Id(id)                    -> ([], [], decide_option id (not (StringSet.mem id bound)))
                     | Anonymous(_, args, funcs) -> (args, List.map (func_to_task bound) funcs, None) in

                let rest = exprs @ rest in
                let todo = tasks @ todo in
                let free = match id with
                     | Some(id) -> StringSet.add id free
                     | None -> free in
                get_free_exprs free bound todo rest in

    get_free_vars StringSet.empty [(bound, Left(stmts))]

(** Get the free variables in a function. *)
let free_vars_func bound func =
    let params = formal_vars func in
    free_vars (StringSet.union bound params) func.body

(** Get the free variables in a whole list of functions. *)
let free_vars_funcs bound funcs =
    let sets = List.map (free_vars_func bound) funcs in
    List.fold_left StringSet.union StringSet.empty sets
```

Source 80: `Variables.ml`

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

Source 81: `ctest/compile`

```
open Util

let show_classes builder classes = match builder classes with
    | Left(data) -> KlassData.print_class_data data; exit(0)
    | Right(issue) -> Printf.fprintf stderr "%s\n" (KlassData.errstr issue); exit(1)

let from_input builder =
    let tokens = Inspector.from_channel stdin in
    let classes = Parser.cdecls (WhiteSpace.lextoks tokens) (Lexing.from_string "") in
    show_classes builder classes
let from_basic builder = show_classes builder []

let basic_info_test () = from_basic KlassData.build_class_data_test
let basic_info () = from_basic KlassData.build_class_data

let test_info () = from_input KlassData.build_class_data_test
let normal_info () = from_input KlassData.build_class_data

```

```
let exec name func = Printf.printf "Executing mode %s\n" name;
    flush stdout; func ()

let _ = try
    Printexc.record_backtrace true;
    match Array.to_list Sys.argv with
        | []      -> raise(Failure("Not even program name given
    as argument."))
        | [_]     -> exec "Normal Info" normal_info
        | _::arg::_ -> match arg with
        | "-"     -> exec "Basic Info"  basic_info
        | "--"    -> exec "Basic Test"  basic_info_test
        | _       -> exec "Test Info"   test_info
with _ ->
    Printexc.print_backtrace stderr
```

Source 82: `classinfo.ml`

```
#!/bin/bash

testdir="$( dirname "$0" )"
testprogram=".testdrive"

"$testdir/$testprogram" "$0" "inspect" "expect-parser" "$@"
```

Source 83: `test/parser`

```
test types:
  * Brace  -- these should be with {, }, and ;
  * Mixed1 -- these should be mixed (closer to Space for now)
  * Space  -- these should be with :

in each type there are test folders:
  * Empty   -- structurally empty tests
  * Trivial -- just above empty, should do something... trivial
  * Simple  -- some basic programs, more than just trivial

each test type requires the same tests. at the end, the outputs
    are compared
```

Source 84: `test/README`

```
#!/bin/bash

program="$( basename "$1" )"
scriptdir="$( dirname "$1" )"
exe="./tools/$2"
old="$3"
shift 3

# Arguments
```

```
justrun=
save=
verbose=
pattern=*
folderpattern=*

# Calculated values change in each iteration
current=
results=

# Don't change per iteration
tmpfile="test/check"
tmperr="test/err"
testdir="test/tests"
maxlength=0
oneline=0
files=()
folders=()
temp=()
errored=0
dropadj=1

# Formatting values
bold=`tput bold`
normal=`tput sgr0`
uline=`tput smul`
green=`tput setaf 2`
red=`tput setaf 1`
blue=`tput setaf 4`
backblue=`tput setab 4`

function errWith {
  echo "$1" >&2
  exit 1
}

function execerror {
  echo "${bold}${uline}${red}ERROR${normal} $1"
  errored=1
}

function dots {
  local len=`echo "$current" | wc -c`
  for i in `seq $len $maxlength` ; do
    echo -n '.'
  done
  echo -n ' '
}

function contains {
  local elem
  for elem in "${@:2}" ; do
    test "$elem" = "$1" && return 0
  done
  return 1
}
```

```
function dropdirprefix {
  echo "$1" | cut -c $(( ${#2} + $dropadj ))-
}

function setdropadj {
  local result=$( dropdirprefix "/dev/null" "/dev/" )
  local null="null"
  dropadj=$(( dropadj + (${#null} - ${#result}) ))
}

function show_standard {
  echo "${red}Standard -- START${normal}"
  cat "$results"
  echo "${red}Stadard -- END${normal}"
}

function testit {
  local testing="${bold}Testing:${normal} ${uline}${current}${normal}"
  test "$oneline" -eq 0 && echo "$testing"
  test "$oneline" -ne 0 && echo -n "$testing"
  test "$oneline" -ne 0 && dots
  test -n "$verbose" && cat "$1"
  if [ -n "$justrun" ] ; then
    cat "$1" | "$exe"
    return 0
  fi
  cat "$1" | "$exe" 1> "$tmpfile" 2> "$tmperr"
  if [ $? -ne 0 ] ; then
    execerror "Error testing $program with $current"
    cat "$tmperr"
  elif [ -n "$save" ] ; then
    echo "${bold}Saving${normal} $current"
    mkdir -p $( dirname "$results" )
    mv "$tmpfile" "$results"
  elif [ ! -e "$results" ] ; then
    execerror "Cannot check results -- standard does not exist"
  else
    if [ -n "$verbose" ] ; then
      echo -n "${bold}Output:${normal} "
      cat "$tmpfile"
    fi
    test "$oneline" -eq 0 && echo -n "${bold}Results:${normal} "
    diff -q "$tmpfile" "$results" &> /dev/null
    if [ $? -eq 0 ] ; then
      echo "${bold}${green}PASS${normal}"
    else
      echo "${bold}${red}MISMATCH${normal}"
      test -n "$verbose" && show_standard
    fi
  fi

  test -e "$tmpfile" && rm "$tmpfile"  # Sometimes happens
  test -e "$tmperr" && rm "$tmperr"    # Always happens

  test "$oneline" -eq 0 && echo ""
}
```

```
function listandexit {
  for afile in $( find "$testdir" -type f -name "$pattern" ) ;
    do
    current=$( dropdirprefix "$afile" "$testdir" )
    echo "$current"
  done
  exit 0
}

function usage {
cat <<USAGE
$program -[chlpsv]
  -f pattern
      Filter meta-folders by pattern

  -h
      Display this help

  -l
      Display the name of all tests; note that pattern can be
    used

  -p pattern
      Filter tests to be used based on pattern (as in find -name)

  -R
      merely run the driving exe and output the result to stdout
    (no checking anything)

  -s
      save results

  -v
      verbose output
USAGE
  exit 0
}

setdropadj

while getopts "f:hlRsvp:" OPTION ; do
  case "$OPTION" in
    f) folderpattern=$OPTARG ;;
    h) usage ;;
    R) justrun=1 ;;
    s) save=1 ;;
    v) verbose=1 ;;
    p) pattern=$OPTARG ;;
    l) list=1;;
    ?) errWith "Unknown option; aborting" ;;
  esac
done
shift $(($OPTIND - 1))

test -n "$list" && listandexit
```

```
test -e "$exe" || errWith "Testing $program but $exe unavailable"
test -f "$exe" || errWith "Testing $program but $exe is not a file"
test -x "$exe" || errWith "Testing $program but $exe unexecutable"

test -z "$verbose" && oneline=1

for adir in $( find "$testdir" -mindepth 1 -maxdepth 1 -type d -name "$folderpattern" ) ; do
  adir=$( dropdirprefix "$adir" "$testdir/" )
  folders+=( "$adir" )
done
test "${#folders[@]}" -eq 0 && errWith "No folders in test directory. Good-bye."

for afolder in "${folders[@]}" ; do
  test -d "$testdir/$afolder" || errWith "$afolder is not a directory ($testdir)"
done

for afile in $( find "$testdir/${folders[0]}" -type f -name "$pattern" ) ; do
  test "README" = $( basename "$afile" ) || files+=( $( dropdirprefix "$afile" "$testdir/${folders[0]}/" ) )
done

for afolder in "${folders[@]}" ; do
  temp=()
  for afile in $( find "$testdir/$afolder" -type f -name "$pattern" ) ; do
    test "README" = $( basename "$afile" ) || temp+=( $( dropdirprefix "$afile" "$testdir/$afolder/" ) )
  done

  for afile in "${files[@]}" ; do
    contains "$afile" "${temp[@]}" || errWith "$afolder does not contain $afile but ${folders[0]} does"
  done
  for bfile in "${temp[@]}" ; do
    contains "$bfile" "${files[@]}" || errWith "$afolder contains $bfile but ${folders[0]} does not"
  done
done
test "${#files[@]}" -eq 0 && errWith "No files match the given pattern. Good-bye."

# All the test directories have the same structure.
for current in "${files[@]}" ; do
  len=`echo "$current" | wc -c`
  test $len -gt $maxlength && maxlength="$len"
done
maxlength=$(( maxlength + 5 ))

for afolder in "${folders[@]}" ; do
  echo "${bold}${blue}Testing:${normal} $afolder"
```

```
  for current in "${files[@]}" ; do
    results="test/$old/$afolder/$current"
    testit "$testdir/$afolder/$current"
  done
done

test $errored -eq 1 && exit 1
test -n "$justrun" && exit 0

# Ensure that all the results are the same.
for current in "${files[@]}" ; do
  master="test/$old/${folders[0]}/$current"
  matched=1

  for afolder in "${folders[@]}" ; do
    target="test/$old/$afolder/$current"
    diff -q "$master" "$target" &> /dev/null
    if [ $? -ne 0 ] ; then
      echo "$current ${bold}${red}DIFFERS${normal} between ${folders[0]} (reference) and $afolder"
      matched=0
    fi
  done
  test $matched -eq 1 && echo "$current ${bold}${green}MATCHES${normal} across all folders"
done
```

Source 85: `test/.testdrive`

```
#!/bin/bash

testdir="$( dirname "$0" )"
testprogram=".testdrive"

"$testdir/$testprogram" "$0" "prettify" "expect-ast-pretty" "$@"
```

Source 86: `test/ast-pretty`

```
#!/bin/bash

testdir="$( dirname "$0" )"
testprogram=".testdrive"

"$testdir/$testprogram" "$0" "streams" "expect-scanner" "$@"
```

Source 87: `test/scanner`

```
class List {
}
```

Source 88: `test/tests/Brace/Empty/Class`

```
class List {
  public {
    init() {
    }
    void noop() {
    }
  }
}
```

Source 89: `test/tests/Brace/Empty/InitMethod`

```
class List {
  refinement {
  }
}
```

Source 90: `test/tests/Brace/Empty/Refinements`

```
class List {
  public {
    void noop() {
    }
  }
}
```

Source 91: `test/tests/Brace/Empty/Method`

```
class List {
  private {
  }
}
```

Source 92: `test/tests/Brace/Empty/Private`

```
class List {
  public {
    void noop() {
      while(true) {
      }
    }
  }
}
```

Source 93: `test/tests/Brace/Empty/WhileMethod`

```
class List {
  public {
    init() {
    }
  }
}
```

Source 94: `test/tests/Brace/Empty/Init`

```
class List {
  public {
  }
}
```

Source 95: `test/tests/Brace/Empty/Public`

```
class List {
  protected {
  }
}
```

Source 96: `test/tests/Brace/Empty/Protected`

```
class List {
  public {
    void noop() {
      if(true) {
      }
    }
  }
}
```

Source 97: `test/tests/Brace/Empty/IfMethod`

```
class Collection {
  protected {
    init() {
    }
  }

  public {
    Boolean mutable() {
      return refine answer() to Boolean;
```

```
    }

    void add(Object item) {
      refine do(item) to void;
    }

    void addAll(Collection other) {
      if(refinable(do)) {
        refine combine(other) to void;
      } else {
        Iterator items := other.iterator();
        while(not items.done()) {
          add(items.next());
        }
      }
    }

    void clear() {
      refine do() to void;
    }

    Boolean contains(Object item) {
      if(refinable(check)) {
        return refine check(item) to Boolean;
      }

      Iterator items := this.iterator();
      while(not items.done()) {
        if(items.next() = item) {
          return true;
        }
      }
      return false;
    }

    Boolean containsAll(Collection other) {
      if(refinable(check)) {
        return refine check(other) to Boolean;
      }

      Iterator items := other.iterator();
      while(not items.done()) {
        if(not this.contains(items.next())) {
          return false;
        }
      }
      return true;
    }
  }
}
```

Source 98: `test/tests/Brace/Multi/Collection`

```
class List extends Node {
  public {
```

```
    init() {
      Int c;
      c := 1234;
    }
  }
}
```

Source 99: `test/tests/Brace/Trivial/InitStatement`

```
class Rectangle extends Shape {
  public {
    init(Int width, Int height) {
      this.width := width;
      this.height := height;
    }
    Int area() {
      return width * height;
    }
    Int perimeter() {
      return 2 * (width + height);
    }
  }
  protected {
    Int width;
    Int height;
  }
}
```

Source 100: `test/tests/Brace/Simple/Rectangle`

```
class List:
```

Source 101: `test/tests/Mixed1/Empty/Class`

```
class List:
  public:
    init():
    void noop() {
    }
```

Source 102: `test/tests/Mixed1/Empty/InitMethod`

```
class List:
  refinement {
  }
```

Source 103: `test/tests/Mixed1/Empty/Refinements`

```
class List:
  public:
    void noop() {
    }
```

Source 104: `test/tests/Mixed1/Empty/Method`

```
class List:
  private {
  }
```

Source 105: `test/tests/Mixed1/Empty/Private`

```
class List:
  public:
    void noop():
      while(true){

      }
```

Source 106: `test/tests/Mixed1/Empty/WhileMethod`

```
class List:
  public:
    init() {
    }
```

Source 107: `test/tests/Mixed1/Empty/Init`

```
class List:
  public {
  }
```

Source 108: `test/tests/Mixed1/Empty/Public`

```
class List:
  protected {
  }
```

Source 109: `test/tests/Mixed1/Empty/Protected`

```
class List:
  public:
    void noop(){
```

```
    if(true){}
  }
```

Source 110: `test/tests/Mixed1/Empty/IfMethod`

```
class Collection:
  protected:
    init() {
    }

  public:
    Boolean mutable() {
      return refine answer() to Boolean;
    }

    void add(Object item):
      refine do(item) to void

    void addAll(Collection other):
      if(refinable(do)) {
        refine combine(other) to void;
      } else:
        Iterator items := other.iterator()
        while(not items.done()) {
          add(items.next());
        }

    void clear():
      refine do() to void

    Boolean contains(Object item):
      if(refinable(check)):
        return refine check(item) to Boolean

      Iterator items := this.iterator()
      while(not items.done()):
        if(items.next() = item) {
          return true;
        }
      return false

    Boolean containsAll(Collection other):
      if(refinable(check)) {
        return refine check(other) to Boolean;
      }

      Iterator items := other.iterator()
      while(not items.done()):
        if(not this.contains(items.next())):
          return false
      return true
```

Source 111: `test/tests/Mixed1/Multi/Collection`

```
class List extends Node:
  public:
    init() {
      Int c;
      c := 1234;
    }
```

Source 112: `test/tests/Mixed1/Trivial/InitStatement`

```
class Rectangle extends Shape:
  public:
    init(Int width, Int height) {
      this.width := width;
      this.height := height;
    }

    Int area():
      return width * height

    Int perimeter():
      return 2 * (width + height)

  protected {
    Int width;
    Int height;
  }
```

Source 113: `test/tests/Mixed1/Simple/Rectangle`

```
class List:
```

Source 114: `test/tests/Space/Empty/Class`

```
class List:
  public:
    init():
    void noop():
```

Source 115: `test/tests/Space/Empty/InitMethod`

```
class List:
  refinement:
```

Source 116: `test/tests/Space/Empty/Refinements`

```
class List:
```

```
  public:
    void noop():
```

Source 117: `test/tests/Space/Empty/Method`

```
class List:
  private:
```

Source 118: `test/tests/Space/Empty/Private`

```
class List:
  public:
    void noop():
      while(true):
```

Source 119: `test/tests/Space/Empty/WhileMethod`

```
class List:
  public:
    init():
```

Source 120: `test/tests/Space/Empty/Init`

```
class List:
  public:
```

Source 121: `test/tests/Space/Empty/Public`

```
class List:
  protected:
```

Source 122: `test/tests/Space/Empty/Protected`

```
class List:
  public:
    void noop():
      if(true):
```

Source 123: `test/tests/Space/Empty/IfMethod`

```
class Collection:
  protected:
```

```
    /* Only subclasses can be created */
    init():

  public:
    Boolean mutable():
      return refine answer() to Boolean

    void add(Object item):
      refine do(item) to void

    void addAll(Collection other):
      if (refinable(do)):
        refine combine(other) to void
      else:
        Iterator items := other.iterator()
        while (not items.done()):
          add(items.next())

    void clear():
      refine do() to void

    Boolean contains(Object item):
      if (refinable(check)):
        return refine check(item) to Boolean

      Iterator items := this.iterator()
      while (not items.done()):
        if (items.next() = item):
          return true
      return false

    Boolean containsAll(Collection other):
      if (refinable(check)):
        return refine check(other) to Boolean

      Iterator items := other.iterator()
      while (not items.done()):
        if (not this.contains(items.next())):
          return false
      return true
```

Source 124: `test/tests/Space/Multi/Collection`

```
class List extends Node:
  public:
    init():
      Int c;
      c := 1234;
```

Source 125: `test/tests/Space/Trivial/InitStatement`

```
class Rectangle extends Shape:
  public:
```

```
    init(Int width, Int height):
      this.width := width
      this.height := height

    Int area():
      return width * height

    Int perimeter():
      return 2 * (width + height)

  protected:
    Int width
    Int height
```

Source 126: `test/tests/Space/Simple/Rectangle`

```
open StringModules
open Sast
open Ast
open Util

(** Take a collection of Sast class_defs and deanonymize them. *)


(** The data needed to deanonymize a list of classes and store the results. *)
type anon_state = {
    labeler : int lookup_map ;      (** Label deanonymized classes *)
    deanon : Ast.class_def list ;   (** List of Ast.class_def classes that are deanonymized. *)
    clean : Sast.class_def list ;   (** List of clean Sast.class_def classes *)
    data : GlobalData.class_data ;  (** A class_data record used for typing *)
    current : string ;              (** The class that is currently being examined *)
}

(**
    Given the initial anon_state, an environment, and an expr_detail, remove all
    anonymous object instantiations from the expr and replace them with the
    instantiation of a newly constructed class. This returns a changed expr_detail
    value and an updated state -- i.e. maybe a new ast class is added to it.
    @param init_state anon_state value
    @param env an environment (like those attached to statements in sAST)
    @param expr_deets an expr_detail to transform
    @return (new expr detail, updated state)
  *)
```

```
let rec deanon_expr_detail init_state env expr_deets =
    let get_label state klass =
        let (n, labeler) = match map_lookup klass state.labeler with
            | None -> (0, StringMap.add klass 0 state.labeler)
            | Some(n) -> (n+1, StringMap.add klass (n+1) state.labeler) in
        (Format.sprintf "anon_%s_%d" klass n, { state with labeler = labeler }) in

    let get_var_type state env var_name =
        match map_lookup var_name env with
            | Some(vinfo) -> Some(fst vinfo)
            | None -> match Klass.class_field_lookup state.data state.current var_name with
                | Some((_, vtype, _)) -> Some(vtype)
                | _ -> None in

    let deanon_init args formals klass : Ast.func_def =
        let givens = List.map (fun (t, _) -> (t, "Anon_v_" ^ UID.uid_counter ())) args in
        let all_formals = givens @ formals in
        let super = Ast.Super(List.map (fun (_, v) -> Ast.Id(v)) givens) in
        let assigner (_, vname) = Ast.Expr(Ast.Assign(Ast.Field(Ast.This, vname), Ast.Id(vname))) in
        {   returns = None;
            host = None;
            name = "init";
            static = false;
            formals = all_formals;
            body = super::(List.map assigner formals);
            section = Publics;
            inklass = klass;
            uid = UID.uid_counter ();
            builtin = false } in

    let deanon_klass args freedefs klass parent refines =
        let init = deanon_init args freedefs klass in
        let vars = List.map (fun vdef -> Ast.VarMem(vdef)) freedefs in
        let sections =
            {   privates = vars;
                protects = [];
                publics = [InitMem(init)];
                refines = List.map (fun r -> { r with inklass= klass }) refines;
                mains = []; } in
        let theklass =
            {   klass = klass;
                parent = Some(parent);
                sections = sections } in
        (init.uid, theklass) in

    let deanon_freedefs state env funcs =
        let freeset = Variables.free_vars_funcs StringSet.empty funcs in
```

```
        let freevars = List.sort compare (StringSet.elements freeset) in

        let none_snd = function
          | (None, v) -> Some(v)
          | _ -> None in
        let some_fst = function
          | (Some(t), v) -> Some((t, v))
          | _ -> None in
        let add_type v = (get_var_type state env v, v) in

        let typed = List.map add_type freevars in
        let unknowns = List.map none_snd typed in
        let knowns = List.map some_fst typed in

        match Util.filter_option unknowns with
            | [] -> Util.filter_option knowns
            | vs -> raise(Failure("Unknown variables " ^ String.concat ", " vs ^ " within anonymous object definition.")) in

    match expr_deets with
        | Sast.Anonymous(klass, args, refines) ->
            let (newklass, state) = get_label init_state klass in
            let freedefs = deanon_freedefs state env refines in
            let (init_id, ast_class) = deanon_klass args freedefs newklass klass refines in
            let freeargs = List.map (fun (t, v) -> (t, Sast.Id(v))) freedefs in
            let instance = Sast.NewObj(newklass, args @ freeargs, Sast.FuncId init_id) in
            let state = { state with deanon = ast_class::state.deanon } in
            (instance, state)
        | Sast.This -> (Sast.This, init_state)
        | Sast.Null -> (Sast.Null, init_state)
        | Sast.Id(id) -> (Sast.Id(id), init_state)
        | Sast.NewObj(klass, args, funcid) ->
            let (args, state) = deanon_exprs init_state env args in
            (Sast.NewObj(klass, args, funcid), state)
        | Sast.Literal(lit) -> (Sast.Literal(lit), init_state)
        | Sast.Assign(mem, data) ->
            let (mem, state) = deanon_expr init_state env mem in
            let (data, state) = deanon_expr state env data in
            (Sast.Assign(mem, data), state)
        | Sast.Deref(arr, idx) ->
            let (arr, state) = deanon_expr init_state env arr in
            let (idx, state) = deanon_expr state env idx in
            (Sast.Deref(arr, idx), state)
        | Sast.Field(expr, mbr) ->
            let (expr, state) = deanon_expr init_state env expr in
            (Sast.Field(expr, mbr), state)
        | Sast.Invoc(recvr, klass, args, funcid) ->
            let (recvr, state) = deanon_expr init_state env recvr in
```

```
                let (args, state) = deanon_exprs state env args in
                (Sast.Invoc(recvr, klass, args, funcid), state)
            | Sast.Unop(op, expr) ->
                let (expr, state) = deanon_expr init_state env expr
    in
                (Sast.Unop(op, expr), state)
            | Sast.Binop(l, op, r) ->
                let (l, state) = deanon_expr init_state env l in
                let (r, state) = deanon_expr state env r in
                (Sast.Binop(l, op, r), state)
            | Sast.Refine(refine, args, ret, switch) ->
                let (args, state) = deanon_exprs init_state env args
     in
                (Sast.Refine(refine, args, ret, switch), state)
            | Sast.Refinable(refine, switch) ->
                (Sast.Refinable(refine, switch), init_state)

(**
    Update an type-tagged sAST expression to be deanonymized.
    Returns the deanonymized expr and a possibly updated
    anon_state
    @param init_state anon_state value
    @param env an environment like those attached to stmts in
    the sAST
    @param t the type of the expr_detail exp
    @param exp an expression detail
    @return ((t, exp'), state') where exp' is exp but
    deanonymized and
    state' is an updated version of init_state
  *)
and deanon_expr init_state env (t, exp) =
    let (deets, state) = deanon_expr_detail init_state env exp
    in
    ((t, deets), state)

(**
    Deanonymize a list of expressions maintaining the state
    properly throughout.
    Returns the list of expressions (deanonymized) and the
    updated state.
    @param init_state an anon_state value
    @param env an environment like those attached to statments (
    sAST)
    @param list a list of expressions (sAST exprs)
    @return (list', state') where list' is the deanonymized list
     and
    state' is the updated state
  *)
and deanon_exprs init_state env list =
    let folder (rexprs, state) expr =
        let (deets, state) = deanon_expr state env expr in
        (deets::rexprs, state) in
    let (rexprs, state) = List.fold_left folder ([], init_state)
     list in
    (List.rev rexprs, state)

(**
```

```
    Deanonymize a statement.
    Returns the deanonymized statement and the updated state.
    @param input_state an anon_state value
    @param stmt a statement to deanonymize
    @return (stmt', state') the statement and state, updated.
  *)
and deanon_stmt input_state stmt =
    let deanon_decl init_state env = function
        | (vdef, Some(expr)) ->
            let (deets, state) = deanon_expr init_state env expr in
            (Sast.Decl(vdef, Some(deets), env), state)
        | (vdef, _) -> (Sast.Decl(vdef, None, env), init_state) in

    let deanon_exprstmt init_state env expr =
        let (deets, state) = deanon_expr init_state env expr in
        (Sast.Expr(deets, env), state) in

    let deanon_return init_state env = function
        | None -> (Sast.Return(None, env), init_state)
        | Some(expr) ->
            let (deets, state) = deanon_expr init_state env expr in
            (Sast.Return(Some(deets), env), state) in

    let deanon_super init_state env args built_in init_id =
        let (deets, state) = deanon_exprs init_state env args in
        (Sast.Super(deets, init_id, built_in, env), state) in

    let deanon_while init_state env (expr, stmts) =
        let (test, state) = deanon_expr init_state env expr in
        let (body, state) = deanon_stmts state stmts in
        (Sast.While(test, body, env), state) in

    let deanon_if init_state env pieces =
        let folder (rpieces, state) piece =
            let (piece, state) = match piece with
                | (None, stmts) ->
                    let (body, state) = deanon_stmts state stmts in
                    ((None, body), state)
                | (Some(expr), stmts) ->
                    let (test, state) = deanon_expr state env expr in
                    let (body, state) = deanon_stmts state stmts in
                    ((Some(test), body), state) in
            (piece::rpieces, state) in
        let (rpieces, state) = List.fold_left folder ([], init_state) pieces in
        (Sast.If(List.rev rpieces, env), state) in

    match stmt with
        | Sast.Decl(vdef, opt_expr, env) -> deanon_decl input_state env (vdef, opt_expr)
        | Sast.If(pieces, env) -> deanon_if input_state env
```

```
pieces
        | Sast.While(test, body, env) -> deanon_while
    input_state env (test, body)
        | Sast.Expr(expr, env) -> deanon_exprstmt input_state
    env expr
        | Sast.Return(opt_expr, env) -> deanon_return
    input_state env opt_expr
        | Sast.Super(args, init_id, built_in, env) ->
    deanon_super input_state env args built_in init_id

(**
    Update an entire list of statements to be deanonymized.
    Maintains the update to the state throughout the computation
    .
    Returns a deanonymized list of statements and an updated
    state.
    @param init_state an anon_state value
    @param stmts a list of statements
    @return (stmts', state') the updated statements and state
  *)
and deanon_stmts init_state stmts =
    let folder (rstmts, state) stmt =
        let (stmt, state) = deanon_stmt state stmt in
        (stmt::rstmts, state) in
    let (rstmts, state) = List.fold_left folder ([], init_state)
     stmts in
    (List.rev rstmts, state)

(**
    Deanonymize the body of a function.
    Return the updated function and updated state.
    @param init_state an anon_state value
    @param func a func_def (sAST)
    @return (func', state') the updated function and state
  *)
let deanon_func init_state (func : Sast.func_def) =
    let (stmts, state) = deanon_stmts init_state func.body in
    ({ func with body = stmts }, state)

(**
    Deanonymize an entire list of functions, threading the state
    throughout and maintaining the changes. Returns the list of
    functions, updated, and the updated state.
    @param init_state an anon_state value
    @param funcs a list of functions
    @return (funcs', state') the updated functions and state
  *)
let deanon_funcs init_state funcs =
    let folder (rfuncs, state) func =
        let (func, state) = deanon_func state func in
        (func::rfuncs, state) in
    let (funcs, state) = List.fold_left folder ([], init_state)
    funcs in
    (List.rev funcs, state)

(**
    Deanonymize an Sast member_def
```

```
    Returns the deanonymized member and a possibly updated state.
    @param init_state an anon_state value
    @param mem a member to deanonymize
    @return (mem', state') the updated member and state
  *)
let deanon_member init_state mem = match mem with
    | Sast.MethodMem(f) ->
        let (func, state) = deanon_func init_state f in
        (Sast.MethodMem(func), state)
    | Sast.InitMem(f) ->
        let (func, state) = deanon_func init_state f in
        (Sast.InitMem(func), state)
    | mem -> (mem, init_state)

(**
    Deanonymize a list of members.  Return the deanonymized list
    and a possibly updated state.
    @param init_state an anon_state value
    @param members a list of members to deanonymize
    @return (mems', state') the updated members and state
  *)
let deanon_memlist (init_state : anon_state) (members : Sast.member_def list) : (Sast.member_def list * anon_state) =
    let folder (rmems, state) mem =
        let (mem, state) = deanon_member state mem in
        (mem::rmems, state) in
    let (rmems, state) = List.fold_left folder ([], init_state) members in
    (List.rev rmems, state)

(**
    Deanonymize an entire class. Return the deanonymized class
    and an updated state.
    @param init_state an anon_state value
    @param aklass an sAST class to deanonymize
    @return (class', state') the udpated class and state.
  *)
let deanon_class init_state (aklass : Sast.class_def) =
    let s = aklass.sections in
    let state = { init_state with current = aklass.klass } in
    let (publics, state) = deanon_memlist state s.publics in
    let (protects, state) = deanon_memlist state s.protects in
    let (privates, state) = deanon_memlist state s.privates in
    let (refines, state) = deanon_funcs state s.refines in
    let (mains, state) = deanon_funcs state s.mains in
    let sections : Sast.class_sections_def =
        {   publics = publics;
            protects = protects;
            privates = privates;
            refines = refines;
            mains = mains } in
    let cleaned = { aklass with sections = sections } in
    (state.deanon, { state with clean = cleaned::state.clean; current = ""; deanon = [] })

(** A startng state for deanonymization. *)
```

```
let empty_deanon_state data =
    {   labeler = StringMap.empty;
        deanon = [];
        clean = [];
        data = data;
        current = ""; }

(**
    Given global class information and parsed and tagged classes
    ,
    deanonymize the classes. This will add more classes to the
    global data, which will be updated accordingly.
    @param klass_data global class_data info
    @param sast_klasses tagged sAST class list
    @return If everything goes okay with updating the global
    data
    for each deanonymization, then left((state', data')) will be
    returned where state' contains all (including newly created)
    sAST classes in its clean list and data' has been updated to
    reflect any new classes. If anything goes wrong, Right(issue
    )
    is returned, where the issue is just as in building the
    global
    class_data info to begin with, but now specific to what goes
    on in deanonymization (i.e. restricted to those restricted
    classes themselves).
  *)
let deanonymize klass_data sast_klasses =
    let is_empty = function
        | [] -> true
        | _ -> false in

    let rec run_deanon init_state asts sasts = match asts, sasts
     with
        (* Every sAST has been deanonymized, even the
    deanonymized ones converted into sASTs
         * Every Ast has been sAST'd too. So we are done.
         *)
        | [], [] ->
            if is_empty init_state.deanon then Left((init_state.
    data, init_state.clean))
            else raise(Failure("Deanonymization somehow did not
    recurse properly."))

        | [], klass::rest ->
            let (asts, state) = deanon_class init_state klass in
            run_deanon state asts rest

        | klass::rest, _ -> match KlassData.append_leaf
    init_state.data klass with
            | Left(data) ->
                let sast_klass = BuildSast.ast_to_sast_klass
    data klass in
                let state = { init_state with data = data } in
                run_deanon state rest (sast_klass::sasts)
            | Right(issue) -> Right(issue) in

```

```
    run_deanon (empty_deanon_state klass_data) [] sast_klasses
```

Source 127: `Unanonymous.ml`

```
open StringModules
open Util

val fold_classes : GlobalData.class_data -> ('a -> Ast.class_def
    -> 'a) -> 'a -> 'a
val map_classes : GlobalData.class_data -> ('a StringMap.t ->
    Ast.class_def -> 'a StringMap.t) -> 'a StringMap.t
val dfs_errors : GlobalData.class_data -> (string -> 'a -> 'b ->
    ('a * 'b)) -> 'a -> 'b -> 'b

val build_class_data : Ast.class_def list -> (GlobalData.
    class_data, GlobalData.class_data_error) either
val build_class_data_test : Ast.class_def list -> (GlobalData.
    class_data, GlobalData.class_data_error) either

val append_leaf : GlobalData.class_data -> Ast.class_def -> (
    GlobalData.class_data, GlobalData.class_data_error) either
val append_leaf_test : GlobalData.class_data -> Ast.class_def ->
    (GlobalData.class_data, GlobalData.class_data_error) either

val print_class_data : GlobalData.class_data -> unit
val errstr : GlobalData.class_data_error -> string
```

Source 128: `KlassData.mli`

```
open Ast
open Util
open StringModules
open GlobalData
open Klass

(** Build a class_data object. *)

(** Construct an empty class_data object *)
let empty_data : class_data = {
    known = StringSet.empty;
    classes = StringMap.empty;
    parents = StringMap.empty;
    children = StringMap.empty;
    variables = StringMap.empty;
    methods = StringMap.empty;
    refines = StringMap.empty;
    mains = StringMap.empty;
    ancestors = StringMap.empty;
    distance = StringMap.empty;
    refinable = StringMap.empty;
}

(**
```

```
    Map function collisions to the type used for collection that
     information.
    This lets us have a ‘standard’ form of method / refinement
    collisions and so
    we can easily build up a list of them.
    @param aklass the class we are currently examining (class
    name -- string)
    @param funcs a list of funcs colliding in aklass
    @param reqhost are we requiring a host (compiler error if no
     host and true)
    @return a tuple representing the collisons - (class name,
    collision tuples)
    where collision tuples are ([host.]name, formals)
  *)
let build_collisions aklass funcs reqhost =
    let to_collision func =
        let name = match func.host, reqhost with
            | None, true -> raise(Invalid_argument("Cannot build
     refinement collisions -- refinement without host [compiler
    error]."))
            | None, _ -> func.name
            | Some(host), _ -> host ^ "." ^ func.name in
        (name, List.map fst func.formals) in
    (aklass, List.map to_collision funcs)

(** Fold over the values in a class_data record's classes map.
    *)
let fold_classes data folder init =
    let do_fold _ aklass result = folder result aklass in
    StringMap.fold do_fold data.classes init

(**
    Fold over the values in a class_data record's classes map,
    but
    enforce building up a StringMap.
  *)
let map_classes data folder = fold_classes data folder StringMap
    .empty

(**
    Recursively explore the tree starting at the root,
    accumulating errors
    in a list as we go. The explorer function should take the
    current class
    the current state, the current errors and return a new state
     / errors
    pair (updating state when possible if there are errors for
    further
    accumulation). This is the state that will be passed to all
    children,
    and the errors will accumulate across all children.
    @param data A class_data record value
    @param explore Something that goes from the current node to
    a new state/error pair
    @init_state the initial state of the system
    @init_error the initial errors of the system
    @return The final accumulated errors
```

```
  *)
let dfs_errors data explore init_state init_error =
    let rec recurse aklass state errors =
        let (state, errors) = explore aklass state errors in
        let explore_kids errors child = recurse child state errors in
        let children = map_lookup_list aklass data.children in
        List.fold_left explore_kids errors children in
    recurse "Object" init_state init_error

(**
    Given a list of classes, build an initial class_data object with
    the known and classes fields set appropriately. If there are any
    duplicate class names a StringSet of the collisions will then be
    returned in Right, otherwise the data will be returned in Left.
    @param klasses A list of classes
    @return Left(data) which is a class_data record with the known
    set filled with names or Right(collisions) which is a set of
    collisions (StringSet.t)
  *)
let initialize_class_data klasses =
    let build_known (set, collisions) aklass =
        if StringSet.mem aklass.klass set
            then (set, StringSet.add aklass.klass collisions)
            else (StringSet.add aklass.klass set, collisions) in
    let klasses = BuiltIns.built_in_classes @ klasses in
    let build_classes map aklass = StringMap.add aklass.klass aklass map in
    let (known, collisions) = List.fold_left build_known (StringSet.empty, StringSet.empty) klasses in
    let classes = List.fold_left build_classes StringMap.empty klasses in
    if StringSet.is_empty collisions
        then Left({ empty_data with known = known; classes = classes })
        else Right(collisions)

(**
    Given an initialized class_data record, build the children map
    from the classes that are stored within it.
    The map is from parent to children list.
    @param data A class_data record
    @return data but with the children.
  *)
let build_children_map data =
    let map_builder map aklass = match aklass.klass with
        | "Object" -> map
        | _ -> add_map_list (klass_to_parent aklass) aklass.klass map in
    let children_map = map_classes data map_builder in
    { data with children = children_map }
```

```
(**
    Given an initialized class_Data record, build the parent map
    from the classes that are stored within it.
    The map is from child to parent.
    @param data A class_data record
    @return data but with the parent map updated.
  *)
let build_parent_map data =
    let map_builder map aklass = match aklass.klass with
        | "Object" -> map
        | _ -> StringMap.add (aklass.klass) (klass_to_parent aklass) map in
    let parent_map = map_classes data map_builder in
    { data with parents = parent_map }

(**
    Validate that the parent map in a class_data record represents a tree rooted at object.
    @param data a class_data record
    @return An optional string (Some(string)) when there is an issue.
  *)
let is_tree_hierarchy data =
    let rec from_object klass checked =
        match map_lookup klass checked with
            | Some(true) -> Left(checked)
            | Some(false) -> Right("Cycle detected.")
            | _ -> match map_lookup klass data.parents with
                | None -> Right("Cannot find parent after building parent map: " ^ klass)
                | Some(parent) -> match from_object parent (StringMap.add klass false checked) with
                    | Left(updated) -> Left(StringMap.add klass true updated)
                    | issue -> issue in
    let folder result aklass = match result with
        | Left(checked) -> from_object aklass.klass checked
        | issue -> issue in
    let checked = StringMap.add "Object" true StringMap.empty in
    match fold_classes data folder (Left(checked)) with
        | Right(issue) -> Some(issue)
        | _ -> None

(**
    Add the class (class name - string) -> ancestors (list of ancestors - string list) map to a
    class_data record. Note that the ancestors go from `youngest' to `oldest' and so should start
    with the given class (hd) and end with Object (last item in the list).
    @param data The class_data record to update
    @return An updated class_data record with the ancestor map added.
  *)
let build_ancestor_map data =
    let rec ancestor_builder klass map =
```

```
        if StringMap.mem klass map then map
        else
            let parent = StringMap.find klass data.parents in
            let map = ancestor_builder parent map in
            let ancestors = StringMap.find parent map in
            StringMap.add klass (klass::ancestors) map in
    let folder map aklass = ancestor_builder aklass.klass map in
    let map = StringMap.add "Object" ["Object"] StringMap.empty in
    let ancestor_map = fold_classes data folder map in
    { data with ancestors = ancestor_map }

(**
    For a given class, build a map of variable names to variable information.
    If all instance variables are uniquely named, returns Left (map) where map
    is  var name -> (class_section, type)  otherwise returns Right (collisions)
    where collisions are the names of variables that are multiply declared.
    @param aklass A parsed class
    @return a map of instance variables in the class
  *)
let build_var_map aklass =
    let add_var section map (typeId, varId) = add_map_unique varId (section, typeId) map in
    let map_builder map (section, members) = List.fold_left (add_var section) map members in
    build_map_track_errors map_builder (klass_to_variables aklass)

(**
    Add the class (class name - string) -> variable (var name - string) -> info (section/type
    pair - class_section * string) table to a class_data record.
    @param data A class_data record
    @return Either a list of collisions (in Right) or the updated record (in Left).
    Collisions are pairs (class name, collisions (var names) for that class)
  *)
let build_class_var_map data =
    let map_builder (klass_map, collision_list) (_, aklass) =
        match build_var_map aklass with
            | Left(var_map) -> (StringMap.add (aklass.klass) var_map klass_map, collision_list)
            | Right(collisions) -> (klass_map, (aklass.klass, collisions)::collision_list) in
    match build_map_track_errors map_builder (StringMap.bindings data.classes) with
        | Left(variable_map) -> Left({ data with variables = variable_map })
        | Right(collisions) -> Right(collisions) (* Same value different types parametrically *)

(**
```

```
    Given a class_data record and a class_def value, return the instance variables (just the
    var_def) that have an unknown type.
    @param data A class_data record value
    @param aklass A class_def value
    @return A list of unknown-typed instance variables in the class
  *)
let type_check_variables data aklass =
    let unknown_type (var_type, _) = not (is_type data var_type) in
    let vars = List.flatten (List.map snd (klass_to_variables aklass)) in
    List.filter unknown_type vars

(**
    Given a class_data record, verify that all instance variables of all classes are of known
    types. Returns the Left of the data if everything is okay, or the Right of a list of pairs,
    first item being a class, second being variables of unknown types (type, name pairs).
    @param data A class_data record value.
    @return Left(data) if everything is okay, otherwise Right(unknown types) where unknown types
    is a list of (class, var_def) pairs.
  *)
let verify_typed data =
    let verify_klass klass_name aklass unknowns = match type_check_variables data aklass with
        | [] -> unknowns
        | bad -> (klass_name, bad)::unknowns in
    match StringMap.fold verify_klass data.classes [] with
        | [] -> Left(data)
        | bad -> Right(bad)

(**
    Given a function, type check the signature (Return, Params).
    @param data A class_data record value.
    @param func An Ast.func_def record
    @return Left(data) if everything is alright; Right([host.]name, option string, (type, name)
    list) if wrong.
  *)
let type_check_func data func =
    let atype = is_type data in
    let check_ret = match func.returns with
        | Some(vtype) -> if atype vtype then None else Some(vtype)
        | _ -> None in
    let check_param (vtype, vname) = if not (atype vtype) then Some((vtype, vname)) else None in
    let bad_params = filter_option (List.map check_param func.formals) in
    match check_ret, bad_params, func.host with
        | None, [], _ -> Left(data)
        | _, _, None -> Right((func.name, check_ret, bad_params)
```

```
    )
        | _, _, Some(host) -> Right((host ^ "." ^ func.name, check_ret, bad_params))

(**
    Given a class_data object and a klass, verify that all of its methods have good types
    (Return and parameters).
    @param data A class_data record object
    @param aklass A class_def object
    @return Left(data) if everything went okay; Right((klass name, (func name, option string,
    (type, name) list) list))
  *)
let type_check_class data aklass =
    let folder bad func = match type_check_func data func with
        | Left(data) -> bad
        | Right(issue) -> issue::bad in
    let funcs = List.flatten (List.map snd (klass_to_functions aklass)) in
    match List.fold_left folder [] funcs with
        | [] -> Left(data)
        | bad -> Right((aklass.klass, bad))

(**
    Given a class_data object, verify that all classes have methods with good signatures
    (Return and parameters)
    @param data A class_data record object
    @param aklass A class_def object
    @return Left(data) if everything went okay; Right((klass name, bad_sig list) list)
    where bad_sig is (func_name, string option, (type, var) list))
  *)
let type_check_signatures data =
    let folder klass_name aklass bad = match type_check_class data aklass with
        | Left(data) -> bad
        | Right(issue) -> issue::bad in
    match StringMap.fold folder data.classes [] with
        | [] -> Left(data)
        | bad -> Right(bad)

(**
    Build a map of all the methods within a class, returing either a list of collisions
    (in Right) when there are conflicting signatures or the map (in Left) when there
    are not. Keys to the map are function names and the values are lists of func_def's.
    @param aklass A klass to build a method map for
    @return Either a list of collisions or a map of function names to func_def's.
  *)
let build_method_map aklass =
    let add_method (map, collisions) fdef =
```

```
        if List.exists (conflicting_signatures fdef) (map_lookup_list fdef.name map)
            then (map, fdef::collisions)
            else (add_map_list fdef.name fdef map, collisions) in
    let map_builder map funcs = List.fold_left add_method map funcs in
    build_map_track_errors map_builder (List.map snd (klass_to_methods aklass))

(**
    Add the class name (string) -> method name (string) -> methods (func_def list)
    methods table to a class_data record, given a list of classes. If there are no
    collisions, the updated record is returned (in Left), otherwise the collision
    list is returned (in Right).
    @param data A class data record
    @return Either a list of collisions (in Right) or the updated record (in Left).
    Collisions are pairs (class name, colliding methods for that class). Methods collide
    if they have conflicting signatures (ignoring return type).
  *)
let build_class_method_map data =
    let map_builder (klass_map, collision_list) (_, aklass) =
        match build_method_map aklass with
            | Left(method_map) -> (StringMap.add aklass.klass method_map klass_map, collision_list)
            | Right(collisions) -> (klass_map, (build_collisions aklass.klass collisions false)::collision_list) in
    match build_map_track_errors map_builder (StringMap.bindings data.classes) with
        | Left(method_map) -> Left({ data with methods = method_map })
        | Right(collisions) -> Right(collisions) (* Same value different types parametrically *)

(**
    Build the map of refinements for a given class. Keys to the map are 'host.name'
    @param aklass aklass A class to build a refinement map out of
    @return Either a list of collisions (in Right) or the map (in left). Refinements
    conflict when they have the same name ('host.name' in this case) and have the same
    argument type sequence.
  *)
let build_refinement_map aklass =
    let add_refinement (map, collisions) func = match func.host with
        | Some(host) ->
            let key = func.name ^ "." ^ host in
            if List.exists (conflicting_signatures func) (map_lookup_list key map)
```

```
                then (map, func::collisions)
                else (add_map_list key func map, collisions)
        | None -> raise(Failure("Compilation error -- non-refinement found in searching for refinements.")) in
    build_map_track_errors add_refinement aklass.sections.refines

(**
    Add the class name (string) -> refinement (`host.name' - string) -> func list
    map to a class_data record. If there are no collisions (conflicting signatures
    given the same host), then the updated record is returned (in Left) otherwise
    a list of collisions is returned (in Right).
    @param data A class_data record
    @param klasses A list of parsed classes
    @return either a list of collisions (in Right) or the updated record (in Left).
    Collisions are (class, (host, method, formals) list)
  *)
let build_class_refinement_map data =
    let map_builder (klass_map, collision_list) (_, aklass) =
        match build_refinement_map aklass with
            | Left(refinement_map) -> (StringMap.add aklass.klass refinement_map klass_map, collision_list)
            | Right(collisions) -> (klass_map, (build_collisions aklass.klass collisions true)::collision_list) in
    match build_map_track_errors map_builder (StringMap.bindings data.classes) with
        | Left(refinement_map) -> Left({ data with refines = refinement_map })
        | Right(collisions) -> Right(collisions) (* Same value different types parametrically *)

(**
    Add a map of main functions, from class name (string) to main (func_def) to the
    class_data record passed in. Returns a list of collisions if any class has more
    than one main (in Right) or the updated record (in Left)
    @param data A class_data record
    @param klasses A list of parsed classes
    @return Either the collisions (Right) or the updated record (Left)
  *)
let build_main_map data =
    let add_klass (map, collisions) (_, aklass) = match aklass.sections.mains with
        | [] -> (map, collisions)
        | [main] -> (StringMap.add aklass.klass main map, collisions)
        | _ -> (map, aklass.klass :: collisions) in
    match build_map_track_errors add_klass (StringMap.bindings data.classes) with
        | Left(main_map) -> Left({ data with mains = main_map })
        | Right(collisions) -> Right(collisions) (* Same value
```

```
    different types parametrically *)

(**
    Given a class_data record, verify that there are no double
    declarations of instance
    variables as you go up the tree. This means that no two
    classes along the same root
    leaf path can have the same public / protected variables,
    and a private cannot be
    a public/protected variable of an ancestor.
    @param data A class_data record.
    @return Left(data) if everything was okay or Right(
    collisions) where collisions is
    a list of pairs of collision information - first item class,
     second item a list of
    colliding variables for that class (name, ancestor where
    they collide)
  *)
let check_field_collisions data =
    let check_vars aklass var (section, _) (fields, collisions)
    = match map_lookup var fields, section with
        | Some(ancestor), _ -> (fields, (ancestor, var)::
    collisions)
        | None, Privates -> (fields, collisions)
        | None, _ -> (StringMap.add var aklass fields,
    collisions) in

    let check_class_vars aklass fields =
        let vars = StringMap.find aklass data.variables in
        StringMap.fold (check_vars aklass) vars (fields, []) in

    let dfs_explorer aklass fields collisions =
        match check_class_vars aklass fields with
            | (fields, []) -> (fields, collisions)
            | (fields, cols) -> (fields, (aklass, cols)::
    collisions) in

    match dfs_errors data dfs_explorer StringMap.empty [] with
        | [] -> Left(data)
        | collisions -> Right(collisions)

(**
    Check to make sure that we don't have conflicting signatures
     as we go down the class tree.
    @param data A class_data record value
    @return Left(data) if everything is okay, otherwise a list
    of (string
  *)
let check_ancestor_signatures data =
    let check_sigs meth_name funcs (methods, collisions) =
        let updater (known, collisions) func =
            if List.exists (conflicting_signatures func) known
                then (known, func::collisions)
                else (func::known, collisions) in
        let apriori = map_lookup_list meth_name methods in
        let (known, collisions) = List.fold_left updater (
    apriori, collisions) funcs in
```

```
            (StringMap.add meth_name known methods, collisions) in

    let skip_init meth_name funcs acc = match meth_name with
        | "init" -> acc
        | _ -> check_sigs meth_name funcs acc in

    let check_class_meths aklass parent_methods =
        let methods = StringMap.find aklass data.methods in
        StringMap.fold skip_init methods (parent_methods, []) in

    let dfs_explorer aklass methods collisions =
       match check_class_meths aklass methods with
            | (methods, []) -> (methods, collisions)
            | (methods, cols) -> (methods, (build_collisions aklass cols false)::collisions) in

    match dfs_errors data dfs_explorer StringMap.empty [] with
        | [] -> Left(data)
        | collisions -> Right(collisions)

(**
    Verifies that each class is able to be instantiated.
    @param data A class_data record
    @return Either the data is returned in Left or a list of uninstantiable classes in Right
  *)
let verify_instantiable data =
    let uninstantiable klass =
        let inits = class_method_lookup data klass "init" in
        not (List.exists (fun func -> func.section <> Privates) inits) in
    let klasses = StringSet.elements data.known in
    match List.filter uninstantiable klasses with
        | [] -> Left(data)
        | bad -> Right(bad)

(**
    Given a class and a list of its ancestors, build a map detailing the distance
    between the class and any of its ancestors. The distance is the number of hops
    one must take to get from the given class to the ancestor. The distance between
    an Object and itself should be 0, and the largest distance should be to object.
    @param klass The class to build the table for
    @param ancestors The list of ancestors of the given class.
    @return A map from class names to integers
  *)
let build_distance klass ancestors =
    let map_builder (map, i) item = (StringMap.add item i map, i+1) in
    fst (List.fold_left map_builder (StringMap.empty, 0) ancestors)

(**
    Add a class (class name - string) -> class (class name -
```

```
    string) -> distance (int option)
    table a given class_data record. The distance is always a
    positive integer and so the
    first type must be either the same as the second or a
    subtype, else None is returned.
    Note that this requires that the ancestor map be built.
    @param data The class_data record to update.
    @return The class_data record with the distance map added.
  *)
let build_distance_map data =
    let distance_map = StringMap.mapi build_distance data.
    ancestors in
    { data with distance = distance_map }

(**
    Update the refinement dispatch uid table with a given set of
     refinements.
    @param parent The class the refinements will come from
    @param refines A list of refinements
    @param table The refinement dispatch table
    @return The updated table
  *)
let update_refinable parent refines table =
    let toname f = match f.host with
        | Some(host) -> host
        | _ -> raise(Invalid_argument("Compiler error; we have
    refinement without host for " ^ f.name ^ " in " ^ f.inklass
    ^ ".")) in
    let folder amap f = add_map_list (toname f) f amap in
    let map = if StringMap.mem parent table then StringMap.find
    parent table else StringMap.empty in
    let map = List.fold_left folder map refines in
    StringMap.add parent map table

(**
    Add the refinable (class name -> host.name -> refinables
    list) table to the
    given class_data record, returning the updated record.
    @param data A class_data record info
    @return A class_data object with the refinable updated
  *)
let build_refinable_map data =
    let updater klass_name aklass table = match klass_name with
        | "Object" -> table
        | _ -> let parent = klass_to_parent aklass in
    update_refinable parent aklass.sections.refines table in
    let refinable = StringMap.fold updater data.classes
    StringMap.empty in
    { data with refinable = refinable}

(** These are just things to pipe together building a class_data
     record pipeline *)
let initial_data klasses = match initialize_class_data klasses
    with
    | Left(data) -> Left(data)
    | Right(collisions) -> Right(DuplicateClasses(StringSet.
    elements collisions))
```

```
let append_children data = Left(build_children_map data)
let append_parent data = Left(build_parent_map data)
let test_tree data = match is_tree_hierarchy data with
    | None -> Left(data)
    | Some(problem) -> Right(HierarchyIssue(problem))
let append_ancestor data = Left(build_ancestor_map data)
let append_distance data = Left(build_distance_map data)
let append_variables data = match build_class_var_map data with
    | Left(data) -> Left(data)
    | Right(collisions) -> Right(DuplicateVariables(collisions))
let test_types data = match verify_typed data with
    | Left(data) -> Left(data)
    | Right(bad) -> Right(UnknownTypes(bad))
let test_fields data = match check_field_collisions data with
    | Left(data) -> Left(data)
    | Right(collisions) -> Right(DuplicateFields(collisions))
let append_methods data = match build_class_method_map data with
    | Left(data) -> Left(data)
    | Right(collisions) -> Right(ConflictingMethods(collisions))
let test_init data = match verify_instantiable data with
    | Left(data) -> Left(data)
    | Right(bad) -> Right(Uninstantiable(bad))
let test_inherited_methods data = match check_ancestor_signatures data with
    | Left(data) -> Left(data)
    | Right(collisions) -> Right(ConflictingInherited(collisions))
let append_refines data = match build_class_refinement_map data with
    | Left(data) -> Left(data)
    | Right(collisions) -> Right(ConflictingRefinements(collisions))
let test_signatures data = match type_check_signatures data with
    | Left(data) -> Left(data)
    | Right(bad) -> Right(PoorlyTypedSigs(bad))
let append_refinable data = Left(build_refinable_map data)
let append_mains data = match build_main_map data with
    | Left(data) -> Left(data)
    | Right(collisions) -> Right(MultipleMains(collisions))

let test_list =
    [ append_children ; append_parent ; test_tree ; append_ancestor ;
      append_distance ; append_variables ; test_fields ; test_types ;
      append_methods ; test_init ; test_inherited_methods ; append_refines ;
      test_signatures ; append_refinable ; append_mains ]

let production_list =
    [ append_children ; append_parent ; test_tree ; append_ancestor ;
      append_distance ; append_variables ; test_fields ; append_methods ;
      test_init ; append_refines ; append_mains ]

let build_class_data klasses = seq (initial_data klasses)
```

```
    test_list (*production_list*)
let build_class_data_test klasses = seq (initial_data klasses) test_list

let append_leaf_known aklass data =
    let updated = StringSet.add aklass.klass data.known in
    if StringSet.mem aklass.klass data.known
        then Right(DuplicateClasses([aklass.klass]))
        else Left({ data with known = updated })
let append_leaf_classes aklass data =
    let updated = StringMap.add aklass.klass aklass data.classes in
    Left({ data with classes = updated })
let append_leaf_tree aklass data =
    (* If we assume data is valid and data has aklass's parent then we should be fine *)
    let parent = klass_to_parent aklass in
    if StringMap.mem parent data.classes
        then Left(data)
        else Right(HierarchyIssue("Appending a leaf without a known parent."))
let append_leaf_children aklass data =
    let parent = klass_to_parent aklass in
    let updated = add_map_list parent aklass.klass data.children in
    Left({ data with children = updated })
let append_leaf_parent aklass data =
    let parent = klass_to_parent aklass in
    let updated = StringMap.add aklass.klass parent data.parents in
    Left({ data with parents = updated })
let append_leaf_variables aklass data = match build_var_map aklass with
    | Left(vars) ->
        let updated = StringMap.add aklass.klass vars data.variables in
        Left({ data with variables = updated })
    | Right(collisions) -> Right(DuplicateVariables([(aklass.klass, collisions)]))
let append_leaf_test_fields aklass data =
    let folder collisions var = match class_field_lookup data (klass_to_parent aklass) var with
        | Some((_, _, Privates)) -> collisions
        | Some((ancestor, _, section)) -> (ancestor, var)::collisions
        | _ -> collisions in
    let variables = List.flatten (List.map snd (klass_to_variables aklass)) in
    let varnames = List.map snd variables in
    match List.fold_left folder [] varnames with
        | [] -> Left(data)
        | collisions -> Right(DuplicateFields([(aklass.klass, collisions)]))
let append_leaf_type_vars aklass data =
    match type_check_variables data aklass with
        | [] -> Left(data)
        | bad -> Right(UnknownTypes([(aklass.klass, bad)]))
```

```
let append_leaf_methods aklass data = match build_method_map aklass with
    | Left(meths) ->
        let updated = StringMap.add aklass.klass meths data.methods in
        Left({ data with methods = updated })
    | Right(collisions) -> Right(ConflictingMethods([build_collisions aklass.klass collisions false]))
let append_leaf_test_inherited aklass data =
    let folder collisions meth = match class_ancestor_method_lookup data aklass.klass meth.name true with
        | [] -> collisions
        | funcs -> match List.filter (conflicting_signatures meth) funcs with
            | [] -> collisions
            | cols -> cols in
    let skipinit (func : Ast.func_def) = match func.name with
        | "init" -> false
        | _ -> true in
    let functions = List.flatten (List.map snd (klass_to_methods aklass)) in
    let noninits = List.filter skipinit functions in
    match List.fold_left folder [] noninits with
        | [] -> Left(data)
        | collisions -> Right(ConflictingInherited([build_collisions aklass.klass collisions false]))
let append_leaf_instantiable aklass data =
    let is_init mem = match mem with
        | InitMem(_) -> true
        | _ -> false in
    if List.exists is_init (aklass.sections.protects) then Left(data)
    else if List.exists is_init (aklass.sections.publics) then Left(data)
    else Right(Uninstantiable([aklass.klass]))
let append_leaf_refines aklass data = match build_refinement_map aklass with
    | Left(refs) ->
        let updated = StringMap.add aklass.klass refs data.refines in
        Left({ data with refines = updated })
    | Right(collisions) -> Right(ConflictingRefinements([build_collisions aklass.klass collisions true]))
let append_leaf_mains aklass data = match aklass.sections.mains with
    | [] -> Left(data)
    | [main] ->
        let updated = StringMap.add aklass.klass main data.mains in
        Left({ data with mains = updated })
    | _ -> Right(MultipleMains([aklass.klass]))
let append_leaf_signatures aklass data = match type_check_class data aklass with
    | Left(data) -> Left(data)
    | Right(bad) -> Right(PoorlyTypedSigs([bad]))
let append_leaf_ancestor aklass data =
```

```
    let parent = klass_to_parent aklass in
    let ancestors = aklass.klass::(StringMap.find parent data.ancestors) in
    let updated = StringMap.add aklass.klass ancestors data.ancestors in
    Left({ data with ancestors = updated })
let append_leaf_distance aklass data =
    let ancestors = StringMap.find aklass.klass data.ancestors in
    let distance = build_distance aklass.klass ancestors in
    let updated = StringMap.add aklass.klass distance data.distance in
    Left({ data with distance = updated })
let append_leaf_refinable aklass data =
    let parent = klass_to_parent aklass in
    let updated = update_refinable parent aklass.sections.refines data.refinable in
    Left({ data with refinable = updated })

let production_leaf =
    [ append_leaf_known ; append_leaf_classes ; append_leaf_children ; append_leaf_parent ;
      append_leaf_ancestor ; append_leaf_distance ; append_leaf_variables ; append_leaf_test_fields ;
      append_leaf_methods ; append_leaf_instantiable ; append_leaf_refines ; append_leaf_signatures ;
      append_leaf_mains ]
let test_leaf =
    [ append_leaf_known ; append_leaf_classes ; append_leaf_children ; append_leaf_parent ;
      append_leaf_ancestor ; append_leaf_distance ; append_leaf_variables ; append_leaf_test_fields ;
      append_leaf_type_vars ; append_leaf_methods ; append_leaf_instantiable ; append_leaf_test_inherited ;
      append_leaf_refines ; append_leaf_refinable ; append_leaf_mains ]

let leaf_with_klass actions data klass = seq (Left(data)) (List.map (fun f -> f klass) actions)
let append_leaf = leaf_with_klass test_leaf (* production_leaf *)
let append_leaf_test = leaf_with_klass test_leaf

let append_leaf_test data aklass =
    let with_klass f = f aklass in
    let actions =
        [ append_leaf_known ; append_leaf_classes ; append_leaf_children ; append_leaf_parent ;
          append_leaf_ancestor ; append_leaf_distance ; append_leaf_variables ; append_leaf_test_fields ;
          append_leaf_type_vars ; append_leaf_methods ; append_leaf_instantiable ; append_leaf_test_inherited ;
          append_leaf_refines ; append_leaf_refinable ; append_leaf_mains ] in
    seq (Left(data)) (List.map with_klass actions)

(**
```

```
    Print class data out to stdout.
  *)
let print_class_data data =
    let id x = x in
    let from_list lst = Format.sprintf "[%s]" (String.concat ", " lst) in
    let table_printer tbl name stringer =
        let printer p s i = Format.sprintf "\t%s : %s => %s\n" p s (stringer i) in
        print_string (name ^ ":\n");
        print_lookup_table tbl printer in
    let map_printer map name stringer =
        let printer k i = Format.sprintf "\t%s => %s\n" k (stringer i) in
        print_string (name ^ ":\n");
        print_lookup_map map printer in

    let func_list = function
        | [one] -> full_signature_string one
        | list -> let sigs = List.map (fun f -> "\n\t\t" ^ (full_signature_string f)) list in
            String.concat "" sigs in

    let func_of_list funcs =
        let sigs = List.map (fun f -> "\n\t\t" ^ f.inklass ^ "-> " ^ (full_signature_string f)) funcs in
        String.concat "" sigs in

    let class_printer cdef =
        let rec count sect = function
            | (where, members)::_ when where = sect -> List.length members
            | _::rest -> count sect rest
            | [] -> raise(Failure("The impossible happened -- searching for a section that should exist doesn't exist.")) in
        let vars = klass_to_variables cdef in
        let funcs = klass_to_functions cdef in
        let format = ""^^"from %s: M(%d/%d/%d) F(%d/%d/%d) R(%d) M(%d)" in
        let parent = match cdef.klass with
            | "Object" -> "----"
            | _ -> klass_to_parent cdef in
        Format.sprintf format parent
            (count Privates funcs) (count Protects funcs) (count Publics funcs)
            (count Privates vars) (count Protects vars) (count Publics vars)
            (count Refines funcs) (count Mains funcs) in

    let print_list list =
        let rec list_printer spaces endl space = function
            | [] -> if endl then () else print_newline ()
            | list when spaces = 0 -> print_string "\t"; list_printer 8 false false list
            | list when spaces > 60 -> print_newline (); list_printer 0 true false list
```

```
              | item::rest ->
                if space then print_string " " else ();
                print_string item;
                list_printer (spaces + String.length item) false true rest in
          list_printer 0 true false list in

    Printf.printf "Types:\n";
    print_list (StringSet.elements data.known);
    print_newline ();
    map_printer data.classes "Classes" class_printer;
    print_newline ();
    map_printer data.parents "Parents" id;
    print_newline ();
    map_printer data.children "Children" from_list;
    print_newline ();
    map_printer data.ancestors "Ancestors" from_list;
    print_newline ();
    table_printer data.distance "Distance" string_of_int;
    print_newline ();
    table_printer data.variables "Variables" (fun (sect, t) -> Format.sprintf "%s %s" (section_string sect) t);
    print_newline ();
    table_printer data.methods "Methods" func_list;
    print_newline ();
    table_printer data.refines "Refines" func_list;
    print_newline ();
    map_printer data.mains "Mains" full_signature_string;
    print_newline ();
    table_printer data.refinable "Refinable" func_of_list


(* ERROR HANDLING *)

let args lst = Format.sprintf "(%s)" (String.concat ", " lst)
let asig (name, formals) = Format.sprintf "%s %s" name (args formals)
let aref (name, formals) = asig (name, formals)

let dupvar (klass, vars) = match vars with
    | [var] -> "Class " ^ klass ^ "'s instance variable " ^ var ^ " is multiply declared"
    | _ -> "Class " ^ klass ^ " has multiply declared variables: [" ^ (String.concat ", " vars) ^ "]"

let dupfield (klass, fields) = match fields with
    | [(ancestor, var)] -> "Class " ^ klass ^ "'s instance variable " ^ var ^ " was declared in ancestor " ^ ancestor ^ "."
    | _ -> "Class " ^ klass ^ " has instance variables declared in ancestors: [" ^ String.concat ", " (List.map (fun (a, v) -> v ^ " in " ^ a) fields) ^ "]"

let show_vdecls vs = "[" ^ String.concat ", " (List.map (fun (t, v) -> t ^ ":" ^ v) vs) ^ "]"

let unknowntypes (klass, types) = match types with
```

```
    | [(vtype, vname)] -> "Class " ^ klass ^ "'s instancevariable " ^ vname ^ " has unknown type " ^ vtype ^ "."
    | _ -> "Class " ^ klass ^ " has instance variables with unknown types: " ^ show_vdecls types

let badsig1 klass (func, ret, params) = match ret, params with
    | None, params -> "Class " ^ klass ^ "'s " ^ func ^ " has poorly typed parameters: " ^ show_vdecls params
    | Some(rval), [] -> "Class " ^ klass ^ "'s " ^ func ^ " has an invalid return type: " ^ rval ^ "."
    | Some(rval), p -> "Class " ^ klass ^ "'s " ^ func ^ " has invalid return type " ^ rval ^ " and poorly typed parameters: " ^ show_vdecls p
let badsig (klass, badfuncs) = String.concat "\n" (List.map (badsig1 klass) badfuncs)

let dupmeth (klass, meths) =
    match meths with
        | [(name, formals)] -> Format.sprintf "Class %s's method %s has multiple implementations taking %s" klass name (args formals)
        | _ -> Format.sprintf "Class %s has multiple methods with conflicting signatures:\n\t%s" klass (String.concat "\n\t" (List.map asig meths))

let dupinherit (klass, meths) =
    match meths with
        | [(name, formals)] -> Format.sprintf "Class %s's method %s has conflicts with an inherited method taking %s" klass name (args formals)
        | _ -> Format.sprintf "Class %s has multiple methods with conflicting with inherited methods:\n\t%s" klass (String.concat "\n\t" (List.map asig meths))

let dupref (klass, refines) =
    match refines with
        | [refine] -> Format.sprintf "Class %s refinment %s is multiply defined." klass (aref refine)
        | _ -> Format.sprintf "Class %s has multiple refinements multiply defined:\n\t%s" klass (String.concat "\n\t" (List.map aref refines))

let errstr = function
    | HierarchyIssue(s) -> s
    | DuplicateClasses(klasses) -> (match klasses with
        | [klass] -> "Multiple classes named " ^ klass
        | _ -> "Multiple classes share the names [" ^ (String.concat ", " klasses) ^ "]")
    | DuplicateVariables(list) -> String.concat "\n" (List.map dupvar list)
    | DuplicateFields(list) -> String.concat "\n" (List.map dupfield list)
    | UnknownTypes(types) -> String.concat "\n" (List.map unknowntypes types)
    | ConflictingMethods(list) -> String.concat "\n" (List.map dupmeth list)
```

```
    | ConflictingInherited(list) -> String.concat "\n" (List.map dupinherit list)
    | PoorlyTypedSigs(list) -> String.concat "\n" (List.map badsig list)
    | Uninstantiable(klasses) -> (match klasses with
        | [klass] -> "Class " ^ klass ^ " does not have a usable init."
        | _ -> "Multiple classes are not instantiable: [" ^ String.concat ", " klasses ^ "]")
    | ConflictingRefinements(list) -> String.concat "\n" (List.map dupref list)
    | MultipleMains(klasses) -> (match klasses with
        | [klass] -> "Class " ^ klass ^ " has multiple mains defined."
        | _ -> "Multiple classes have more than one main: [" ^ String.concat ", " klasses ^ "]")
```

Source 129: `KlassData.ml`