FOGL (Figure Oriented Graphics Language)
Julian Rosenblum, Evgenia Nitishinskaya, Richard Zou

1 - Introduction

FOGL is a language for the sole purpose of creating static and interactive graphics. As the name suggests, programs are primarily made up of specialized data structures called figures. Figures are similar to classes except they are more tailored towards graphics. FOGL is statically typed with the minor exception of generic figure types (see 3.2.8.1). FOGL source code is translated into Java source code which is then compiled along with Java boilerplate files into an executable .jar.

A FOGL program consists of a list of statements. Any statement that is not a figure definition is considered to be part of an implicit main function. All figure instances declared in the main function are implicitly drawn (see 3.1.9). The program then loops. During a cycle, each figure's _update function is called if and only if its _animating property is set to true (see 3.1.6, 3.1.8). If any figure is mutated during a cycle, all figures are erased and redrawn.

2 - Language Tutorial

2.1 - Hello World! #1

A basic Hello World program can be written in FOGL as follows:

```
print("Hello World!")
```

This calls the built-in print function, which takes a single string argument and prints it to the console, returning no value. This line of code alone is sufficient to be a FOGL program. However, it is not very interesting.

2.2 - Hello World #2

The following is a slightly more interesting Hello World program, utilizing graphics and animation:

```
figure Hello {
	param loc = <<0,0>>, col = color
	comp text = Text(text:"Hello World!" loc:loc col:col)
}
Hello(col:#00f)
Hello(col:#f00@.5 loc:<<100,100>>)
```

This code defines a "Hello" figure which takes location and color parameters, and draws the text hello world to the screen. Location is an optional parameter, since it has a default value. Two instances of the code are then drawn: one which is blue and at the default location of <<0,0>>, and one which is red (.5 opacity) at the location <<100,100>>

2.3 - Compiling a Program

FOGL programs can be compiled using the shell command `path/to/fogl -c filename.fogl`. The compiler then outputs a jar called filename.jar in the same directory, which can be executed with the command `java -jar filename.jar`.

3 - Language Reference Manual

3.1 - Figures

A figure is a class-like definition of an object that can be drawn from its components given parameters. Like classes, figures can have multiple instances. Figures can contain any number of any of the following (collectively known as properties) within them:

3.1.1 - Parameters

```
param rad = int,
loc = <<0,0>>,
col = #ff0
```

Parameters are the given arguments that are also treated as public properties. They are bound by reference. If a parameter is not given a default value, then it must be declared for each instance of a figure; otherwise, it is optional (see 3.1.9). Parameters eliminate most of the need for constructors. For example, the code:

```
figure PacMan { param loc = pair }
```

is essentially equivalent to the following in Java:

```
public class PacMan {
	private Location loc;
	public PacMan (Location myLoc) {
		loc = myLoc;
	}
}
```

Constructors, in the rare event that they are necessary, can be simulated using the _update function (see 3.1.6).

3.1.2 - Components

```
comp body = Arc(loc:loc radius:rad col:col),
mouth = Arc(loc:loc radius:rad start:PI/4 end:-PI/4 col:#000 fill:true),
eye = Arc(loc:loc+<<rad/2,-rad/2>> radius:3 col=#000)
```

Components are properties that are drawn whenever the figure is drawn. A component must be a figure (built-in or user-defined). Components are bound to figure references and cannot be assigned directly.

3.1.3 - Vars

```
var mouthOpenness = 1,
isAlive = true
```

Vars are private and can only be accessed by instances of the figure they are defined within.

3.1.4 - Methods

```
var doSomething = func (arg1:int, arg2:bool) {
	; statements
}
```

Methods are vars (or rarely params) that are assigned to functions.

3.1.5 - Recursive Figures

Figures can be recursive, meaning they have components that are instances of themselves. Recursive figures must have at least one component that isn't recursive. The base case is considered to be when all non-recursive the drawn components of the figure combined are smaller than a single pixel. For example, consider the Sierpinski triangle (assume the figure `Equilateral` has already been defined with loc referring to the top vertex):

```
figure Sierpinski {
	param loc = <<0,0>>, sideLength = int
	comp triangle = Equilateral(loc:loc length:sideLength fill:false),
	s1 = Sierpinski(loc:loc sideLength:sideLength/2),
	s2 = Sierpinski(loc:loc+<<-sideLength/4,sideLength/4*RT_3>>
sideLength:sideLength/2),
	s3 = Sierpinski(loc:loc+<<sideLength/4,sideLength/4*RT_3>>
sideLength:sideLength/2)
}
var s = Sierpinski(sideLength:5)
```

3.1.6 - The _update Method

Figures may have a var called _update which is used by the compiler to perform animations. With each cycle, every figure's _update method is called (if such a method exists) with one argument: the current number of milliseconds since the program began. This function can be used to perform animations by modifying the figure based on the milliseconds parameter. The function does not return anything. The following example smoothly opens and closes PacMan's mouth where cycleLength is the number of milliseconds it takes to go from open to closed:

```
var _update = func (ms:int) {
	var state = (cycleLength * 2) % ms
	if state > cycleLength { ; If the mouth is closing
		state = cycleLength * 2 - state
	}
	mouth.start = (state / cycleLength) * PI/4
	mouth.end = -mouth.start
	if ms == 3000 { ; additionally, at 3 seconds
		body.col = #00f ; change to blue
	}
}
```

3.1.7 - The _visible Property

Every figure has an implicit `_visible` property. It is set to true by default. If explicitly set to false, the figure's components are not drawn. `_visible` is a param and can be bound in a figure initialization.

3.1.8 - The _animating Property

The `_animating` property is a param that is implicitly set to true if an `_update` function is defined and false otherwise. The `_update` function is only called if the `_animating` property is true, and if the `_animating` property is true and no `_update` function is defined, the program throws an error.

3.1.9 - Initializing Figures

Instances of figures are initialized using the syntax: `PacMan(rad:4 loc:<<6,6>>)`. Within the parentheses is a list of whitespace/newline separated parameter bindings, with the name of the parameter (as defined by the figure) on the left of a colon and an expression on the right. Parameters are bound to the *value* of the expression. All parameters of the figure that do not have default values *must* be bound for each instance of the figure (otherwise an error is thrown). Parameters that do have default values are not required but may be bound for any figure instance. "Empty" figure initializations, if all parameters have default values, are initialized using the syntax: `MyFigure(:)`. The colon allows the program to distinguish between figure initializations and function

calls at compile time.

3.2 - Data Types

3.2.1 - Int

An int is a 32-bit signed integer. An int literal is formally defined as 1 or more digits. Ints can be assigned to float types, but not the other way around.

3.2.2 - Float

A float is a 64-bit signed floating-point number. A float literal is formally defined as `((digit+ '.') | digit* '.' digit+) exp? | digit+ exp` where exp is defined as `['e' 'E'] ['+' '-']? digit+`.

3.2.3 - Boolean

A boolean value defined using the keywords `true` and `false`.

3.2.4 - String

A string is a string of text. String literals are placed between either single or double quotes.

3.2.5 - Ordered Pair

An ordered pair is a pair of x and y floats which can be expressed as a literal `<<x,y>>`. Note: the origin is the top left corner of the canvas and the y-value increases in a downward direction. Arithmetic operators may be used on two ordered pairs or one ordered pair and one integer or float. Assuming *op* is a binary operator, these operations are defined as follows: `<<x,y>>` *op* `<<z,w>>` evaluates to `<<x` *op* `z,y` *op* `w>>`. `<<x,y>>` *op* `n` is equal to `<<x` *op* `n,y` *op* `n>>`. `n` *op* `<<x,y>>` evaluates to `<<n` *op* `x,n` *op* `y>>`. The individual x and y values can be retrieved using the built-in `x()` and `y()` functions (see 3.5.2.3).

3.2.6 - List

A list is a Linked List which can be expressed as comma-separated items inside brackets. Lists can only contains elements of the same type. A list cannot contain two different types of figures.

3.2.7 - Color

A color is a hex colorvalue which can be expressed as the literal `#rrggbb`, `#rgb`, `#rrggbb@a`, or `#rgb@a` where `r`, `g`, and `b` are hex digits and `a` is a float literal representing alpha (opacity) on a decimal scale from 0 to 1. Note: if single digit hex values are used, the digits are duplicated to produce double-digit values. For example, `#f03@.5` evaluates as `#ff0033@.5`. If alpha is not declared, it is assumed to be 1.0.

3.2.8 - Figure

An instance of a figure as defined in section 1.

3.2.8.1 - Generic Figure

Generic figures can be declared, in which case the specific figure type is unknown at compile-time. The `_visible` and `_animating` properties can be accessed for generic figures, while specific properties cannot.

3.2.9 - Function

```
var doSomething = func:bool (arg1:int, arg2:bool) {
	; statements
}
```

All functions are lambdas assigned to variables. Arguments in method must be declared with types and function calls are referenced by numerical order, not by name. The return type is specified after a colon after the `func` keyword. For void functions, the return type is left out.

A simple function expression whose body consists only of a return statement can be written with a shorthand. The following two expressions are equivalent:

```
func[...] (...) (x + y)

func[...] (...) { return x + y }
```

Note: parentheses around the return expression will almost always be required due to operator precedence. Without the parentheses, the first expression is interpreted as `(func[...] (...) x) + y`.

3.2.10 - Type System

All variables, properties, components, and parameters must be declared with either a value or a type, for example `var foo = 5` or `var foo = int`. If a variable is declared with a default value, its type is inferred by the compiler. Function arguments must also be given a type, for example `foo(bar:int, baz:color)`. If an int is assigned to a float type, it is automatically cast. The following are valid type declarations:

- `int`
- `float`
- `string`
- `bool`
- `color`
- `pair`
- `id{} ; a figure named id`
- `figure{} ; generic figure`
- `t[] ; a list of type t`
- `tr(t0, t1, t2) ; a function where tr is the return type (left out if function doesn't return a value), and t0,...,tn are the types of the arguments in order.`

3.2.11 - Type Casting

Integers are automatically cast to floats when necessary. If an arithmetic operator is used on two integers, the result is an integer. Otherwise, the result is a float. The only exceptions are the `/` and `^` operators, in which case the result is always a float, and the `//` operator, in which case the result is always an integer. In instances when explicit type casting is necessary, there are built-in functions that take convert an argument to the indicated type if such a conversion is possible and throw an error otherwise (see 3.5.2.1).

A generic figure can be cast to a specific figure type using the syntax `name(fig)` where name is the type of figure (e.g., Rect) and fig is of a generic figure type. Specific figures can be assigned to values of generic figure types, but are then treated as generic. Generic figures can be assigned to specific figure types, in which case they are implicitly cast. The specific type of a generic figure can be determined using the `type()` function (see 3.5.2.1)

3.3 - Operators

3.3.1 - Arithmetic Operators

- + (addition, string concatenation, list concatenation)
- - (subtraction, unary negation)

- * (multiplication)
- / (floating-point division, both operands cast to float)
- // (integer division, both operands cast to int)
- % (modulo)
- ^ (exponentiation)

3.3.2 - Assignment Operators

- = (assigns RHS to LHS or passes RHS to argument with the identifier LHS)
- An arithmetic operator followed by = performs the operation with LHS and RHS and then assigns the value to LHS.

3.3.2 - Comparison Operators

- == (equal to)
- != (not equal to)
- > (greater than)
- < (less than)
- >= (greater than or equal to)
- <= (less than or equal to)

3.3.3 - Logical Operators

- ! (unary, logical not, casts operand to boolean)
- & (logical and, casts operands to boolean)
- | (logical or, casts operands to boolean)
- ? (unary, casts operand to boolean)

3.3.4 - Misc. Operators

- . (dot operator) - Selects property or method of a figure.
- : (param-bind operator) - Binds a parameter (lhs) to value (rhs)
- : (argument-type operator) - Binds an argument (lhs) to a type (rhs)
- [n] (list indexing operator) - Returns the nth element of the list (0-indexed)

3.4 - Statements, Comments, etc.

Statements are terminated with newlines.

3.4.1 - Reserved words

```
bool color comp do else elseif false figure float func if int pair param
return string true var while
```

3.4.2 - Comments

Single-line comments are declared using a `;` (semicolon) and terminate at the end of a line.

3.4.3 - If-elseif-else Statements

An if statement contains an expression and a block. The block of an if statement is executed if and only if the expression evaluates to true. An else statement has only a block, which is executed if the nearest if statement's expression evaluates to true. Elseif statements are syntactic sugar for easily writing nested if-else statements, such that the following:

```
if x < 3 {
        ; statements1
```

```
}
elseif x > 7 {
	; statements2
}
else {
	; statements3
}
```

is semantically equivalent to:

```
if x < 3 {
	; statements1
}
else {
	if x > 7 {
		; statements2
	}
	else {
		; statements3
	}
}
```

3.4.4 - While Statements

A while statement contains an expression and a block. If the expression evaluates to true, the body of the while loop is executed and the expression is evaluated again, repeating the cycle.

```
while i < 5 {
	; statements
	i += 1
}
```

3.4.5 - Do-while Statements

A do-while statement contains a block and an expression. The block is executed and then the expression is evaluated. If the expression evaluates to true, the block is executed again, repeating the cycle.

```
do {
	; statements
	i += 1
} while i < 5
```

3.4.6 - Global Variables

```
var foo = 5, bar = "hello"
```

Global variables can be accessed within any scope.

3.5 - Standard Library

3.5.1 - Built-in figures

- `figure Rect { param loc=<<0,0>>, width=int, height=int, col=#000, fill=true, lineWidth=1 }`
- `figure Arc { param loc=<<0,0>>, radius=int, start=0, end=2*PI, col=#000, fill=true, lineWidth=1, clockwise=true }`

- `figure Poly { param loc=<<0,0>>, points=pair[], col=#000, fill=true, lineWidth=1 }` Note: points are relative to loc
- `figure Text { param loc=<<0,0>>, text=string, col=#000, size=12, font="Arial" }`
  Note: lineWidth only applies if fill=false

3.5.2 - Built-in functions

- **3.5.2.1 - General**
  - `print(str:string)` - Prints a string to the console.
  - `itof(foo:int)`, `ftoi(foo:float)`, `itos(foo:int)`, `ftos(foo:float)`, `stoi(foo:string)`, `stof(foo:string)` - Performs type conversions.
  - `type(fig:figure{})` - Returns type (string) of generic figure.
- **3.5.2.2 - Color Operations**
  - `rgb(r:int, g:int, b:int)` - Returns a color literal given rgba values.
  - `rgba(r:int, g:int, b:int, a:float)` - Returns a color literal given rgba values.
- **3.5.2.3 - Pair Operations**
  - `x(p:pair)`, `y(p:pair)` - Returns the x or y value of an ordered pair.
  - `dist(p1:pair, p2:pair)` - Returns the Euclidian distance between two pairs.
- **3.5.2.4 - Vector Operations**
  - `mag(p:pair)` - Returns the magnitude of the vector from the origin to pair.
  - `ang(p:pair)` - Returns the angle (in radians) of the vector from the origin to pair. An error is thrown if `pair` is `<<0,0>>`.
  - `dot(p1:pair, p2:pair)` - Returns the dot product between two vectors v1 and v2, where v1 is a vector formed from the origin to `p1`, and v2 is a vector formed from the origin to `p2`.
  - `cross(p1:pair, p2:pair)` - Returns the magnitude of the cross product between two vectors v1 and v2, where v1 is a vector formed from the origin to `p1`, and v2 is a vector from the origin to `p2`.
  - `bet(pair1, pair2)` - Returns the angle (in radians) between two vectors v1 and v2, where v1 is a vector formed from the origin to `p1`, and v2 is a vector from the origin to `p2`. An error is thrown either ordered pair is `<<0,0>>`.
- **3.5.2.5 - Mathematical Operations**
  - `toDeg(theta:float)` - Converts radians to degrees.
  - `toRad(theta:float)` - Converts degrees to radians.
  - `sin(x:float)`, `cos(x:float)`, `tan(x:f:float)` - where `x` is in radians
  - `asin(x:float)`, `acos(x:float)`, `atan(x:float)` - returns the angle as a radian value
  - `log(b:float, e:float)` - where `e`, $b > 0$, $b \neq 1$
  - `ln(x:float)` - where $x > 0$
  - `ceil(x:float)`, `floor(x:float)`
  - `sqrt(x:float)`
  - `round(x:float)`
  - `min(x:float, y:float)`, `max(x:float, y:float)`
  - `abs(x:float)`
  - `random()`

3.5.3 - Built-in constants

- `PI = float`
- `E = float`

- `RT_2 = float` (Square root of 2)
- `RT_3 = float` (Square root of 3)

3.6 - Syntax and Parsing

3.6.1 - A Note on Whitespace

Since newlines are semantically significant in FOGL, they are handled by a combination of the scanner and parser. A newline is formally defined as `([' ' '\r' '\t']* '\n' | [' ' '\r' '\t']* ';' [^ '\n'] * '\n')+`, meaning a newline character or single-line comment followed by any combination of empty lines or lines consisting of only single-line comments. Grouping punctuation (parentheses, brackets, braces, double angle brackets) are considered to "absorb" newlines at the beginning and end of the group. Opening punctuation is defined as the punctuation followed by 0 or more newlines. Closing punctuation is defined as 0 or more newlines followed by the punctuation. Additionally, the keywords `else` and `elseif` are defined as 0 or more newlines followed by the keyword. For additional instances of optional whitespace, the `ws` grammar rule is used, which is defined as 0 or more newlines.

3.6.2 - Grammar

```
program:
   program_contents
program_contents:
   ws
   program_contents stmt NL
   program_contents stmt EOF
ws:
   /* nothing */
   NL
figure:
   FIGURE ID ws { figure_body figure_decl ws }
   FIGURE ID ws { ws }
main:
   /* nothing */
   main stmt NL
   main stmt EOF
block:
   { }
   { stmt_list stmt ws }
type:
   T_INT
   T_FLOAT
   T_STRING
   T_BOOL
   T_COLOR
   T_PAIR
   ID { }
   T_FIGURE { }
   type [ ]
   type ( type_list_opt )
   ( type_list_opt )
```

```
type_list:
   type
   type_list COMMA ws type
type_list_opt:
   /* nothing */
   type_list
stmt_list:
   /* nothing */
   stmt_list stmt NL
figure_decl:
   PARAM decl_list
   COMP decl_list
   VAR decl_list
figure_body:
   /* nothing */
   figure_body figure_decl NL
decl_list:
   decl
   decl_list COMMA ws decl
decl:
   ID = type
   ID = expr
arg_list:
   ID : type
   arg_list COMMA ws ID : type
arg_list_opt:
   /* nothing */
   arg_list
expr_list:
   expr
   expr_list COMMA ws expr
expr_list_opt:
   /* nothing */
   expr_list
param:
   ID : expr
param_list:
   param
   param_list ws param
param_list_opt:
   :
   param_list
elseif:
   /* nothing */
   ELSE ws block
   ELSEIF expr ws block elseif
stmt:
   expr
```

```
   figure
   VAR decl_list
   expr = expr
   expr += expr
   expr -= expr
   expr *= expr
   expr /= expr
   expr //= expr
   expr ^= expr
   expr %= expr
   RETURN expr
   RETURN
   WHILE expr ws block
   DO ws block ws WHILE expr
   IF expr ws block elseif
expr:
   ( expr )
   INT
   STRING
   FLOAT
   TRUE
   FALSE
   COLOR
   << expr COMMA ws expr >>
   ID
   FUNC ( arg_list_opt ) block
   FUNC [ type ] ( arg_list_opt ) block
   FUNC [ type ] ( arg_list_opt ) expr
   [ expr_list_opt ]
   expr [ expr ]
   expr . ID
   expr + expr
   expr - expr
   expr * expr
   expr / expr
   expr // expr
   expr ^ expr
   expr % expr
   expr == expr
   expr != expr
   expr > expr
   expr < expr
   expr >= expr
   expr <= expr
   expr & expr
   expr | expr
   ! expr
   ? expr
```

```
   - expr
   expr ( expr_list_opt )
   func
   ID ( param_list_opt )
func:
    ID ( expr_list_opt )
    expr . ID ( expr_list_opt )
    func ( expr_list_opt )
    ( expr ) ( expr_list_opt )
```

4 - Project Plan

4.1 - Planning, Specification, Development, Testing

The project plan started with a basic concept and then got more and more specific over time. The original concept was some sort of graphics language. The concept of figures came fairly early as a non-class-based way to handle graphics. This also came with the concept of components, parameters, and vars. The animation functionality went through a variety of drafts, going from draw, update, and wait functions to a standard update cycle with update functions for different figures that take a time parameter. The language was originally dynamically typed, until we realized that the nature of figures made the language more suited as a statically-typed language, with the small exception of generic figure types.

Once the basic ideas and semantics of the language were established, the more nit-picky elements came fairly easily. Then, we started on the scanner, parser, and AST. The LRM was written in conjunction with these. Few changes were made to the scanner, parser, and AST were made after their initial completion. At that point, we decided to split the project up into analysis and code generation. Development for the most part went in the same order that the code follows through the compilation process. Scanning, parsing, and the AST were done at the same time, followed by analysis, followed by code generation. It was not necessarily optimal, but it's how the schedule worked out time-wise. We tried to write as many test cases as we could along the way, but many of them wound up being written at the end.

4.2 - Style Guide

In general, we didn't set very many strict stylistic guidelines, but we did agree on handful of conventions for the purpose of organization. First off, we tried to keep all lines of code fewer than 80 characters. This was more of a guideline than a rule since we did not always follow it, but keeping it in mind helped lead to cleaner code. We also put whitespace between function definitions for cleanliness. Similarly, we agreed to put comments before function definitions stating what the function does.

We decided to centralize most data structures in the ast.ml file for the sake of managing dependencies, since ast.ml doesn't depend on any other files. We elected to have a strings.ml file for pretty printing, which was helpful both in debugging the compiler and in creating error messages. Also, to the extent that we could, we tried to keep only one person working on each file, for organization's sake.

4.3 - Project Timeline

- September 21- Begin project proposal
- October 6 - Begin LRM
- October 9 - Begin scanner, parser, AST
- November 11 - Begin analyzer
- December 10 - Begin translator

4.4 - Roles and Responsibilities

4.4.1 - Julian
Project leader. Break ties in the decision making process, divide tasks, send angry emails about deadlines, write the scanner/parser/ast, write the analyzer, write print test cases.

4.4.2 - Richard
Handle core Java generation. Implement basic language elements, captures, bindings.

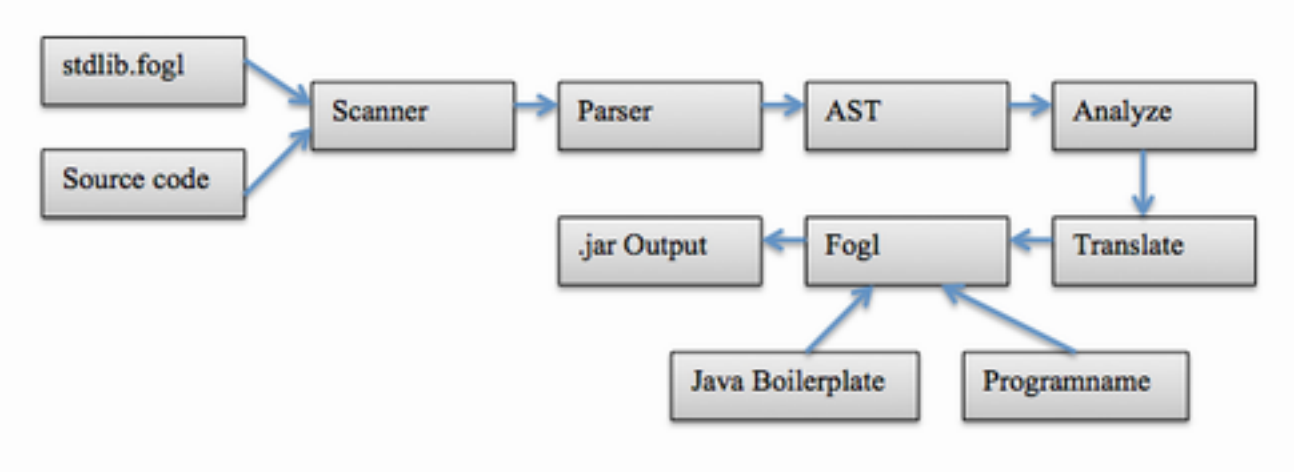
4.4.3 - Jenny
Handle all Java generation relating to graphics and animation. Write graphical test cases.

4.5 - Software Development Environment

The majority of the code was written using Ocaml and the Ocamllex and Ocamlyacc libraries. There are also Java boilerplate files that are pre-written and used by the outputted Java. There is also a .fogl file with standard library information that is inserted before the user's FOGL code in compilation. Python was used for test automation scripting. We used Subversion through Sourcefourge for version control. Julian used Sublime Text 2 as his main editor.

5 - Architectural Design

5.1 - Block Diagram



5.2 - Modules

5.2.1 - Scanner (Julian)
Takes stream of characters and outputs stream of tokens. Depends on: Parser.

5.2.2 - Parser (Julian)
Takes stream of tokens and returns AST. Also generates indices for each scope (this requires using imperative features and the actual functions are defined in ast.ml). Depends on: AST.

5.2.3 - AST (Julian)
Defines the AST types as well as almost all types that are used by the compiler. Also includes a counter that is used by the Parser. Depends on: nothing.

5.2.4 - Strings (Everyone)
Functions for pretty printing. These are mostly used for error messages and compiler debugging. Depends on: AST.

5.2.5 - Analyze (Julian)
Takes the AST, checks/infers types, and returns a list of scopes. Each scope type contains the index of its parent scope, a map of identifiers to their types within the scope, and various information about specific to scopes of functions and figures. Depends on: AST, Strings.
Analyze.ml will give an error for the following things.

- Trying to access an undeclared identifier.
- Assigning a value of one type to a variable of a different type.
- Declaring an identifier already declared in the current scope.
- Performing an operator on non-compatible types. (The analyzer defines which operators can operate on which combinations of types and what the type of the result is)
- A function returning something of a different type than the type returned by the first return statement.
- Mixing types in a list.
- Declaring a variable with an empty list (type can't be inferred)
- Using non-int/float types in a pair.
- Trying to subscript a non-list.
- Using the dot operator on a non-figure.
- Using the wrong argument types in a function call.
- Trying to call a non-function.
- Leaving out required parameters in a figure initialization.
- Trying to assign something that is not assignable.
- Trying to assign a component directly.
- Declaring a component with a non-figure type.
- Declaring _update, _visible, _animating with the wrong types or with the wrong declarations.

5.2.6 - stdlib.fogl (Everyone)
stdlib.fogl declares the types of variables, functions, and figures that are included in the standard library. Some of the functions are written with their bodies, but most of them are written in Java. The types are used by the analyzer during type-checking. Depends on: nothing.

5.2.7 - Translate (Richard)
Handles the bulk of the code generation. Responsible for turning AST/Scope list into Java source code. Depends on: AST, Strings, Analyze
Call translate through "fogl -t <source file>." Translatewill take a fogl source file (<name>.fogl file), create a source directory called <name>, and generate Java code into <name>/GeneratedCode from <name>.fogl.

5.2.8 - Fogl (Everyone)
Main FOGL compiler. Depends on: Analyze, Translate
fogl is used through fogl <options> <source file>

5.2.9 - Programname (Richard)
Determines the name of the program. Used to name output files.

5.2.10 - Java Boilerplate (Jenny)
Various pre-written Java files included in each outputted jar. Handle drawing, animation, etc.

6 - Test Plan

Because of the graphical nature of the language, coming up with a testing method was not a simple process. We decided to split testing into two kinds of tests. Print tests were tests where no graphics or animation were involved. These tests are automated using a Python script which runs each .fogl test file and compares it to a corresponding .out file.

Graphical tests are more complicated, since they involve non-textual outputs and time delays. Thus, we decided to have a short description of the expected output for each graphical test and compare them manually.

6.1 - A Written Test

fib.fogl is a written test that determines the nth fibonacci number.

6.1.1 - Source Program

```
var fib = func[int] (x:int) {
  if x < 2 { return 1 }
  else { return fib(x-1) + fib(x-2) }
}
print(itos(fib(0)))
print(itos(fib(1)))
print(itos(fib(2)))
print(itos(fib(3)))
print(itos(fib(4)))
print(itos(fib(5)))
```

6.1.2 - Print Output

```
1
1
2
3
5
8
```

6.1.3 - Target Program

```
// Func_fib.java
package GeneratedCode;

import Stdlib.*;
import Type.*;
import Main.*;
import BuiltIns.*;

class Func_fib extends FunctionType {
public static Func_fib fib = new Func_fib();
public IntType call(IntType x) {
if ((Arop.fogl_lt(x, new IntType(2))).value) {
return new IntType(1);

} else {
return Arop.fogl_plus(fib.call(Arop.fogl_minus(x, new IntType(1))),
fib.call(Arop.fogl_minus(x, new IntType(2))));
```

```
 }
}
}

// GeneratedMain.java
package GeneratedCode;

import Stdlib.*;
import Type.*;
import Main.*;
import BuiltIns.*;

public abstract class GeneratedMain {
public static void generatedMain() {
// Standard Library things are left out from here
Func_fib fib = new Func_fib();
Stdlib.Lib.print.call(Stdlib.Lib.itos.call(fib.call(new IntType(0))));
Stdlib.Lib.print.call(Stdlib.Lib.itos.call(fib.call(new IntType(1))));
Stdlib.Lib.print.call(Stdlib.Lib.itos.call(fib.call(new IntType(2))));
Stdlib.Lib.print.call(Stdlib.Lib.itos.call(fib.call(new IntType(3))));
Stdlib.Lib.print.call(Stdlib.Lib.itos.call(fib.call(new IntType(4))));
Stdlib.Lib.print.call(Stdlib.Lib.itos.call(fib.call(new IntType(5))));
}
}
```

6.2 - A Graphical Test

stopsign.fogl defines a stopsign figure and draws an instance of it. The stopsign figure consistz of two components, a polygon with eight sides and a text element with the text “STOP.”

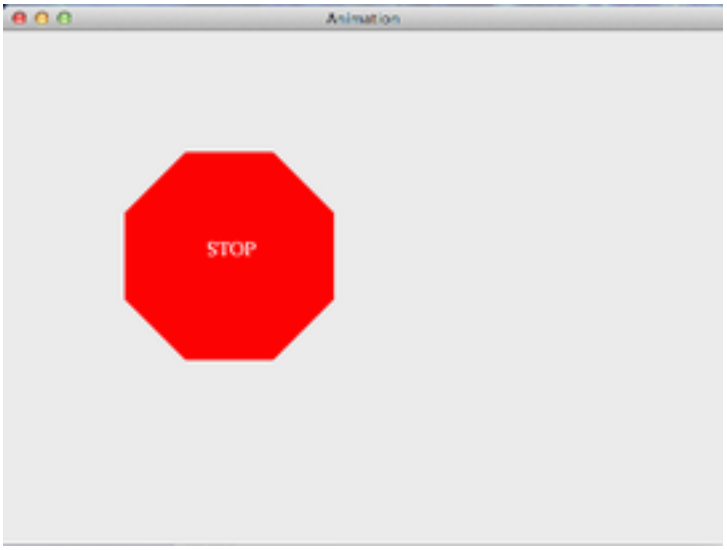
6.2.1 - Source Program

```
; Draws a red octagon at <<200, 200>> and writes STOP on it in white Times New Roman

figure StopSign {
	param loc = <<200, 200>>, col = color
	comp octagon = Poly(loc:loc points:[<<100*cos(PI/8), 100*sin(PI/8)>>,
<<100*cos(3*PI/8), 100*sin(3*PI/8)>>, <<100*cos(5*PI/8), 100*sin(5*PI/8)>>,
<<100*cos(7*PI/8), 100*sin(7*PI/8)>>, <<100*cos(9*PI/8), 100*sin(9*PI/8)>>,
<<100*cos(11*PI/8), 100*sin(11*PI/8)>>, <<100*cos(13*PI/8), 100*sin(13*PI/8)>>,
<<100*cos(15*PI/8), 100*sin(15*PI/8)>>] col:col)
	comp text = Text(loc:loc-<<20, 0>> text:"STOP" col:#fff size:18 font:"Times
New Roman")
}

StopSign(col:#f00)
```

6.2.2 - Graphical Output



6.2.3 - Target Program

```
// StopSign.java
package GeneratedCode;

import Stdlib.*;
import Type.*;
import Main.*;
import BuiltIns.*;

public class StopSign extends Figure {
PairType loc = new PairType(new IntType(200), new IntType(200));
ColorType col;
Poly octagon;
Text text;
void first_init() {
components.add(octagon);
components.add(text);
}
void create() {
octagon = new Poly(loc, new ListType(new PairType(Arop.fogl_times(new IntType(100),
Stdlib.Lib.cos.call(Arop.fogl_div(Stdlib.Lib.PI, new IntType(8)))),
Arop.fogl_times(new IntType(100), Stdlib.Lib.sin.call(Arop.fogl_div(Stdlib.Lib.PI,
new IntType(8))))), new PairType(Arop.fogl_times(new IntType(100),
Stdlib.Lib.cos.call(Arop.fogl_div(Arop.fogl_times(new IntType(3), Stdlib.Lib.PI),
new IntType(8)))), Arop.fogl_times(new IntType(100),
Stdlib.Lib.sin.call(Arop.fogl_div(Arop.fogl_times(new IntType(3), Stdlib.Lib.PI),
new IntType(8))))), new PairType(Arop.fogl_times(new IntType(100),
Stdlib.Lib.cos.call(Arop.fogl_div(Arop.fogl_times(new IntType(5), Stdlib.Lib.PI),
new IntType(8)))), Arop.fogl_times(new IntType(100),
Stdlib.Lib.sin.call(Arop.fogl_div(Arop.fogl_times(new IntType(5), Stdlib.Lib.PI),
new IntType(8))))), new PairType(Arop.fogl_times(new IntType(100),
```

```
Stdlib.Lib.cos.call(Arop.fogl_div(Arop.fogl_times(new IntType(7), Stdlib.Lib.PI),
new IntType(8)))), Arop.fogl_times(new IntType(100),
Stdlib.Lib.sin.call(Arop.fogl_div(Arop.fogl_times(new IntType(7), Stdlib.Lib.PI),
new IntType(8))))), new PairType(Arop.fogl_times(new IntType(100),
Stdlib.Lib.cos.call(Arop.fogl_div(Arop.fogl_times(new IntType(9), Stdlib.Lib.PI),
new IntType(8)))), Arop.fogl_times(new IntType(100),
Stdlib.Lib.sin.call(Arop.fogl_div(Arop.fogl_times(new IntType(9), Stdlib.Lib.PI),
new IntType(8))))), new PairType(Arop.fogl_times(new IntType(100),
Stdlib.Lib.cos.call(Arop.fogl_div(Arop.fogl_times(new IntType(11), Stdlib.Lib.PI),
new IntType(8)))), Arop.fogl_times(new IntType(100),
Stdlib.Lib.sin.call(Arop.fogl_div(Arop.fogl_times(new IntType(11), Stdlib.Lib.PI),
new IntType(8))))), new PairType(Arop.fogl_times(new IntType(100),
Stdlib.Lib.cos.call(Arop.fogl_div(Arop.fogl_times(new IntType(13), Stdlib.Lib.PI),
new IntType(8)))), Arop.fogl_times(new IntType(100),
Stdlib.Lib.sin.call(Arop.fogl_div(Arop.fogl_times(new IntType(13), Stdlib.Lib.PI),
new IntType(8))))), new PairType(Arop.fogl_times(new IntType(100),
Stdlib.Lib.cos.call(Arop.fogl_div(Arop.fogl_times(new IntType(15), Stdlib.Lib.PI),
new IntType(8)))), Arop.fogl_times(new IntType(100),
Stdlib.Lib.sin.call(Arop.fogl_div(Arop.fogl_times(new IntType(15), Stdlib.Lib.PI),
new IntType(8))))))), col, null, null);
text = new Text(Arop.fogl_minus(loc, new PairType(new IntType(20), new IntType(0))
), new StringType("STOP"), new ColorType(0xff, 0xff, 0xff, 1.), new IntType(18), new
StringType("Times New Roman"));
}
void b_init() {
}
public StopSign(AbstractType... arg) {
b_init();
if (arg[0] != null) { loc = (PairType) arg[0]; }
if (arg[1] != null) { col = (ColorType) arg[1]; }
create();
first_init();
if (_update != null)
_animating =true;
}
}

// GeneratedMain.java
package GeneratedCode;

import Stdlib.*;
import Type.*;
import Main.*;
import BuiltIns.*;

public abstract class GeneratedMain {
public static void generatedMain() {
// Standard Library things are left out from here
```

```
new StopSign(null, new ColorType(0xff, 0x00, 0x00, 1.));
}
}
```

7 - Lessons Learned

7.1 - Julian

I learned to make sure that the project you're taking on is manageable both for you and for your group mates. I also learned to be more assertive about deadlines and making sure code is being written on time. I learned to make sure that everyone in the group has a clear understanding of the language as a whole as well as his/her individual part. also learned not to get too attached to language features and to remember that everything can be changed need be.

7.2 - Richard

I learned the importance of coordinating with people, in particular the importance of dividing up work evenly and determining common interfaces between the code of group members. Using SVN for the first time, I learned the limitations of a version control system and how to resolve conflicts. I also learned the importance of time in any programming project - with a bit more time and analysis, anything can be done by adding extra layers of indirection. Though I was not the leader of the group, I saw the importance of coordinating and helping each other with programming, as the final result is entirely a group effort.

7.3 - Jenny

It's surprisingly hard to communicate and coordinate with people! Most of my problems arose because I was unclear on what precisely I was supposed to be working on, what other parts of the code would pass to me, etc. I think I gained an appreciation for why compilers are made the way they are; when I first heard an overview I thought it was incredibly redundant but now I see why it's useful to have them split up like this.

8 - Code Listing

8.1 - scanner.mll

```
{ open Parser
  let dbl_char c = (String.make 1 c) ^ (String.make 1 c) (* Turns 'x' into "xx" *)
}

let digit = ['0'-'9']
let exp = ['e' 'E'] ['+' '-']? digit+
let float = ((digit+ '.') | digit* '.' digit+) exp? | digit+ exp
let hex = digit | ['a'-'f' 'A'-'F']
let nl = [' ' '\r' '\t']* '\n' | [' ' '\r' '\t']* ';' [^ '\n']* '\n' (* Single-line
comments are also newlines *)

rule token = parse
    [' ' '\r' '\t'] { token lexbuf }
  | ('"' ([^ '"' ]* as lit) '"')
  | ('\'' ([^ '\'' ]* as lit) '\'') { STRING(lit) }
  | digit+ as lit { INT(int_of_string lit) }
  | float as lit { FLOAT(float_of_string lit) }
```

```
  | '#' (hex hex as r) (hex hex as g) (hex hex as b) { COLOR(r, g, b, 1.0) }
  | '#' (hex hex as r) (hex hex as g) (hex hex as b) '@' (float as a) { COLOR(r, g,
b, float_of_string a) }
  | '#' (hex as r) (hex as g) (hex as b) { COLOR(dbl_char r, dbl_char g, dbl_char b,
1.0) }
  | '#' (hex as r) (hex as g) (hex as b) '@' (float as a) { COLOR(dbl_char r,
dbl_char g, dbl_char b, float_of_string a) }
  | "+" { PLUS }
  | "-" { MINUS }
  | "*" { TIMES }
  | "/" { DIV }
  | "//" { DIVDIV }
  | "^" { EXP }
  | "%" { PERCENT }
  | "+=" { PLUSEQ }
  | "-=" { MINUSEQ }
  | "*=" { TIMESEQ }
  | "/=" { DIVEQ }
  | "//=" { DIVDIVEQ }
  | "^=" { EXPEQ }
  | "%=" { PERCENTEQ }
  | "=" { EQ }
  | "==" { EQEQ }
  | "!=" { NEQ }
  | ">" { GT }
  | "<" { LT }
  | ">=" { GTE }
  | "<=" { LTE }
  | "!" { NOT }
  | "&" { AND }
  | "|" { OR }
  | "?" { QUESTION }
  | "." { DOT }
  | ":" { COLON }
  | "," { COMMA }
  | "(" nl* { LPAREN }
  | nl* ")" { RPAREN }
  | "{" nl* { LBRACE }
  | nl* "}" { RBRACE }
  | "[" nl* { LBRACKET }
  | nl* "]" { RBRACKET }
  | "<<" nl* { LANGLE }
  | nl* ">>" { RANGLE }
  | nl+ { NL }
  | "param" { PARAM }
  | "comp" { COMP }
  | "var" { VAR }
  | "if" { IF }
```

```
  | nl* [' ' '\r' '\t']*  "elseif" { ELSEIF }
  | nl* [' ' '\r' '\t']*  "else" { ELSE }
  | "do" { DO }
  | "while" { WHILE }
  | "true" { TRUE }
  | "false" { FALSE }
  | "return" { RETURN }
  | "figure" { FIGURE }
  | "int" { T_INT }
  | "float" { T_FLOAT }
  | "string" { T_STRING }
  | "bool" { BOOL }
  | "func" { FUNC }
  | "color" { T_COLOR }
  | "pair" { PAIR }
  | ['a'-'z' 'A'-'Z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_']* as lit { ID(lit) }
  | eof { EOF }
  | _ as char { raise (Failure("Illegal character " ^
                          Char.escaped char)) }
```

8.2 - parser.mly

```
%{ open Ast %}

%token <string> STRING ID
%token <int> INT
%token <float> FLOAT
%token <string * string * string * float> COLOR
%token PLUS MINUS TIMES DIV DIVDIV EXP PERCENT PLUSEQ MINUSEQ TIMESEQ DIVEQ DIVDIVEQ
EXPEQ PERCENTEQ
%token EQ PLUSEQ MINUSEQ TIMESEQ DIVEQ DIVDIVEQ EXPEQ PERCENTEQ
%token EQEQ NEQ GT LT GTE LTE NOT AND OR QUESTION
%token DOT COLON COMMA LPAREN RPAREN LBRACKET RBRACKET LBRACE RBRACE LANGLE RANGLE
%token PARAM COMP VAR IF ELSEIF ELSE DO WHILE TRUE FALSE RETURN
%token FIGURE T_INT T_FLOAT T_STRING FUNC BOOL T_COLOR PAIR
%token NL EOF

%nonassoc NL
%left COMMA
%right EQ PLUSEQ MINUSEQ TIMESEQ DIVEQ DIVDIVEQ EXPEQ PERCENTEQ
%nonassoc EQEQ NEQ
%nonassoc LT GT LTE GTE
%left OR
%left AND
%left PLUS MINUS
%left TIMES DIV DIVDIV PERCENT
%left EXP
%nonassoc UMINUS NOT QUESTION
```

```
%left DOT
%nonassoc LPAREN RPAREN LBRACKET RBRACKET LBRACE RBRACE LANGLE RANGLE

%start program
%type <Ast.program> program

%%

program:
    program_contents { List.rev $1 }

program_contents:
    ws { [] }
  | program_contents stmt NL { $2 :: $1 }
  | program_contents stmt EOF { $2 :: $1 }

ws:
    /* nothing */ {}
  | NL {}

figure:
    FIGURE ID ws LBRACE figure_body figure_decl ws RBRACE { Figure(Ast.nextScope(),
$2, $5 @ $6) }
  | FIGURE ID ws LBRACE ws RBRACE { Figure(0, $2, []) }

block:
    LBRACE RBRACE { Ast.nextScope(), [] }
  | LBRACE stmt_list stmt ws RBRACE { Ast.nextScope(), List.rev ($3 :: $2) }

typ:
    T_INT { IntT }
  | T_FLOAT { FloatT }
  | T_STRING { StringT }
  | BOOL { BoolT }
  | T_COLOR { ColorT }
  | PAIR { PairT }
  | ID LBRACE RBRACE { FigureT(Some($1)) }
  | FIGURE LBRACE RBRACE { FigureT(None) }
  | typ LBRACKET RBRACKET { ListT($1) }
  | typ LPAREN typ_list_opt RPAREN { FuncT($1, $3) }
  | LPAREN typ_list_opt RPAREN { FuncT(VoidT, $2) }

typ_list:
    typ { [$1] }
  | typ_list COMMA ws typ { $4 :: $1 }

typ_list_opt:
    /* nothing */ { [] }
```

```
  | typ_list { List.rev $1 }

stmt_list:
    /* nothing */ { [] }
  | stmt_list stmt NL { $2 :: $1 }

figure_decl:
    PARAM ws decl_list { List.map (fun (x, y, z) -> (Param, x, y, z)) (List.rev $3) }
  | COMP ws decl_list { List.map (fun (x, y, z) -> (Comp, x, y, z)) (List.rev $3) }
  | VAR ws decl_list { List.map (fun (x, y, z) -> (Var, x, y, z)) (List.rev $3) }

figure_body:
    /* nothing */ { [] }
  | figure_body figure_decl NL { $1 @ $2 }

decl_list:
    decl { [$1] }
  | decl_list COMMA ws decl { $4 :: $1 }

decl:
    ID EQ typ { $1, Some($3), None }
  | ID EQ expr { $1, None, Some($3) }

arg_list:
    ID COLON typ { [$1, $3] }
  | arg_list COMMA ws ID COLON typ { ($4, $6) :: $1 }

arg_list_opt:
    /* nothing */ { [] }
  | arg_list { List.rev $1 }

expr_list:
    expr { [$1] }
  | expr_list COMMA ws expr { $4 :: $1 }

expr_list_opt:
    /* nothing */ { [] }
  | expr_list { List.rev $1 }

param:
    ID COLON expr { $1, $3 }

param_list:
    param { [$1] }
  | param_list ws param { $3 :: $1 }

param_list_opt:
    COLON { [] }
```

```
  | param_list { List.rev $1 }

elseif:
    /* nothing */ { Ast.nextScope(), [] }
  | ELSE ws block { $3 }
  | ELSEIF expr ws block elseif { Ast.nextScope(), [If($2, $4, $5)] }

stmt:
    expr { Expr($1) }
  | figure { $1 }
  | VAR ws decl_list { Decl(List.map (fun (x, y, z) -> (Var, x, y, z)) (List.rev $3))
 }
  | expr EQ expr { Assign($1, $3) }
  | expr PLUSEQ expr { Assign($1, Arop($1, Plus, $3)) }
  | expr MINUSEQ expr { Assign($1, Arop($1, Minus, $3)) }
  | expr TIMESEQ expr { Assign($1, Arop($1, Times, $3)) }
  | expr DIVEQ expr { Assign($1, Arop($1, Div, $3)) }
  | expr DIVDIVEQ expr { Assign($1, Arop($1, Divdiv, $3)) }
  | expr EXPEQ expr { Assign($1, Arop($1, Exp, $3)) }
  | expr PERCENTEQ expr { Assign($1, Arop($1, Percent, $3)) }
  | RETURN expr { Return(Some($2)) }
  | RETURN { Return(None) }
  | WHILE expr ws block { While($2, $4) }
  | DO ws block ws WHILE expr { DoWhile($3, $6) }
  | IF expr ws block elseif { If($2, $4, $5) }

expr:
    LPAREN expr RPAREN { $2 }
  | INT { Int($1) }
  | STRING { String($1) }
  | FLOAT { Float($1) }
  | TRUE { Bool(true) }
  | FALSE { Bool(false) }
  | COLOR { Color($1) }
  | LANGLE expr COMMA ws expr RANGLE { Pair($2, $5) }
  | ID { Id($1) }
  | FUNC LBRACKET typ RBRACKET LPAREN arg_list_opt RPAREN block { Function($3, $6,
$8) }
  | FUNC LPAREN arg_list_opt RPAREN block { Function(VoidT, $3, $5) }
  | FUNC LBRACKET typ RBRACKET LPAREN arg_list_opt RPAREN expr { Function($3, $6,
(Ast.nextScope(), [Return(Some($8))])) }
  | LBRACKET expr_list_opt RBRACKET { List($2) }
  | expr LBRACKET expr RBRACKET { Subscript($1, $3) }
  | expr DOT ID { Dot($1, $3) }
  | expr PLUS expr { Arop($1, Plus, $3) }
  | expr MINUS expr { Arop($1, Minus, $3) }
  | expr TIMES expr { Arop($1, Times, $3) }
  | expr DIV expr { Arop($1, Div, $3) }
```

```
  | expr DIVDIV expr { Arop($1, Divdiv, $3) }
  | expr EXP expr { Arop($1, Exp, $3) }
  | expr PERCENT expr { Arop($1, Percent, $3) }
  | expr EQEQ expr { Logop($1, Eqeq, $3) }
  | expr NEQ expr { Logop($1, Neq, $3) }
  | expr GT expr { Logop($1, Gt, $3) }
  | expr LT expr { Logop($1, Lt, $3) }
  | expr GTE expr { Logop($1, Gte, $3) }
  | expr LTE expr { Logop($1, Lte, $3) }
  | expr AND expr { Logop($1, And, $3) }
  | expr OR expr { Logop($1, Or, $3) }
  | NOT expr { Unop(Not, $2) }
  | QUESTION expr { Unop(Question, $2) }
  | MINUS expr %prec UMINUS { Unop(Neg, $2) }
  | func { $1 }
  | ID LPAREN param_list_opt RPAREN { FigInstance($1, $3) }

func:
    ID LPAREN expr_list_opt RPAREN { Call(Id($1), $3) }
  | expr DOT ID LPAREN expr_list_opt RPAREN { Call(Dot($1, $3), $5) }
  | func LPAREN expr_list_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN LPAREN expr_list_opt RPAREN { Call($2, $5) }
```

8.3 - ast.ml

```
(* For use by parser *)
let numScopes = ref 0
let nextScope unit = numScopes := !numScopes + 1 ; !numScopes

type arop = Plus | Minus | Times | Div | Divdiv | Exp | Percent (* Arithmetic
operator *)
type logop = Eqeq | Neq | Gt | Lt | Gte | Lte | And | Or (* Logical operator *)
type unop = Neg | Not | Question (* Unary operator *)

type decl_type = Param | Comp | Var

type typ = IntT | FloatT | StringT | BoolT | ColorT | PairT
| FigureT of string option
| VoidT (* Used for function return type, empty list type *)
| FigDefT (* Used for figure definitions *)
| ListT of typ
| FuncT of typ * typ list (* return type * argument types *)

and expr =
    Int of int
  | Float of float
  | String of string
  | Bool of bool
  | Function of typ * (string * typ) list * block
```

```
  | List of expr list
  | Color of (string * string * string * float)
  | Pair of (expr * expr)
  | Subscript of expr * expr
  | Dot of expr * string
  | Id of string
  | Arop of expr * arop * expr
  | Logop of expr * logop * expr
  | Unop of unop * expr
  | Call of expr * expr list
  | FigInstance of string * (string * expr) list
and stmt =
        Expr of expr
  | Figure of int * string * decls (* int represents scope index *)
  | Decl of decls
  | Assign of expr * expr
  | Return of expr option
  | If of expr * block * block
  | While of expr * block
  | DoWhile of block * expr
and block = int * stmt list (* int represents scope index *)
and decls = (decl_type * string * typ option * expr option) list

type program = stmt list

module StringMap = Map.Make(String)

type figureInfo = {
  name: string; (* Figure name, e.g. Rect *)
  params: string list; (* All params *)
  rParams: string list; (* Required params *)
  comps: string list; (* All components *)
  vars: string list; (* All vars *)
}
type scope = {
  (* Map of symbols to types *)
  symbols: typ StringMap.t;
  (* Array index of outer scope.  None for global scope. *)
  outer: int option;
  (* Figure information, if scope is a figure *)
  figure: figureInfo option;
  (* For functions only. Map of symbols to int scope index *)
  capture: int StringMap.t option;
  (* For functions only. Return type. *)
  typ: typ option;
}
```

8.4 - strings.ml

```
(* strings.ml
  Contains functions for pretty printing
*)

open Ast

let of_list = function
  [] -> ""
| hd::tl -> List.fold_left (fun s x -> s ^ ", " ^ x) hd tl

let rec of_type = function
  IntT -> "int"
| FloatT -> "float"
| StringT -> "string"
| BoolT -> "bool"
| ColorT -> "color"
| PairT -> "pair"
| FigureT(Some(id)) -> id ^ "{}"
| FigureT(None) -> "figure{}"
| VoidT -> ""
| FigDefT -> "figureDefinition"
| ListT(typ) -> (of_type typ) ^ "[]"
| FuncT(ret, args) -> (of_type ret) ^ "(" ^
    of_list (List.map (fun t -> of_type t) args) ^ ")"

let of_arop = function
  Plus -> "+"
| Minus -> "-"
| Times -> "*"
| Div -> "/"
| Divdiv -> "//"
| Exp -> "^"
| Percent -> "%"

let of_logop = function
  Eqeq -> "=="
| Neq -> "!="
| Gt -> ">"
| Lt -> "<"
| Gte -> ">="
| Lte -> "<="
| And -> "&"
| Or -> "|"

let of_unop = function
  Neg -> "-"
| Not -> "!"
| Question -> "?"
```

```
let of_scope scope = (match scope.outer with
  None -> "none" | Some(int) -> string_of_int int) ^
  (match scope.capture with None -> "" | Some(captures) -> "(Captures " ^
    StringMap.fold (fun sym _ s -> s ^ sym ^ ", ") captures "" ^
  ")") ^
  " -> " ^ (StringMap.fold
  (fun sym ty s -> s ^ sym ^ ": " ^ (of_type ty) ^ ", ") scope.symbols "")

(*
let rec of_expr = function
  Int(i) -> string_of_int i
| Float(fl) -> string_of_float fl
| String(str) -> str
| Bool(boo) -> string_of_bool boo
| Function(arg1, arg2, arg3) -> "function"
| List (arg1) -> let rawr = List.map (fun x -> of_expr x) arg1 in
                 (of_list rawr)
| Color((arg1)) -> "color"
    (*"col" ^ arg1 ^ arg2 ^ arg3 ^ (string_of_float arg4) *)
| Pair(arg1) -> "pair"
| Subscript(arg1, arg2) -> (of_expr arg1) ^ "_" ^ (of_expr arg2)
| Dot(arg1, arg2)-> "dot(" ^ (of_expr arg1) ^ arg2 ^ ")"
| Id(str) -> str
| Arop(expr1, arop, expr2) -> (of_arop arop) ^ "(" ^ (of_expr expr1) ^
                           ", " ^ (of_expr expr2) ^ ")"
| Logop(expr1, logop, expr2) -> (of_logop logop) ^ "(" ^ (of_expr expr1) ^
                           ", " ^ (of_expr expr2) ^ ")"

| Unop(unop, expr) -> (of_unop unop) ^ "("^(of_expr expr)^ ")"
| Call(str, expr_list) -> "call"
| FigInstance(str, some_list) -> "FigInstance"

let rec of_stmt =
    let of_block block = match block with
                      | (k , lst) -> (let rawr = List.map
                         (fun x -> of_stmt x) lst in
                            (of_list rawr)) in
function
  Expr(expr) -> (of_expr expr)
| Figure(arg1,arg2,decls) -> "Figure"
| Decl(arg1) -> "Decl"
| Assign(expr1, expr2) -> "Assign(" ^ (of_expr expr1) ^ ", "
                        ^ (of_expr) expr2 ^ ")"
| Return(expr) -> "Return(" ^ ")"
| If(expr, b1, b2) -> "If(" ^ (of_expr expr) ^ ", " ^
                    (of_block b1) ^ ", " ^ (of_block b2) ^ ")"
| While(expr, bb) -> "While(" ^ (of_expr expr) ^ ", " ^ (of_block bb) ^
```

```
                    ")"
| DoWhile(b1, expr) -> "DoWhile(" ^ (of_block b1) ^ ", " ^ (of_expr expr)
                       ^ ")"
                       *)

let of_scopes scopes = List.fold_left (fun s scope -> s ^ (match scope.figure with
  None -> "" | Some({name=id; _}) -> id ^ ", ") ^ of_scope scope ^
  "\n") "" scopes

(*
let of_program program =
    let rawr = List.map (fun x -> of_stmt x) program in
       (of_list rawr)
       *)
```

8.5 - analyze.ml

```
(*  analyze.ml takes the AST and does the following
  - Checks/infers types
  - Creates a scope list/symbol map(s)
  - Handles captures for lambdas
 *)

open Ast
open Strings

exception Name_error of string (* Identifier is undeclared or declared more than once
*)
exception Type_error of string (* Something is of the wrong type *)
exception Access_error of string (* Something cannot be accessed or is in the wrong
place *)
exception Internal_error (* Thrown when code that should not be reached is reached *)

(* Returns closest scope containing a symbol.
   Raises exception if symbol can't be found *)
let rec locateSymbol scopes scope sym =
  if StringMap.mem sym scope.symbols
  then scope
  else match scope.outer with
    None -> raise (Name_error ("Undeclared identifier: " ^ sym))
  | Some(outer) -> locateSymbol scopes (List.nth scopes outer) sym

(* Returns true if types are compatible *)
let rec compatible = function
    FloatT, IntT -> true (* Ints can be used as floats *)
  | ListT(ListT(l1)), ListT(ListT(l2)) -> compatible (ListT(l1), ListT(l2))
  | ListT(_), ListT(VoidT) -> true (* Empty list matches any list type *)
  | FigureT(None), FigureT(_) | FigureT(_), FigureT(None) -> true
  | FigDefT, _ | _, FigDefT -> false (* Figure definitions are never equal *)
```

```
  | t1, t2 -> if t1 = t2 then true else false

(* Adds symbol to scope, throws error if it's already there *)
let addSymbol sym typ scope = match sym, typ with
  "_animating", BoolT -> StringMap.add "_animating" BoolT scope
| "_visible", BoolT -> StringMap.add "_visible" BoolT scope
| "_animating", _ -> raise (Type_error "_animating must be a bool")
| "_visible", _ -> raise (Type_error "_visible must be a bool")
| sym, typ -> if StringMap.mem sym scope
    then raise (Name_error ("Identifier already declared: " ^ sym))
    else StringMap.add sym typ scope

(* Performs a function on a scope's symbols and returns updated scope list *)
let modSymbols func s t scopes ind =
  let list = func s t (List.nth scopes ind).symbols in
  let (_, l) = List.fold_left
    (fun (i, l) x -> (i+1, (
      if i = ind
      then {symbols=list; outer=x.outer; figure=x.figure; capture=x.capture;
typ=x.typ}
      else x
    )::l)) (0, []) scopes
  in List.rev l

(* Add capture to a scope given scope list and index *)
let addCapture sym loc scopes ind =
  let map = match (List.nth scopes ind).capture with
    None -> raise Internal_error | Some(capture) -> StringMap.add sym loc capture in
  let (_, l) = List.fold_left
    (fun (i, l) x -> (i+1, (
      if i = ind
      then {symbols=x.symbols; outer=x.outer; figure=x.figure; capture=Some(map);
typ=x.typ}
      else x
    )::l)) (0, []) scopes
  in List.rev l

(* Add capture to function if necessary *)
let capture scopes scope sym =
  if StringMap.mem sym (List.nth scopes scope).symbols
  then scopes
  else let rec locate sym scope =
    if StringMap.mem sym (List.nth scopes scope).symbols
    then scope
    else match (List.nth scopes scope).outer with
      None -> raise (Name_error ("Undeclared identifier: " ^ sym))
    | Some(outer) -> locate sym outer
    in addCapture sym (locate sym scope) scopes scope
```

```
(* Add declaration to figure scope *)
let modFigureScope declType value scopes ind =
  let (_, l) = List.fold_left
    (fun (i, l) x -> (i+1, (
      match x.figure with None -> x | Some(fig) ->
      if i = ind
      then {symbols=x.symbols;
        outer=x.outer;
        figure = Some(match declType with
          "param" -> {name=fig.name; params=value::fig.params;
            rParams=fig.rParams; comps=fig.comps; vars=fig.vars}
        | "rParam" -> {name=fig.name; params=value::fig.params;
            rParams=value::fig.rParams; comps=fig.comps; vars=fig.vars}
        | "comp" -> {name=fig.name; params=fig.params;
            rParams=fig.rParams; comps=value::fig.comps; vars=fig.vars}
        | "var" -> {name=fig.name; params=fig.params;
            rParams=fig.rParams; comps=fig.comps; vars=value::fig.vars}
        | _ -> raise Internal_error);
        capture=x.capture; typ=x.typ}
      else x
    )::l)) (0, []) scopes
  in List.rev l

(* Return the type of arithmetic operation if valid *)
let aropType =
  let err t1 op t2 = raise (Type_error ("Cannot perform " ^ Strings.of_arop op ^
    " on " ^ Strings.of_type t1 ^ " and " ^ Strings.of_type t2)) in
  function
  IntT, Div, FloatT | FloatT, Div, IntT | FloatT, Div, FloatT | IntT, Div, IntT ->
FloatT
| IntT, Divdiv, FloatT | FloatT, Divdiv, IntT | FloatT, Divdiv, FloatT | IntT,
Divdiv, IntT -> IntT
| IntT, _, IntT -> IntT
| IntT, _, FloatT | FloatT, _, IntT -> FloatT | FloatT, _, FloatT -> FloatT
| PairT, _, IntT | PairT, _, FloatT | IntT, _, PairT
| FloatT, _, PairT | PairT, _, PairT -> PairT
| StringT, Plus, StringT -> StringT
| (ListT(_) as t1), Plus, (ListT(_) as t2) when t1 = t2 -> t1
| t1, op, t2 -> (err t1 op t2)

(* Return the type of logical operation if valid *)
let logopType =
    let err t1 op t2 = raise (Type_error ("Cannot perform " ^ Strings.of_logop op ^
    " on " ^ Strings.of_type t1 ^ " and " ^ Strings.of_type t2)) in
  function
  BoolT, And, BoolT | BoolT, Or, BoolT -> BoolT
| t1, (And as op), t2 | t1, (Or as op), t2 -> (err t1 op t2)
```

```
| IntT, _, IntT | IntT, _, FloatT | FloatT, _, IntT | FloatT, _, FloatT -> BoolT
| (FigDefT as t1), op, (FigDefT as t2)
| (FigureT(_) as t1), op, (FigureT(_) as t2)
| (FuncT(_, _) as t1), op, (FuncT(_, _) as t2) -> (err t1 op t2)
| t1, Eqeq, t2 when t1 = t2 -> BoolT
| t1, Neq, t2 when t2 = t2 -> BoolT
| t1, op, t2 -> (err t1 op t2)

(* Return the type of unary operation if valid *)
let unopType =
    let err op t = raise (Type_error ("Cannot perform " ^ Strings.of_unop op ^
    " on " ^ Strings.of_type t)) in
  function
  Not, BoolT -> BoolT
| Neg, IntT -> IntT
| Neg, FloatT -> FloatT
| Question, (FigDefT as t)
| Question, (FigureT(_) as t)
| Question, (FuncT(_, _) as t) -> (err Question t)
| Question, _ -> BoolT
| op, t -> (err op t)

(* Add nth scope to scope list *)
let addScope scopes n outer figure capture typ =
  let scope = {symbols=StringMap.empty; outer=outer;
    figure=figure; capture=capture; typ=typ} in
  let (_, l) = List.fold_left
    (fun (i, l) x -> (i+1, if i = n then scope::l else x::l)) (0, []) scopes
  in List.rev l

(* Find scope given figure type *)
let figureScope figure scopes =
  let scopeList = List.map
    (fun x -> match x.figure with None -> None
    | Some(info) -> if figure = info.name then Some(x) else None) scopes
  in let scope = List.fold_left (fun s x -> match x with None -> s | Some(x) ->
Some(x))
    None scopeList
  in match scope with
    None -> raise (Name_error ("Undeclared identifier: " ^ figure)) | Some(scope) ->
scope

(* Creates a list with n empty scopes *)
let rec initScopeList l = function
  0 -> List.rev l
| n -> initScopeList
    ({symbols=StringMap.empty; outer=None; figure=None; capture=None; typ=None}::l)
(n-1)
```

```
(* Determine the type of an expression
   Raise exception if type cannot be determined
   Raise exception if types are invalid *)
let rec exprType scope scopes expr =
  let rec exprType_ = function
    Int(_) -> IntT
  | Float(_) -> FloatT
  | String(_) -> StringT
  | Bool(_) -> BoolT
  | Function(typ, args, (scope, stmts)) -> FuncT(typ, List.map snd args)
  | List(l) -> let rec listType = function
      [] -> VoidT
    | l -> match List.fold_right (fun el t ->
          match el with List(_) -> t | expr -> exprType_ el) l VoidT with
        VoidT -> ListT(listType (List.fold_left (fun l el ->
          match el with List(n) -> l @ n | _ -> l) [] l))
      | t -> t
    in ListT(List.fold_left
      (fun t e -> if compatible (t, exprType_ e)
        then t else raise (Type_error ("Unexpected " ^ Strings.of_type (exprType_ e)
^
          " in list of " ^ Strings.of_type t)))
    (listType l) l)
  | Color(_) -> ColorT
  | Pair(x, y) -> (match ((exprType_ x), (exprType_ y)) with
    IntT, IntT -> PairT
    | IntT, FloatT -> PairT
    | FloatT, IntT -> PairT
    | FloatT, FloatT -> PairT
    | _, _ -> raise (Type_error "Pair type must have int or float parts"))
  | Subscript(list, index) ->
    (match ((exprType_ list), (exprType_ index)) with
      ListT(typ), IntT -> typ
    | typ, _ -> raise (Type_error ("Cannot subscript " ^ Strings.of_type typ)))
  | Dot(figure, id) -> (match (exprType_ figure) with
      FigureT(Some(fig)) -> let figScope = figureScope fig scopes in
        if match figScope.figure with None -> raise Internal_error
        | Some(fig) -> List.mem id fig.vars
        then raise (Access_error ("Cannot access private var " ^ id))
        else (try StringMap.find id figScope.symbols
      with Not_found -> raise (Name_error ("Undeclared identifier: " ^ id)))
    | FigureT(None) -> (match id with
          "_visible" | "_animating" -> BoolT
        | _ -> raise (Access_error ("Cannot access " ^ id ^ " from generic figure
type")))
    | typ -> raise (Type_error ("Dot operator cannot be used on "
      ^ Strings.of_type typ)))
```

```
  | Id(id) -> StringMap.find id (locateSymbol scopes scope id).symbols
  | Arop(e1, op, e2) -> aropType ((exprType_ e1), op, (exprType_ e2))
  | Logop(e1, op, e2) -> logopType ((exprType_ e1), op, (exprType_ e2))
  | Unop(op, expr) -> unopType (op, (exprType_ expr))
  | Call(expr, args) ->
      (match exprType_ expr with
        FuncT(ret, argTypes) ->
          if argTypes = List.map (fun x -> exprType_ x) args
          then ret
          else raise (Type_error ("Arguments of function not of proper types"))
      | FigDefT ->
          if List.length args <> 1 then raise (Type_error "Figure casts take exactly
one argument")
          else (match (expr, exprType_ (List.nth args 0)) with
            Id(id), FigureT(None) -> FigureT(Some(id))
          | _, t -> raise (Type_error "Figure casts can only be done on generic
figure types"))
      | typ -> raise (Type_error ("Cannot make call to " ^ Strings.of_type typ)))
  | FigInstance(id, params) -> let figScope = (figureScope id scopes) in
      let types = List.map
        (fun (id, value) -> StringMap.find id figScope.symbols, (exprType_ value))
params in
      List.iter (fun (expected, typ) ->
        if compatible (expected, typ) then ()
        else raise (Type_error ("Expected " ^ Strings.of_type expected ^
        ", instead found " ^ Strings.of_type typ))) types ;
      match figScope.figure with None -> raise Internal_error | Some(fig) ->
      List.iter (fun x -> if List.mem x (fst (List.split params)) then ()
        else raise (Type_error ("Missing required parameter " ^ x ^
        " in instance of " ^ id))) fig.rParams ;
      FigureT(Some(id))
  in exprType_ expr

(* Returns true if the expression represents a component.
   Used to make sure components aren't directly assigned *)
let isComp scope scopes = function
  Dot(figure, id) -> (match (exprType scope scopes figure) with
      FigureT(Some(fig)) -> let figScope = (figureScope fig scopes) in
        (match figScope.figure with None -> raise Internal_error | Some(figInfo) ->
        if List.mem id figInfo.comps then true else false)
    | _ -> false)
| Id(id) -> (match (locateSymbol scopes scope id).figure with
      None -> false
    | Some(figInfo) -> if List.mem id figInfo.comps then true else false)
| _ -> false

(* Extract leftMost id from assignable expression.
   Throws error if expression is not assignable.
```

```
   Used for handling captures *)
let rec extractId = function
  Id(id) -> id
| Dot(Id(id), _) | Subscript(Id(id), _) -> id
| Dot(expr, _) | Subscript(expr, _) -> extractId expr
| _ -> raise (Type_error "Expression is not assignable")

(* Perform captures for a function *)
let makeCaptures scopes current stmts =
  let rec makeCapturesExpr scopes = function
    Id(id) -> capture scopes current id
  | Dot(expr, _) | Subscript(expr, _) -> capture scopes current (extractId expr)
  | Pair(expr1, expr2) | Arop(expr1, _, expr2) | Logop(expr1, _, expr2) ->
      makeCapturesExpr (makeCapturesExpr scopes expr2) expr1
  | Unop(_, expr) -> makeCapturesExpr scopes expr
  | List(exprs) | Call(_, exprs) ->
      List.fold_left (fun scopes e -> makeCapturesExpr scopes e) scopes exprs
  | FigInstance(_, list) ->
      List.fold_left (fun scopes (_, e) -> makeCapturesExpr scopes e) scopes list
  | _ -> scopes
  in let rec _makeCaptures scopes = function
    [] -> scopes
  | Expr(expr)::tl -> _makeCaptures (makeCapturesExpr scopes expr) tl
  | Assign(expr, value)::tl -> let scopes = capture scopes current (extractId expr)
in
      _makeCaptures (makeCapturesExpr scopes value) tl
  | Return(Some(expr))::tl -> _makeCaptures (makeCapturesExpr scopes expr) tl
  | If(expr, (_, stmts1), (_, stmts2))::tl ->
      let scopes = _makeCaptures (_makeCaptures scopes stmts1) stmts2 in
      _makeCaptures (makeCapturesExpr scopes expr) tl
  | While(expr, (_, stmts))::tl | DoWhile((_, stmts), expr)::tl ->
    _makeCaptures (makeCapturesExpr (_makeCaptures scopes stmts) expr) tl
  | _::tl -> _makeCaptures scopes tl
in _makeCaptures scopes stmts

(* Make the scope list/symbol tables
   This is the main function of analyze.ml
   It takes one argument, an Ast.program
   It returns a list of scopes (scope type defined in ast.ml) *)
let makeTable sList =
  let rec makeScope scopes current = function
    [] -> scopes
  | Figure(index, id, decls)::tl -> (* Figure definition *)
      if current <> 0 then raise (Access_error "Figure definitions must be in the
global scope")
      else let scopes = modSymbols addSymbol id FigDefT scopes current in
      let scopes = addScope scopes index (Some current)
        (Some {name=id; params=[]; rParams=[]; comps=[]; vars=[]}) None None in
```

```
      let decls = [(Param, "_visible", Some(BoolT), None);
        (Param, "_animating", Some(BoolT), None)] @ decls in
      makeScope (List.fold_left (fun scopes decl -> let scopes = match decl with
          Param, "_animating", _, _ | Param, "_visible", _, _ -> scopes
        | Var, "_update", Some(FuncT(VoidT, [IntT])), _ -> scopes
        | Var, "_update", None, Some(expr)
            when (exprType (List.nth scopes current) scopes expr) = FuncT(VoidT,
[IntT]) -> scopes
        | Var, "_update", _, _ -> raise (Type_error "_update must have type (int)")
        | _, "_animating", _, _ -> raise (Type_error "_animating must be declared as
a param")
        | _, "_visible", _, _ -> raise (Type_error "_visible must be declared as a
param")
        | _, "_update", _, _ -> raise (Type_error "_update must be declared as a
var")
        | Param, sym, Some(_), None -> modFigureScope "rParam" sym scopes index
        | Param, sym, None, Some(_) -> modFigureScope "param" sym scopes index
        | Comp, sym, _, _ -> modFigureScope "comp" sym scopes index
        | Var, sym, _, _ -> modFigureScope "var" sym scopes index
        | _ -> scopes in
        parseDecl scopes index decl) scopes decls) current tl
  | Decl(decls)::tl -> let scopes = List.fold_left
    (fun scopes decl -> parseDecl scopes current decl) scopes decls
    in makeScope scopes current tl
  | Expr(expr)::tl -> ignore (exprType (List.nth scopes current) scopes expr) ;
      makeScope scopes current tl
  | Assign(expr, value)::tl ->
      (match expr with
        Id(_) | Dot(_, _) | Subscript(_, _) ->
          let t1 = exprType (List.nth scopes current) scopes expr
          in let t2 = exprType (List.nth scopes current) scopes value
          in if isComp (List.nth scopes current) scopes expr
          then raise (Access_error ("Cannot assign components directly"))
          else if compatible (t1, t2)
          then makeScope scopes current tl
          else raise (Type_error ("Cannot assign " ^ Strings.of_type t2 ^
            " to " ^ Strings.of_type t1))
      | _ -> raise (Type_error "Expression is not assignable"))
  | If(expr, (ind1, stmts1), (ind2, stmts2))::tl ->
      ignore (exprType (List.nth scopes current) scopes expr) ;
      let scopes = makeScope (addScope scopes ind1 (Some current) None None None)
ind1 stmts1
      in let scopes = makeScope (addScope scopes ind2 (Some current) None None None)
ind2 stmts2
      in makeScope scopes current tl
  | DoWhile ((ind, stmts), expr)::tl
  | While(expr, (ind, stmts))::tl ->
      ignore (exprType (List.nth scopes current) scopes expr) ;
```

```
      let scopes = makeScope (addScope scopes ind (Some current) None None None) ind
stmts
      in makeScope scopes current tl
  | Return(e)::tl ->
      (* Returns type of current function *)
      let rec funcType scope scopes = match (List.nth scopes scope).typ with
        Some(typ) -> typ
      | None -> match (List.nth scopes scope).outer with
          None -> raise (Access_error "Return statements must be inside functions")
        | Some(outer) -> funcType outer scopes in
      let t = funcType current scopes in match e, t with
        Some(Function(rt, args, (fIndex, stmts))), t
          when (exprType (List.nth scopes current) scopes (Function(rt, args,
(fIndex, stmts)))) = t ->
            let scopes = makeFunctionScope scopes current rt args fIndex stmts in
            makeScope scopes current tl
      | Some(e), t when (exprType (List.nth scopes current) scopes e) = t ->
makeScope scopes current tl
      | None, VoidT -> makeScope scopes current tl
      | None, t -> raise (Type_error ("Function must return " ^ Strings.of_type t ^ "
not void"))
      | Some(e), t -> raise (Type_error ("Function must return " ^ Strings.of_type t
^ " not " ^
          Strings.of_type (exprType (List.nth scopes current) scopes e)))
  and parseDecl scopes index = function
    Comp, sym, None, Some(expr) ->
    (match exprType (List.nth scopes index) scopes expr with
      FigureT(f) -> modSymbols addSymbol sym (FigureT f) scopes index
    | _ -> raise (Type_error ("Component " ^ sym ^
      " must be of figure type.")))
  | Comp, sym, Some(FigureT(f)), None -> modSymbols addSymbol sym (FigureT f) scopes
index
  | Comp, sym, _, None -> raise (Type_error ("Component " ^ sym ^
      " must be of figure type."))
  | _, sym, Some(typ), None -> modSymbols addSymbol sym typ scopes index
  | _, sym, None, Some(expr) ->
    let typ = (exprType (List.nth scopes index) scopes expr)
    in let rec isEmpty = function VoidT -> true | ListT(l) -> isEmpty l | _ -> false
    in let scopes = if isEmpty typ then raise (Type_error "Cannot infer type of empty
list")
      else modSymbols addSymbol sym typ scopes index
    in (match expr with
      Function(t, args, (fIndex, stmts)) -> makeFunctionScope scopes index t args
fIndex stmts
    | _ -> scopes)
  | _, sym, _, _ -> raise Internal_error
  and makeFunctionScope scopes index t args fIndex stmts =
    let scopes = List.fold_left
```

```
      (fun scopes (sym, typ) -> modSymbols addSymbol sym typ scopes fIndex)
      (addScope scopes fIndex (Some index) None (Some StringMap.empty) (Some t)) args
    in let scopes = makeScope scopes fIndex stmts
    in makeCaptures scopes fIndex stmts
  in makeScope (initScopeList [] (!Ast.numScopes + 1)) 0 sList
```

8.6 - translate.ml

```
(* translate.ml takes the AST and a scopes list and is responsible for
    java code generation *)

open Ast
open Strings
open Analyze
open Printf
open Sys
open Programname

exception Internal_error of string

(* write a string to file *)
let dir = Programname.get_name ^ "_java/"

let write str file =
    if (Sys.file_exists (dir ^ "GeneratedCode/" ^ file))
      then
      ()
      else
    let oc = open_out (dir ^ "GeneratedCode/" ^file) in
      fprintf oc "%s\n" str;
      close_out oc


(* list of captures *)
let captures = ref []
(* list of stdlib *)
let stdlib = ref []

let header = "package GeneratedCode;\n\nimport Stdlib.*;\nimport Type.*;\nimport
Main.*;\nimport BuiltIns.*;\n\n"
let seperator = "-----------------------------------------------"

(* prefix on all user-defined identifiers *)
let id_prefix = ""

(* for unnamed functions *)
let arbitrary_function_counter = ref (-1)

(* formats a string list *)
```

```
let tr_list l = List.fold_left (fun s x -> s ^ x ^
  (let c = x.[(String.length x) - 1] in
       if c = ';' || c = '}' || c = '\n'
       then ""
    else ";\n")) "" l

let tr_list2 = function
      [] -> ""
    | hd::tl -> (List.fold_left (fun s x -> s ^ ", " ^ x) hd tl)

let tr_list3 l = List.fold_left (fun s x -> s ^ x) "" l

let helper = (fun a b d -> (match d with
                (a2, b2) -> (a2 = a) || (b2 = b)))

let helper3 = fun s x -> match s with
       (a, b) -> match x with
                 ((c, d),e) -> (a=c)&&(b=d)

let is_captured search_cap = List.exists (helper3 search_cap) !captures
let in_stdlib symbol = List.exists (fun a -> a = symbol) !stdlib

(* finds the index of a symbol *)
let rec scope_of_symbol symbol cur_scope scopes =
    let scope = (try List.nth scopes cur_scope with Invalid_argument(str)
      -> raise (Internal_error ("Scope index out of bounds.")))


    in

    if (StringMap.exists (fun a b -> a = symbol) scope.symbols) then
      cur_scope
    else
      match scope.capture with
        None ->  (match scope.outer with
                    None -> raise (Internal_error ("Scope not found"))
                  | Some(num) -> scope_of_symbol symbol num scopes)
      | Some(map) ->
          try (StringMap.find symbol map) with
          Not_found -> (match scope.outer with
                    None -> raise (Internal_error("Scope not found"))
                  | Some(num) -> scope_of_symbol symbol num scopes)

(* finds the type of a symbol *)
let rec type_of_symbol symbol cur scopes =
       let containingScope = (
           try List.nth scopes cur with Invalid_argument(str) ->
             raise (Internal_error ("Scope not found in type_of_symbol"))) in
```

```
          try (
             (StringMap.find symbol containingScope.symbols)
          ) with Not_found -> match containingScope.outer with
               None -> raise Not_found
             | Some(num) -> type_of_symbol symbol num scopes

(* ignores the next expr *)
let ignore_expr expr = ""

(* am i parsing the std lib? *)
let stdlib_parse = ref true

(*captures helper class*)
let captures_helper search_cap =
             let index =
                 match (List.find (helper3 search_cap) !captures)
                 with (a, b) -> b
             in
             "Captures.record[" ^ (string_of_int index) ^ "]"


(* global record of figures *)
let figure_record = ref StringMap.empty

(* translate a typ *)
let rec tr_typ = function
      IntT -> "IntType"
    | FloatT -> "FloatType"
    | StringT -> "StringType"
    | BoolT -> "BoolType"
    | ColorT -> "ColorType"
    | PairT -> "PairType"
    | FigureT(arg) -> (match arg with
                     None -> "Figure"
                   | Some(str) -> str)
    | VoidT -> "void"
    | FigDefT -> "FigDefT"
    | ListT(typ) -> "ListType<" ^ (tr_typ typ)  ^ ">"
    | FuncT (typ, typ_list) -> "FunctionType"

let rec type_of_expr expr cur scopes = match expr with
      Id(str) -> tr_typ (type_of_symbol str cur scopes)
    | Int(_) -> "IntType"
    | Float(_) -> "FloatType"
    | String(_) -> "StringType"
    | Bool(_) -> "BoolType"
    | Function(_,_,_) -> "FunctionType"
    | List(_) -> "ListType"
```

```
    | Color(_) -> "ColorType"
    | Pair(_,_) -> "PairType"
    | FigInstance(_,_) -> "FigInstance"
    | Subscript(_,_) -> "Subscript"
    | Dot(_,_) -> "Dot"
    | Arop(_,_,_) -> "Arop"
    | Logop (_, _, _) -> "Logop"
    | Unop (_, _) -> "Unop"
    | Call(_, _) -> "Call"

let rec type_of_expr2 expr cur scopes = match expr with
      Id(str) -> "Id"
    | Int(_) -> "IntType"
    | Float(_) -> "FloatType"
    | String(_) -> "StringType"
    | Bool(_) -> "BoolType"
    | Function(_,_,_) -> "FunctionType"
    | List(_) -> "ListType"
    | Color(_) -> "ColorType"
    | Pair(_,_) -> "PairType"
    | FigInstance(_,_) -> "FigInstance"
    | Subscript(_,_) -> "Subscript"
    | Dot(_,_) -> "Dot"
    | Arop(_,_,_) -> "Arop"
    | Logop (_, _, _) -> "Logop"
    | Unop (_, _) -> "Unop"
    | Call(_, _) -> "Call"


(* translate an expression into java *)
let rec tr_expr expr scopes cur =

       let tr_arop arop arg1 arg2 sc cur = match arop with
             Plus -> "Arop.fogl_plus(" ^
              (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
           | Minus -> "Arop.fogl_minus("
                ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
           | Times -> "Arop.fogl_times("
                ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
           | Div -> "Arop.fogl_div("
                ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
           | Divdiv -> "Arop.fogl_divdiv("
                ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
           | Exp -> "Arop.fogl_exp("
                ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
           | Percent -> "Arop.fogl_percent("
                ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
       in
```

```
      let tr_logop logop arg1 arg2 sc cur = match logop with
            Eqeq -> "Arop.fogl_eqeq("
            ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
          | Neq -> "Arop.fogl_neq("
            ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
          | Gt -> "Arop.fogl_gt("
            ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
          | Lt -> "Arop.fogl_lt("
            ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
          | Gte -> "Arop.fogl_gte("
            ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
          | Lte -> "Arop.fogl_lte("
            ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
          | And -> "Arop.fogl_and("
            ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
          | Or -> "Arop.fogl_or("
            ^ (tr_expr arg1 sc cur) ^ ", " ^ (tr_expr arg2 sc cur) ^ ")"
      in
      let tr_unop unop arg sc cur = match unop with
            Neg -> "Arop.fogl_neg(" ^ (tr_expr arg sc cur) ^ ")"
          | Not -> "Arop.fogl_not(" ^ (tr_expr arg sc cur) ^ ")"
          | Question -> "Arop.fogl_question(" ^ (tr_expr arg sc cur) ^ ")"
      in

      let tr_figinstance str stuff scopes cur =
          let fig_cons = StringMap.find str !figure_record
          in

          (*
          (let liststuff = List.map (fun x -> match x with
                              (a,b) -> a ^ " " ^ (tr_expr b scopes )  )
                      stuff
          in

          let constructors = tr_list2 liststuff in
          *)
          let hef = fun x y -> match y with
                              (c, d) -> c = x
          in

          let elt = fun x -> (List.find (hef x) stuff)
          in
          let constructors = List.fold_left (fun a b -> a ^ (if a = "" then ""
else ", ") ^ (
              try
              (match (elt b) with
                  (xx, yy) -> tr_expr yy scopes cur) with
              Not_found -> "null"))
```

```
            "" (List.rev fig_cons)
          in
          if (constructors = "null" ) then
            "new " ^ str ^ "((AbstractType)" ^ constructors ^ ")"
          else
            "new " ^ str ^ "(" ^ constructors ^ ")"

      in
    match expr with
        Int(arg) -> "new IntType(" ^ (string_of_int arg) ^ ")"
      | Float(arg) -> "new FloatType(" ^ (string_of_float arg) ^ ")"
      | String(arg) -> "new StringType(\"" ^ arg ^ "\")"
      | Bool(arg) -> "new BoolType(" ^ (string_of_bool arg) ^ ")"
      | Function(a, b, c) -> "THIS SHOULD NEVER BE CALLED."
      (*
            let func_name =
                arbitrary_function_counter :=
                  !arbitrary_function_counter + 1;
                "Func_" ^ (string_of_int cur) ^ "_" ^
                  (string_of_int !arbitrary_function_counter)
            in
            let tr_arglist arglist =
                let tr_s x = match x with
                (str, typ) -> (tr_typ typ) ^ " " ^ str
            in
                List.map (fun x -> tr_s x) arglist
                in
            let super_str =
                (header ^"public class " ^
                  func_name ^ " extends FunctionType {\n"
                  ^ (tr_typ a) ^ " call(" ^
                  (tr_list2 (tr_arglist b)) ^ ") {\n"
                  ^ (tr_block c scopes cur)
                  ^ "}\n}\n"
                )
            in
            write super_str ("Func_" ^ str ^ ".java");
            " new " ^ func_name ^ "()"
*)
      | List(arg) -> "new ListType(" ^
          (List.fold_left
            (fun str x -> if (str = "") then
                str ^ (tr_expr x scopes cur) else
                str ^ ", " ^ (tr_expr x scopes cur)) "" arg)
          ^ ")"
      | Color(arg) -> (match arg with
           (s1, s2, s3, fl) -> ("new ColorType(0x" ^
```

```
            s1 ^ ", 0x" ^ s2 ^ ", 0x" ^ s3 ^
            ", " ^ (string_of_float fl)^ ")") )
      | Pair(arg1, arg2) -> "new PairType(" ^ (tr_expr arg1 scopes cur) ^ ", "
                          ^ (tr_expr arg2 scopes cur) ^ ")"
      | Subscript(arg1, arg2) -> "(" ^ (tr_expr arg1 scopes cur)
            ^ ").get(" ^ (tr_expr arg2 scopes cur ) ^ ")"
      | Dot(arg1, arg2) -> "(" ^ (tr_expr arg1 scopes cur) ^ ")." ^ arg2 ^ ""
      | Id(arg1) ->
          (* is_captured arg1, cur is not returning true *)
          let arg_scope_index =
            scope_of_symbol arg1 cur scopes
          in
          if arg_scope_index = 9001
            then (print_endline "OVER 9000 ERROR";"") else
          if (is_captured (arg1, arg_scope_index)) then
            "(("^ (tr_typ (type_of_symbol arg1 cur scopes)) ^")"^
            captures_helper (arg1, arg_scope_index) ^ ")"
          else
            if ((in_stdlib arg1)&&(arg_scope_index = 0)) then
                ("Stdlib.Lib." ^ arg1)
            else
                (id_prefix ^ arg1)
      | Arop(arg1, arop, arg2) -> tr_arop arop arg1 arg2 scopes cur
      | Logop(arg1, logop, arg2) -> tr_logop logop arg1 arg2 scopes cur
      | Unop(unop,arg) -> tr_unop unop arg scopes cur
      | Call(arg1, arg2) ->
          let arglist = List.map (fun x -> tr_expr x scopes cur) arg2 in
          (tr_expr arg1 scopes cur) ^ ".call(" ^ (tr_list2 arglist) ^")"
      | FigInstance(arg1, arg2) -> tr_figinstance arg1 arg2 scopes cur

(* BEWARE! THE FOLLOWING IS ALMOST IMPOSSIBLE TO READ. *)

(* translate a statement into java *)
let rec tr_stmt stmt scopes cur =

    let tr_block block scopes cur=
      (let almost =
          (match block with
            (arg1, arg2) -> List.map
            (fun x -> tr_stmt x scopes arg1) arg2)
      in tr_list almost)
    in

(* translate a list of decl *)
let tr_decls decls scopes cur =

    let tr_typ_opt arg str scopes cur = match arg with
        None -> let containingScope = (
```

```
          try List.nth scopes cur with Invalid_argument(str) -> raise
            (Internal_error ("Scope not found in tr_typ_opt"))) (* this is where
you search teh scopes list for the type*)

          in
          tr_typ (StringMap.find str containingScope.symbols)
       | Some(typ) -> tr_typ typ
    in

    let tr_expr_opt arg scopes cur = match arg with
         None -> ""
       | Some(expr) -> " = " ^ (tr_expr expr scopes cur)

    in
   let tr_decl decl scopes cur = (match decl with
      (decl_typ, str, typ_opt, expr_opt) ->

      if is_captured (str, cur) then
          let other = (tr_expr_opt expr_opt scopes cur) in
          if other = "" then "" else
               (captures_helper (str, cur) ^
               other)
      else
      (
      match decl_typ with
        Var ->
          (match expr_opt with
            None ->
             ("" ^ (tr_typ_opt typ_opt str scopes cur) ^
             " " ^ id_prefix ^ str ^ (tr_expr_opt expr_opt scopes cur)) ^
             (match typ_opt with Some(ListT(_)) -> " = new ListType()" | _ -> "")
          | Some(expr) ->
          (match expr with
            Function(a, b, c) ->
             let tr_arglist arglist =
                 let tr_s x = match x with
                 (str, typ) -> (tr_typ typ) ^ " " ^ str
             in
                 List.map (fun x -> tr_s x) arglist
                 in

             (match cur with
             0 ->
             (* function with no scope *)
             let super_str =
                 (header ^"class Func_" ^
                    str ^ " extends FunctionType {\n"
```

```
                    ^ "public static Func_" ^ str ^ " " ^ str ^ " = new Func_" ^ str
^ "();\n"
                    ^ "public " ^ (tr_typ a) ^ " call(" ^
                    (tr_list2 (tr_arglist b)) ^ ") {\n"
                    ^ (tr_block c scopes cur)
                    ^ "}\n}\n"
                )
            in
            write super_str ("Func_" ^ str ^ ".java");
            "Func_" ^ str ^ " " ^ str ^ " = new Func_" ^ str ^ "()"
            | _ ->
                "THIS SHOULD NEVER BE CALLED EITHER"
            (* function in a figure. this should never be called *)
            (*
            (tr_typ a) ^ " " ^ str ^ "("
            ^ (tr_list2 (tr_arglist b)) ^
            ") {\n" ^ (tr_block c scopes cur) ^ "}\n"
            ) *))
          | _ -> "" ^ (tr_typ_opt typ_opt str scopes cur) ^
            " " ^ id_prefix ^ str ^ (tr_expr_opt expr_opt scopes cur )
      )
            )
      | Param -> "private " ^ (tr_typ_opt typ_opt str scopes cur)
          ^ " " ^ id_prefix ^ str ^ (tr_expr_opt expr_opt scopes cur)
      | Comp -> "private " ^ (tr_typ_opt typ_opt str scopes cur)
          ^ " " ^ id_prefix ^ str ^ (tr_expr_opt expr_opt scopes cur)
      )



            )


    in
       let almost = (List.map (fun x -> tr_decl x scopes cur ) decls)
       in tr_list almost

in

(* ----------------------- FIGURE TRANSLATION ----------------------- *)

let tr_figdecls decls scopes cur =

    let tr_typ_opt arg str scopes cur = match arg with
        None -> let containingScope = (try List.nth scopes cur with
          Invalid_argument(str) -> raise (Internal_error ("scopes not valid in
tr_typ_opt")))
         (* this is where you search teh scopes list for the type*)
```

```
      in
      tr_typ (StringMap.find str containingScope.symbols)
  | Some(typ) -> tr_typ typ
in

let tr_expr_opt arg scopes cur = match arg with
    None -> ""
  | Some(expr) -> " = " ^ (tr_expr expr scopes cur)

in

let tmp_init = ref "" in
let tmp_comp = ref "" in
let tmp_binit = ref "" in

let tr_figdecl decl scopes cur = (match decl with
  (decl_typ, str, typ_opt, expr_opt) ->

  if is_captured (str, cur) then
      let other = tr_expr_opt expr_opt scopes cur in
      if other = "" then "" else
        (
        (*
           let mask = tr_expr_opt expr_opt scopes cur in
           let mask2 = String.sub mask 2 ((String.length mask) -2)
           in
           tmp_init := !tmp_init ^
           (captures_helper (str, cur) ^
           " = (AbstractType) (" ^ mask2 ^ ")") ^ ";\n";
           ""
           *)
            tmp_binit := !tmp_binit ^
           (captures_helper (str, cur) ^
           (tr_expr_opt expr_opt scopes cur)) ^ ";\n";
           ""

        )
  else

  (

  match decl_typ with
    Var ->
      (match expr_opt with
        None ->  "" ^ (tr_typ_opt typ_opt str scopes cur) ^ " "
         ^ id_prefix ^ str ^ (tr_expr_opt expr_opt scopes cur)
      | Some(expr) ->
         (match expr with
```

```
          Function(a, b, c) ->
            let tr_arglist arglist =
              let tr_s x = match x with
              (str, typ) -> (tr_typ typ) ^ " " ^ str
            in
              List.map (fun x -> tr_s x) arglist
            (* in
            function in a figure
            let func = (*(tr_typ a) ^ " " ^ str ^ "(" ^
            (tr_list2 (tr_arglist b)) ^ ") {\n" ^*)
            (tr_block c scopes cur) (*^ "}\n"*)*)

            in let func_name =
        "Func_" ^ str ^ (string_of_int cur)

            in let func_cls = header ^ "class " ^ func_name ^

              (if (str = "_update") then
                  " extends UpdateFunctionType {\n" else
                    " extends FunctionType {\n" )

              ^ "public " ^ (tr_typ a)
              ^ " call(" ^
              (tr_list2 (tr_arglist b)) ^ ") {\n"
              ^ (tr_block c scopes cur)
              ^ "}\n}\n"
            in
            write func_cls (func_name ^ ".java");

            let superstr =
            (if (str = "_update") then
              "_update = (UpdateFunctionType) (new "
              ^ func_name ^ "())"
            else
              "public " ^ func_name ^ " " ^ str ^ " = new " ^ func_name ^
              "()")
            in
            tmp_binit := !tmp_binit ^ superstr ^ ";\n";
            ""

        | _ -> "" ^ (tr_typ_opt typ_opt str scopes cur) ^
            " " ^ id_prefix ^ str ^ (tr_expr_opt expr_opt scopes cur )
        )
      )
  | Param ->
      (
      match (expr_opt)  with
        None -> "" ^ (tr_typ_opt typ_opt str scopes cur)
```

```
      ^ " " ^ id_prefix ^ str ^
      (tr_expr_opt expr_opt scopes cur)
    | Some(expr) ->
      (match expr with
        Function(a, b, c) ->
          let tr_arglist arglist =
            let tr_s x = match x with
            (str, typ) -> (tr_typ typ) ^ " " ^ str
          in
            List.map (fun x -> tr_s x) arglist
          (* in
          function in a figure
          let func = (*(tr_typ a) ^ " " ^ str ^ "(" ^
          (tr_list2 (tr_arglist b)) ^ ") {\n" ^*)
          (tr_block c scopes cur) (*^ "}\n"*)*)

          in let func_name =
            "Func_" ^ str ^ (string_of_int cur)

          in let func_cls = header ^ "class " ^ func_name ^
            " extends FunctionType {\n"
            ^ "public " ^ (tr_typ a)
            ^ " call(" ^
            (tr_list2 (tr_arglist b)) ^ ") {\n"
            ^ (tr_block c scopes cur)
            ^ "}\n}\n"
          in
          write func_cls (func_name ^ ".java");

          let superstr =
          ("" ^ "FunctionType" ^
            " " ^ str ^ " = new " ^ func_name ^
            "()")
          in
        (*  tmp_init := !tmp_init ^ superstr ^ ";\n";
        *)
          superstr

      | _ -> "" ^ (tr_typ_opt typ_opt str scopes cur)
      ^ " " ^ id_prefix ^ str ^
      (tr_expr_opt expr_opt scopes cur)
      )
    )
| Comp ->
    (* add str to components list *)
    tmp_comp := (!tmp_comp ^ "components.add(" ^ (str) ^ ");\n");
    (* add the following to comp_init *)
    tmp_init := (!tmp_init ^ str
```

```
                ^(tr_expr_opt expr_opt scopes cur)^";\n");
          (* return the following as the code *)
          ("" ^ (tr_typ_opt typ_opt str scopes cur)
          ^ " " ^ id_prefix ^ str)

            )
      )

    in
      let almost = (List.map (fun x -> tr_figdecl x scopes cur) decls)
      in

      (tr_list almost) ^ ("void first_init() {\n" ^ !tmp_comp ^ "}\n")
      ^ ("void create() {\n" ^ !tmp_init ^ "}\n" ^
      "void b_init() {\n"^ !tmp_binit ^ "}\n")

in

(* translates a figure into java and produces a .java file *)
let tr_figure name decls scopes cur captures =

    (* -------------------- CREATE THE CONSTRUCTOR ------------------ *)
    let create_constructor name decls scopes cur =

      (* helper function will return a strmap of param -> int index *)
      let helper decl lst =
          match decl with
          (decl_typ, str, typ_opt, expr_opt) ->
            (
                match decl_typ with
                  Param ->
                   str::lst
                | _ -> lst
            )

      in

      let counter = ref (-1) in

      let giant_lst =
          List.fold_left (fun a b -> helper b a) [] decls
      in
      figure_record := StringMap.add name giant_lst !figure_record;
      let giant_stmt = List.fold_left
          (fun a b ->
            let scope_b = scope_of_symbol b cur scopes in
            counter := !counter + 1;
```

```
            a ^ "if (arg[" ^ (string_of_int !counter)
            ^ "] != null) { " ^
            (if (is_captured (b, scope_b)) then
            (captures_helper (b, scope_b))
            else b)
            ^ " = ("
            ^ (tr_typ (type_of_symbol b cur scopes)) ^ ") arg[" ^
            (string_of_int !counter)^ "]; }\n"
          )
          "" (List.rev giant_lst)
      in
      "public " ^ name ^ "(AbstractType... arg) {\nb_init();\n" ^
      giant_stmt ^
      "create();\nfirst_init();\nif (_update != null)\n_animating =true;\n}"
    in

    let constructor = create_constructor name decls scopes cur in

    let strstr =
      header ^ "public class " ^ name ^ " extends Figure {\n" ^
      (tr_figdecls decls scopes cur) ^ constructor ^ "\n}\n"

    in
    write strstr (name ^ ".java");

    "//Class " ^ name ^ " created."

in
(* stdlib translation *)
    let tr_stdlib_stmt stmt =
      let tr_stdlib_decl decl =
      match decl with (decl_typ , str , typ_opt , expr_opt) ->
          (match expr_opt with
              None ->
                let typ = (match (typ_opt) with
                     None -> "Very big problem."
                   | Some(typ) -> tr_typ typ)
                in
                if (is_captured (str, 0)) then
                   (captures_helper (str, 0)) ^ " = Stdlib.Lib." ^ str ^
                       ";\n"
                else
                   (stdlib := str::!stdlib;
                   typ ^ " " ^  str ^ " = Stdlib.Lib." ^ str ^ ";\n"
                   )
            | Some(expr) -> "Big problem as well."
          )
```

```
      in (match stmt with
            Decl(decls) -> List.fold_left
             (fun x y -> x ^ tr_stdlib_decl y) "" decls
          | _ -> "BIG PROBLEM"
          )

in
    match stmt with
      Expr(expr) -> tr_expr expr scopes cur
    | Figure(arg1, arg2, decls) -> tr_figure arg2 decls scopes arg1 cur
    | Decl(arg) ->
      if (!stdlib_parse) then
             (stdlib_parse := false;
             tr_stdlib_stmt stmt)
          else tr_decls arg scopes cur
    | Assign(arg1, arg2) ->
      if ( ((type_of_expr arg1 cur scopes) = "FloatType") &&
             ((type_of_expr arg2 cur scopes) = "IntType"))
      then
          (tr_expr arg1 scopes cur) ^
          " = new FloatType(" ^ (tr_expr arg2 scopes cur)^ ")"
      else

      (if ( (type_of_expr arg2 cur scopes) = "FunctionType")

      then
          (match arg2 with
          Function(a,b,c) ->
             let func_name =
                 arbitrary_function_counter :=
                    !arbitrary_function_counter + 1;
                 "Func_" ^ (string_of_int cur) ^ "_" ^
                    (string_of_int !arbitrary_function_counter)
             in
             let tr_arglist arglist =
                 let tr_s x = match x with
                 (str, typ) -> (tr_typ typ) ^ " " ^ str
             in
                 List.map (fun x -> tr_s x) arglist
                 in
             let super_str =
                 (header ^"public class " ^
                    func_name ^ " extends FunctionType {\n"
                    ^ (tr_typ a) ^ " call(" ^
                    (tr_list2 (tr_arglist b)) ^ ") {\n"
                    ^ (tr_block c scopes cur)
```

```
            ^ "}\n}\n"
          )
      in
      write super_str (func_name ^ ".java");
      " new " ^ func_name ^ "()"

    | _ -> "This is purely a physics problem.")
else
(if ( ((type_of_expr arg1 cur scopes)) = "Subscript")
    then
    (match arg1 with
        Subscript(a1, a2) ->
          "(" ^ (tr_expr a1 scopes cur)
          ^ ").set(" ^(tr_expr a2 scopes cur) ^", "^
          (tr_expr arg2 scopes cur ) ^ ")"

      | _ -> "Another pure physics problem.")

else


(if (type_of_expr2 arg1 cur scopes)="Id" then
(match arg1 with
    Id(a1) ->
      (* is_captured arg1, cur is not returning true *)
      let arg_scope_index =
          scope_of_symbol a1 cur scopes
      in
      if arg_scope_index = 9001
          then (print_endline "OVER 9000 ERROR";"") else
      if (is_captured (a1, arg_scope_index)) then
          captures_helper (a1, arg_scope_index)
              ^ " = " ^
            (tr_expr arg2 scopes cur)


      else
          if ((in_stdlib a1)&&(arg_scope_index = 0)) then
            ("Stdlib.Lib." ^ a1) ^ " = " ^
            (tr_expr arg2 scopes cur)

          else
            (tr_expr arg1 scopes cur) ^ " = " ^
            (tr_expr arg2 scopes cur)

    | _ -> "too many laws of physics were broken")
```

```
  else (tr_expr arg1 scopes cur) ^ " = " ^
      (tr_expr arg2 scopes cur)
      )))
| Return(arg) ->
      (match arg with
          None -> "return;"
        | Some(ex) ->

  (if ((type_of_expr ex cur scopes) = "FunctionType")

  then
      (match ex with Function(a,b,c) ->
        let func_name =
            arbitrary_function_counter :=
               !arbitrary_function_counter + 1;
            "Func_" ^ (string_of_int cur) ^ "_" ^
               (string_of_int !arbitrary_function_counter)
        in
        let tr_arglist arglist =
            let tr_s x = match x with
            (str, typ) -> (tr_typ typ) ^ " " ^ str
        in
            List.map (fun x -> tr_s x) arglist
            in
        let super_str =
            (header ^"public class " ^
               func_name ^ " extends FunctionType {\n"
               ^ (tr_typ a) ^ " call(" ^
               (tr_list2 (tr_arglist b)) ^ ") {\n"
               ^ (tr_block c scopes cur)
               ^ "}\n}\n"
            )
        in
        write super_str (func_name ^ ".java");
        "return new " ^ func_name ^ "()"
      | _ -> "THIS is purely a physic problem 2")
  else
      "return " ^ (tr_expr ex scopes cur) ^ ""))
| If(arg1, arg2, arg3) -> "if ((" ^
  (tr_expr arg1 scopes cur) ^ ").value) {\n" ^
  (tr_block arg2 scopes cur) ^ "\n} else {\n" ^
  (tr_block arg3 scopes cur) ^ "\n } \n"
| While(arg1, arg2) -> "while ((" ^
  (tr_expr arg1 scopes cur) ^").value) {\n"
  ^ (tr_block arg2 scopes cur) ^ "\n}\n"
| DoWhile(arg1, arg2)  -> "do {\n" ^
  (tr_block arg1 scopes cur) ^ "\n} while (("
  ^ (tr_expr arg2 scopes cur) ^ ").value)\n"
```

```
(* no bindings work as of now =(  *)

(* ------------------------- BINDINGS --------------------------------*)

let analyze_bindings_decl decl s_num bindings_list =
    match decl with
      (decl_typ, str, typ_opt, expr_opt) -> "" (* TODO: BINDINGS LIST *)

let analyze_bindings_decls decls s_num bindings_list =
    List.fold_left (fun a b -> analyze_bindings_decl b s_num a)
      bindings_list decls

let analyze_bindings_stmt stmt scopes bindings_list =
    match stmt with
        Figure(num,str,decls) ->  bindings_list (* TODO: BINDINGS LIST *)
      | _ -> bindings_list

let create_bindings_class program scopes = ""

(* ------------------------- CAPTURES --------------------------------*)

(* add captures to the list *)
let analyze_captures_in_scope scope captures_list =
    let helper = (fun a b d ->
             (match d with
                 (a2, b2) ->
                   (a2 = a) && (b2 = b)))
    in
    match scope.capture with
        None ->
          captures_list
      | Some(strmap) ->
          StringMap.fold
          (fun a b c ->
            if (List.exists (helper a b) c)
                then c
                else ((a,b)::c))
          strmap captures_list

let analyze_captures_in_scopes scopes captures =
    List.fold_left (fun a b -> (analyze_captures_in_scope b a) ) captures scopes

(* format captures for debug printing *)
let of_captures captures =
    let print_capture x = match x with ((a,b), c) ->
      ( "[(" ^ a ^ ", " ^
          (string_of_int b) ^ ") -> " ^ (string_of_int c) ^ "]")
```

```
    in
    let to_print = "Captures: " ^ List.fold_left (fun x c ->
      x ^(if x="" then "" else ", ") ^(print_capture c))
      "" captures
    in to_print


let get_captures scopes =
    let all_the_captures = analyze_captures_in_scopes scopes []
    in
    let index = ref (-1) in

    let result = (List.fold_left
      (fun lst c ->
          index := !index + 1;
          (c,!index)::lst)
      [] all_the_captures)
    in
(*    (print_endline (of_captures result));
*)    result
(* each 'capture' is a ((variable name, scope), index) *)


(* translate a program into java *)
let tr_program program scopes current =
    captures := get_captures scopes;
    let almost = List.map (fun x -> tr_stmt x scopes current) program
    in

    let almost_main_fun =
      header ^ "public abstract class GeneratedMain {\n" ^
          "public static void generatedMain() {" ^
          "Captures.record = new AbstractType["
            ^ (string_of_int(List.length !captures)) ^ "];" ^
          tr_list almost ^
          "}\n}\n"
    in
      write almost_main_fun "GeneratedMain.java"
```

8.7 - programname.ml

```
(* figures out the program name *)

open Sys
open Array
open String

let get_name =
```

```
    let somewhat = (Array.get Sys.argv 2) in
    let last_index = String.rindex somewhat '/' in
    let length_sub = (String.length somewhat) -5-last_index-1
    in
    let name = String.sub somewhat (last_index+1) length_sub
    in name
```

8.8 - runtests.py

```
import os, subprocess, re
dir = 'tests/print/'
result = ''
files = filter(lambda x: bool(re.match(r'.*\.fogl', x)), os.listdir(dir))
for f in map(lambda x: re.sub(r'\.fogl$', '', x), files):
      os.system('./fogl -c ' + dir + f + '.fogl')
      p1 = subprocess.Popen('java -jar ' + f + '.jar', shell=True,
stdout=subprocess.PIPE)
      result += f + ': ' + 'Passed\n' if open(dir + f +'.out', 'r').read() + '' ==
p1.stdout.read() else f + ': Failed\n'
os.system('rm -rf *_java *.jar')
open('Test Output', 'w').write(result)
```

8.9 - Java Boilerplate

```
// BuiltIns/ArcFigure.java
package BuiltIns;

import java.awt.*;
import Type.*;
import Main.*;
import java.awt.geom.Arc2D;

public class ArcFigure extends Figure {

      IntType radius, lineWidth;
      FloatType start, end;
      BoolType fill, clockwise;

      /* Constructor */
      public ArcFigure(AbstractType... args) {
            if (args[0] != null) {
                  loc = (PairType) args[0];
            }

            if (args[1] != null) {
                  radius = (IntType) args[1];
            }
            else throw new Error("Radius must be specified.");

            if (args[2] != null) {
```

```
                start = (FloatType) args[2];
            }
            else start = new FloatType(0);

            if (args[3] != null) {
                end = (FloatType) args[3];
            }
            else end = new FloatType(2*Math.PI);

            if (args[4] != null) {
                color = (ColorType) args[4];
            }

            if (args[5] != null) {
                fill = (BoolType) args[5];
            }
            else fill = new BoolType(true);

            if (args[6] != null) {
                lineWidth = (IntType) args[6];
            }
            else lineWidth = new IntType(1);

            if (args[7] != null) {
                clockwise = (BoolType) args[7];
            }
            else clockwise = new BoolType(true);
        }

        public void draw(Graphics g) {
            Graphics2D g2 = (Graphics2D) g;
            g2.setStroke(new BasicStroke(lineWidth.value));
            g2.setColor(color.getColor());
            Arc2D.Float a = new Arc2D.Float();
            if (clockwise.value) {
                a = new Arc2D.Float(loc.x.value, loc.y.value, radius.value,
radius.value, (float) (180/Math.PI)*end.value, (float) (180/Math.PI)*(end.value -
start.value), Arc2D.OPEN);
            }
            else {
                a = new Arc2D.Float(loc.x.value, loc.y.value, radius.value,
radius.value, (float) (180/Math.PI)*start.value, (float) (180/Math.PI)*(end.value -
start.value), Arc2D.OPEN);
            }
            if (_visible) {
                g2.draw(a);
                if (fill.value) {
```

```
                        g2.fill(a);
                }
            }
        }
}
// BuiltIns/PolyFigure.java
package BuiltIns;

import java.awt.*;
import java.awt.geom.Path2D;
import Type.*;
import Main.*;

public class PolyFigure extends Figure {

        ListType points;
        IntType lineWidth;
        BoolType fill;

        /* Constructor */
        public PolyFigure(AbstractType... args) {
            if (args[0] != null) {
                loc = (PairType) args[0];
            }

            if (args[1] != null) {
                points = (ListType) args[1];
            }
            else throw new Error("Points must be specified.");

            if (args[2] != null) {
                color = (ColorType) args[2];
            }

            if (args[3] != null) {
                fill = (BoolType) args[3];
            }
            else fill = new BoolType(true);

            if (args[4] != null) {
                lineWidth = (IntType) args[4];
            }
            else lineWidth = new IntType(1);
        }

        public void draw(Graphics g) {
            Graphics2D g2 = (Graphics2D) g;
```

```
            g2.setStroke(new BasicStroke(lineWidth.value));
            g2.setColor(color.getColor());

            Path2D.Float poly = new Path2D.Float();
            boolean isFirst = true;
            for (int i = 0; i < points.size(); ++i) {
                  PairType v = (PairType) points.get(i);
                  if (isFirst) {
                        poly.moveTo(v.x.value + loc.x.value, v.y.value +
loc.y.value);
                        isFirst = false;
                  }
                  else {
                        poly.lineTo(v.x.value + loc.x.value, v.y.value +
loc.y.value);
                  }
            }
            poly.closePath();

            g2.draw(poly);
            if (fill.value) {
                  g2.fill(poly);
            }
      }
}
// BuiltIns/RectFigure.java
package BuiltIns;

import java.awt.*;
import java.awt.geom.Rectangle2D;
import Type.*;
import Main.*;

public class RectFigure extends Figure {

      IntType width, height, lineWidth;
      BoolType fill;

/* Constructor */
      public RectFigure(AbstractType... args) {
            if (args[0] != null) {
                  loc = (PairType) args[0];
            }

            if (args[1] != null) {
                  width = (IntType) args[1];
            }
            else throw new Error("Width must be specified.");
```

```
		if (args[2] != null) {
			height = (IntType) args[2];
		}
		else throw new Error("Height must be specified.");

		if (args[3] != null) {
			color = (ColorType) args[3];
		}

		if (args[4] != null) {
			fill = (BoolType) args[4];
		}
		else fill = new BoolType(true);

		if (args[5] != null) {
			lineWidth = (IntType) args[5];
		}
		else lineWidth = new IntType(1);
	}

/* Draw a rectangle by getting the values of all the parameters */
	public void draw(Graphics g) {
		Graphics2D g2 = (Graphics2D) g;
		g2.setStroke(new BasicStroke(lineWidth.value));
		g2.setColor(color.getColor());
		Rectangle2D.Float r = new Rectangle2D.Float(loc.x.value, loc.y.value,
width.value, height.value);
		if (_visible) {
			g2.draw(r);
			if (fill.value) {
				g2.fill(r);
			}
		}
	}
}
// BuiltIns/TextFigure.java
package BuiltIns;

import java.awt.*;
import java.awt.Font;
import Type.*;
import Main.*;

public class TextFigure extends Figure {

	StringType text, font;
```

```
	IntType size;

/* Constructor */
	public TextFigure(AbstractType... args) {
		if (args[0] != null) {
			loc = (PairType) args[0];
		}

		if (args[1] != null) {
			text = (StringType) args[1];
		}
		else throw new Error("Text must be specified.");

		if (args[2] != null) {
			color = (ColorType) args[2];
		}

		if (args[3] != null) {
			size = (IntType) args[3];
		}
		else size = new IntType(12);

		if (args[4] != null) {
			font = (StringType) args[4];
		}
		else font = new StringType("Arial");
	}

/* Go from the font name and size to a physical font available on the machine */
	private Font getFont() {
		GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();
		String fontFamilies[] = ge.getAvailableFontFamilyNames();
		if (java.util.Arrays.asList(fontFamilies).contains(font.value)) {
			return new Font(font.value, Font.PLAIN, size.value);
		}
		else throw new Error("The given font is not available on this
machine.");
	}

	public void draw(Graphics g) {
		Graphics2D g2 = (Graphics2D) g;
		g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
		g2.setColor(color.getColor());
		g2.setFont(this.getFont());
		g2.drawString(text.value, loc.x.value, loc.y.value);;
	}
```

```
}
// GeneratedCode/Arc.java
package GeneratedCode;

import Stdlib.*;
import Type.*;
import Main.*;
import BuiltIns.*;

public class Arc extends Figure {
PairType loc = new PairType(new IntType(0), new IntType(0));
IntType radius;
FloatType start = new FloatType(0.);
FloatType end = Arop.fogl_times(new IntType(2), Stdlib.Lib.PI);
ColorType col = new ColorType(0x00, 0x00, 0x00, 1.);
BoolType fill = new BoolType(true);
IntType lineWidth = new IntType(1);
BoolType clockwise = new BoolType(true);

ArcFigure arc;
void first_init() {
    components.add(arc);
}
void create() {
    arc = new ArcFigure(loc, radius, start, end, col, fill, lineWidth, clockwise);
}
void b_init() {
}
public Arc(AbstractType... arg) {
b_init();
if (arg[0] != null) { loc = (PairType) arg[0]; }
if (arg[1] != null) { radius = (IntType) arg[1]; }
if (arg[2] != null) { start = (FloatType) arg[2]; }
if (arg[3] != null) { end = (FloatType) arg[3]; }
if (arg[4] != null) { col = (ColorType) arg[4]; }
if (arg[5] != null) { fill = (BoolType) arg[5]; }
if (arg[6] != null) { lineWidth = (IntType) arg[6]; }
if (arg[7] != null) { clockwise = (BoolType) arg[7]; }
create();
first_init();
}
}

// GeneratedCode/Poly.java
package GeneratedCode;

import Stdlib.*;
import Type.*;
```

```
import Main.*;
import BuiltIns.*;

public class Poly extends Figure {
PairType loc = new PairType(new IntType(0), new IntType(0));
ListType<PairType> points;
ColorType col = new ColorType(0x00, 0x00, 0x00, 1.);
BoolType fill = new BoolType(true);
IntType lineWidth = new IntType(1);

PolyFigure poly;
void first_init() {
    components.add(poly);
}
void create() {
    poly = new PolyFigure(loc, points, col, fill, lineWidth);
}
void b_init() {
}
public Poly(AbstractType... arg) {
b_init();
if (arg[0] != null) { loc = (PairType) arg[0]; }
if (arg[1] != null) { points = (ListType<PairType>) arg[1]; }
if (arg[2] != null) { col = (ColorType) arg[2]; }
if (arg[3] != null) { fill = (BoolType) arg[3]; }
if (arg[4] != null) { lineWidth = (IntType) arg[4]; }
create();
first_init();
}
}

// GeneratedCode/Rect.java
package GeneratedCode;

import Stdlib.*;
import Type.*;
import Main.*;
import BuiltIns.*;

public class Rect extends Figure {
PairType loc = new PairType(new IntType(0), new IntType(0));
IntType width;
IntType height;
ColorType col = new ColorType(0x00, 0x00, 0x00, 1.);
BoolType fill = new BoolType(true);
IntType lineWidth = new IntType(1);

RectFigure rect;
```

```
void first_init() {
    components.add(rect);
}
void create() {
    rect = new RectFigure(loc, width, height, col, fill, lineWidth);
}
void b_init() {
}
public Rect(AbstractType... arg) {
b_init();
if (arg[0] != null) { loc = (PairType) arg[0]; }
if (arg[1] != null) { width = (IntType) arg[1]; }
if (arg[2] != null) { height = (IntType) arg[2]; }
if (arg[3] != null) { col = (ColorType) arg[3]; }
if (arg[4] != null) { fill = (BoolType) arg[4]; }
if (arg[5] != null) { lineWidth = (IntType) arg[5]; }
create();
first_init();
}
}

// GeneratedCode/Text.java
package GeneratedCode;

import Stdlib.*;
import Type.*;
import Main.*;
import BuiltIns.*;

public class Text extends Figure {
PairType loc = new PairType(new IntType(0), new IntType(0));
StringType text;
ColorType col = new ColorType(0x00, 0x00, 0x00, 1.);
IntType size = new IntType(12);
StringType font = new StringType("Arial");

TextFigure texting;
void first_init() {
    components.add(texting);
}
void create() {
    texting = new TextFigure(loc, text, col, size, font);
}
void b_init() {
}
public Text(AbstractType... arg) {
b_init();
if (arg[0] != null) { loc = (PairType) arg[0]; }
```

```
if (arg[1] != null) { text = (StringType) arg[1]; }
if (arg[2] != null) { col = (ColorType) arg[2]; }
if (arg[3] != null) { size = (IntType) arg[3]; }
if (arg[4] != null) { font = (StringType) arg[4]; }
create();
first_init();
}
}

// Main/Animation.java
package Main;

import javax.swing.*;
import java.awt.*;
import java.util.List;
import javax.swing.Timer;
import java.awt.event.*;
import GeneratedCode.*;
import Type.*;
import BuiltIns.*;

class Animation extends JPanel implements ActionListener{

	static int timeElapsed = 0;
	static int WIDTH = 640;
	static int HEIGHT = 480;

/* Make a blank panel with the given size */
	public Animation() {
		super();
		setSize(WIDTH, HEIGHT);
		setVisible(true);
		setDoubleBuffered(true);
	}

/* Draw all the figures; if they're not built-in, do it recursicvely */
	public void draw(Graphics g){
		for (Figure f : Figures.figures) {
			if ((f instanceof TextFigure) || (f instanceof RectFigure) || (f
instanceof PolyFigure) || (f instanceof ArcFigure)) {
				f.draw(g);
			}
			else {
			    for (Figure s : f.components) {
				s.draw(g);
			    }
			}
		}
```

```
	}

/* Overriding the JPanel paint method to also draw figures */
	public void paint(Graphics g) {
		super.paint(g);
		draw(g);
	}

/* Timer updates every 1 ms and then calls the panel itself as a listener */
	public void run() {
		Timer animator = new Timer(1, this);
		animator.start();
	}

/* Every ms update the time elapsed and update animating figures */
	public void actionPerformed(ActionEvent e) {
		++timeElapsed;
		for (Figure f : Figures.figures) {
			if (f._animating) {
				f._update.call(new IntType(timeElapsed));
			}
		}
		repaint();
	}

	public static void main(String[] args) {
		GeneratedMain.generatedMain();

		JFrame holder = new JFrame("Animation");
		holder.setVisible(true);
		holder.setSize(WIDTH, HEIGHT);
		holder.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		Animation proj = new Animation();
		holder.add(proj);

		proj.run();
	}
}
// Main/Captures.java
package Main;

import GeneratedCode.*;
import Type.*;
import java.util.List;
import java.util.ArrayList;

public abstract class Captures {
    public static AbstractType[] record;
```

```
}
// Main/Figure.java
package Main;

import Type.*;

import java.util.List;
import java.util.ArrayList;
import java.awt.*;

public abstract class Figure extends AbstractType
{
    //all figures have visible and animating properties
    public boolean _visible = true;
    public boolean _animating;

    public PairType loc = new PairType(new IntType(0), new IntType(0));
    public ColorType color = new ColorType(0,0,0,1);

    public List<Figure> components = new ArrayList<Figure>();

    public Figure() {
       Figures.figures.add(this);
    }

    public void draw(Graphics g) {

    }

    //(* _update.call() gives the update method *)
    public UpdateFunctionType _update;
}
// Main/Figures.java
package Main;

import java.util.List;
import java.util.ArrayList;

public abstract class Figures {
    public static List<Figure> figures = new ArrayList<Figure>();
}
// Stdlib/Abs.java
package Stdlib;

import Type.*;

public class Abs extends FunctionType {
    public FloatType call(FloatType x) {
```

```
       return new FloatType(Math.abs(x.value));
    }
}
// Stdlib/Acos.java
package Stdlib;

import Type.*;

public class Acos extends FunctionType {
    public FloatType call(FloatType a) {
       return new FloatType(Math.acos(a.value));
    }
}
// Stdlib/Asin.java
package Stdlib;

import Type.*;

public class Asin extends FunctionType {
    public FloatType call(FloatType a) {
       return new FloatType(Math.asin(a.value));
    }
}
// Stdlib/Atan.java
package Stdlib;

import Type.*;

public class Atan extends FunctionType {
    public FloatType call(FloatType a) {
       return new FloatType(Math.atan(a.value));
    }
}
// Stdlib/Ceil.java
package Stdlib;

import Type.*;

public class Ceil extends FunctionType {
    public FloatType call(FloatType a) {
       return new FloatType(Math.ceil(a.value));
    }
}
// Stdlib/Cos.java
package Stdlib;

import Type.*;
```

```
public class Cos extends FunctionType {
    public FloatType call(FloatType a) {
       return new FloatType(Math.cos(a.value));
    }
}
// Stdlib/Floor.java
package Stdlib;

import Type.*;

public class Floor extends FunctionType {
    public FloatType call(FloatType a) {
       return new FloatType(Math.floor(a.value));
    }
}
// Stdlib/Ftoi.java
package Stdlib;

import Type.*;

public class Ftoi extends FunctionType {
    public IntType call(FloatType a) {
       return new IntType((int)a.value);
    }
}
// Stdlib/Ftos.java
package Stdlib;

import Type.*;

public class Ftos extends FunctionType {
    public StringType call(FloatType a) {
       return new StringType("" + a.value);
    }
}
// Stdlib/Itof.java
package Stdlib;

import Type.*;

public class Itof extends FunctionType {
    public FloatType call(IntType a) {
       return new FloatType(a.value);
    }
}
// Stdlib/Itos.java
package Stdlib;
```

```
import Type.*;

public class Itos extends FunctionType {
    public StringType call(IntType a) {
       return new StringType("" + a.value);
    }
}
// Stdlib/Lib.java
package Stdlib;

import Type.*;

public class Lib {
    public static Print print = new Print();
    public static FunctionType type;
    public static Itof itof = new Itof();
    public static Ftoi ftoi = new Ftoi();
    public static Itos itos = new Itos();
    public static Ftos ftos = new Ftos();
    public static Stoi stoi = new Stoi();
    public static FunctionType stof = new Stof();
    public static FunctionType rgb;
    public static FunctionType rgba;
    public static X x = new X();
    public static Y y = new Y();
    public static FunctionType dist;
    public static FunctionType mag;
    public static FunctionType ang;
    public static FunctionType dot;
    public static FunctionType cross;
    public static FunctionType bet;
    public static ToDeg toDeg = new ToDeg();
    public static ToRad toRad = new ToRad();
    public static Sin sin = new Sin();
    public static Cos cos = new Cos();
    public static Tan tan = new Tan();
    public static Asin asin = new Asin();
    public static Acos acos = new Acos();
    public static Atan atan = new Atan();
    public static Log log = new Log();
    public static Ln ln = new Ln();
    public static Ceil ceil = new Ceil();
    public static Floor floor = new Floor();
    public static Sqrt sqrt = new Sqrt();
    public static Round round = new Round();
    public static Min min = new Min();
    public static Max max = new Max();
    public static Abs abs = new Abs();
```

```
    public static Random random = new Random();
    public static FloatType PI = new FloatType(3.14159265359);
    public static FloatType E = new FloatType(2.71828182846);
    public static FloatType RT_2 = new FloatType(1.41421356237);
    public static FloatType RT_3 = new FloatType(1.73205080757);
}
// Stdlib/Ln.java
package Stdlib;

import Type.*;

public class Ln extends FunctionType {
    public FloatType call(FloatType a) {
       return new FloatType(Math.log(a.value));
    }
}
// Stdlib/Log.java
package Stdlib;

import Type.*;

public class Log extends FunctionType {
    public FloatType call(FloatType a, FloatType b) {
       return new FloatType(Math.log(a.value)/Math.log( b.value));
    }
}
// Stdlib/Max.java
package Stdlib;

import Type.*;

public class Max extends FunctionType {
    public FloatType call(FloatType a, FloatType b) {
       if (a.value > b.value) {
           return a;
       }else {
           return b;
       }
    }
}
// Stdlib/Min.java
package Stdlib;

import Type.*;

public class Min extends FunctionType {
    public FloatType call(FloatType a, FloatType b) {
       if (a.value > b.value)
```

```
            return b;
        else
            return a;
    }
}
// Stdlib/Print.java
package Stdlib;

import Type.*;

public class Print extends FunctionType {
    public void call(StringType s) {
        System.out.println(s.value);
    }
}
// Stdlib/Random.java
package Stdlib;

import Type.*;

public class Random extends FunctionType {
    public FloatType call() {
        return new FloatType(Math.random());
    }
}
// Stdlib/Round.java
package Stdlib;

import Type.*;

public class Round extends FunctionType {
    public FloatType call(FloatType a) {
        return new FloatType(Math.round(a.value));
    }
}
// Stdlib/Sin.java
package Stdlib;

import Type.*;

public class Sin extends FunctionType {
    public FloatType call(FloatType a) {
        return new FloatType(Math.sin(a.value));
    }
}
// Stdlib/Sqrt.java
package Stdlib;
```

```
import Type.*;

public class Sqrt extends FunctionType {
    public FloatType call(FloatType a) {
       return new FloatType(Math.sqrt(a.value));
    }
}
// Stdlib/Stof.java
package Stdlib;

import Type.*;

public class Stof extends FunctionType {
    public FloatType call(StringType a) {
       return new FloatType(Float.parseFloat(a.value));
    }
}
// Stdlib/Stoi.java
package Stdlib;

import Type.*;

public class Stoi extends FunctionType {
    public IntType call(StringType a) {
       return new IntType(Integer.parseInt(a.value));
    }
}
// Stdlib/Tan.java
package Stdlib;

import Type.*;

public class Tan extends FunctionType {
    public FloatType call(FloatType a) {
       return new FloatType(Math.tan(a.value));
    }
}
// Stdlib/ToDeg.java
package Stdlib;

import Type.*;

public class ToDeg extends FunctionType {
    public FloatType call(FloatType a) {
       return new FloatType(a.value*360/(2*Math.PI));
    }
}
// Stdlib/ToRad.java
```

```
package Stdlib;

import Type.*;

public class ToRad extends FunctionType {
    public FloatType call(FloatType a) {
       return new FloatType(a.value * 2* Math.PI / 360);
    }
}
// Stdlib/X.java
package Stdlib;

import Type.*;

public class X extends FunctionType {
    public FloatType call(PairType a) {
       return new FloatType(((FloatType) a.x).value);
    }
}
// Stdlib/Y.java
package Stdlib;

import Type.*;

public class Y extends FunctionType {
    public FloatType call(PairType a) {
       return new FloatType(((FloatType) a.y).value);
    }
}
// Type/AbstractType.java
package Type;

public abstract class AbstractType {

}
// Type/Arop.java
package Type;

public abstract class Arop {
    public static IntType fogl_plus(IntType a,IntType b) {
       return new IntType(a.value + b.value);
    }
    public static FloatType fogl_plus(FloatType a, FloatType b) {
       return new FloatType(a.value + b.value);
    }
    public static FloatType fogl_plus(FloatType a, IntType b) {
       return new FloatType(a.value + b.value);
    }
```

```
public static FloatType fogl_plus(IntType a, FloatType b) {
   return new FloatType(a.value + b.value);
}
public static StringType fogl_plus(StringType a, StringType b) {
   return new StringType(a.value + b.value);
}
   public static PairType fogl_plus(PairType a, PairType b) {
   return new PairType(fogl_plus(a.x, b.x), fogl_plus(a.y, b.y));
   }
   public static PairType fogl_plus(PairType a, IntType b) {
   return new PairType(fogl_plus(a.x, b), fogl_plus(a.y, b));
   }
   public static PairType fogl_plus(PairType a, FloatType b) {
   return new PairType(fogl_plus(a.x, b), fogl_plus(a.y, b));
   }
   public static PairType fogl_plus(IntType a, PairType b) {
   return new PairType(fogl_plus(a, b.x), fogl_plus(a, b.y));
   }
   public static PairType fogl_plus(FloatType a, PairType b) {
   return new PairType(fogl_plus(a, b.x), fogl_plus(a, b.y));
   }

public static IntType fogl_minus(IntType a,IntType b) {
   return new IntType(a.value - b.value);
}
   public static FloatType fogl_minus(FloatType a,IntType b) {
   return new FloatType(a.value - b.value);
}
   public static FloatType fogl_minus(IntType a,FloatType b) {
   return new FloatType(a.value - b.value);
}
   public static FloatType fogl_minus(FloatType a,FloatType b) {
   return new FloatType(a.value - b.value);
}
   public static PairType fogl_minus(PairType a, PairType b) {
   return new PairType(fogl_minus(a.x, b.x), fogl_minus(a.y, b.y));
   }
   public static PairType fogl_minus(PairType a, IntType b) {
   return new PairType(fogl_minus(a.x, b), fogl_minus(a.y, b));
   }
   public static PairType fogl_minus(PairType a, FloatType b) {
   return new PairType(fogl_minus(a.x, b), fogl_minus(a.y, b));
   }
   public static PairType fogl_minus(IntType a, PairType b) {
   return new PairType(fogl_minus(a, b.x), fogl_minus(a, b.y));
   }
   public static PairType fogl_minus(FloatType a, PairType b) {
   return new PairType(fogl_minus(a, b.x), fogl_minus(a, b.y));
```

```
   }

public static IntType fogl_times(IntType a,IntType b) {
   return new IntType(a.value * b.value);
}
   public static FloatType fogl_times(FloatType a,IntType b) {
   return new FloatType(a.value * b.value);
}
   public static FloatType fogl_times(IntType a,FloatType b) {
   return new FloatType(a.value * b.value);
}
   public static FloatType fogl_times(FloatType a,FloatType b) {
   return new FloatType(a.value * b.value);
}
   public static PairType fogl_times(PairType a, PairType b) {
   return new PairType(fogl_times(a.x, b.x), fogl_times(a.y, b.y));
   }
   public static PairType fogl_times(PairType a, IntType b) {
   return new PairType(fogl_times(a.x, b), fogl_times(a.y, b));
   }
   public static PairType fogl_times(PairType a, FloatType b) {
   return new PairType(fogl_times(a.x, b), fogl_times(a.y, b));
   }
   public static PairType fogl_times(IntType a, PairType b) {
   return new PairType(fogl_times(a, b.x), fogl_times(a, b.y));
   }
   public static PairType fogl_times(FloatType a, PairType b) {
   return new PairType(fogl_times(a, b.x), fogl_times(a, b.y));
   }

public static FloatType fogl_div(IntType a,IntType b) {
   return new FloatType((float) a.value / (float) b.value);
}
   public static FloatType fogl_div(FloatType a,IntType b) {
   return new FloatType(a.value / b.value);
}
   public static FloatType fogl_div(IntType a,FloatType b) {
   return new FloatType(a.value / b.value);
}
   public static FloatType fogl_div(FloatType a,FloatType b) {
   return new FloatType(a.value / b.value);
}
   public static PairType fogl_div(PairType a, PairType b) {
   return new PairType(fogl_div(a.x, b.x), fogl_div(a.y, b.y));
   }
   public static PairType fogl_div(PairType a, IntType b) {
   return new PairType(fogl_div(a.x, b), fogl_div(a.y, b));
   }
```

```
   public static PairType fogl_div(PairType a, FloatType b) {
   return new PairType(fogl_div(a.x, b), fogl_div(a.y, b));
   }
   public static PairType fogl_div(IntType a, PairType b) {
   return new PairType(fogl_div(a, b.x), fogl_div(a, b.y));
   }
   public static PairType fogl_div(FloatType a, PairType b) {
   return new PairType(fogl_div(a, b.x), fogl_div(a, b.y));
   }

   public static IntType fogl_divdiv(IntType a,IntType b) {
   return new IntType((int)(a.value / b.value));
}
   public static IntType fogl_divdiv(FloatType a,IntType b) {
   return new IntType((int)(a.value / b.value));
}
   public static IntType fogl_divdiv(IntType a,FloatType b) {
   return new IntType((int) (a.value / b.value));
}
   public static IntType fogl_divdiv(FloatType a,FloatType b) {
   return new IntType((int)(a.value / b.value));
}
   public static PairType fogl_divdiv(PairType a, PairType b) {
   return new PairType(fogl_divdiv(a.x, b.x), fogl_divdiv(a.y, b.y));
   }
   public static PairType fogl_divdiv(PairType a, IntType b) {
   return new PairType(fogl_divdiv(a.x, b), fogl_divdiv(a.y, b));
   }
   public static PairType fogl_divdiv(PairType a, FloatType b) {
   return new PairType(fogl_divdiv(a.x, b), fogl_divdiv(a.y, b));
   }
   public static PairType fogl_divdiv(IntType a, PairType b) {
   return new PairType(fogl_divdiv(a, b.x), fogl_divdiv(a, b.y));
   }
   public static PairType fogl_divdiv(FloatType a, PairType b) {
   return new PairType(fogl_divdiv(a, b.x), fogl_divdiv(a, b.y));
   }

public static IntType fogl_exp(IntType a,IntType b) {
   return new IntType((int) Math.pow(a.value, b.value));
}
   public static FloatType fogl_exp(FloatType a,IntType b) {
   return new FloatType(Math.pow(a.value, b.value));
}
   public static FloatType fogl_exp(IntType a,FloatType b) {
   return new FloatType(Math.pow(a.value, b.value));
}
   public static FloatType fogl_exp(FloatType a,FloatType b) {
```

```
  return new FloatType(Math.pow(a.value, b.value));
}
  public static PairType fogl_exp(PairType a, PairType b) {
  return new PairType(fogl_exp(a.x, b.x), fogl_exp(a.y, b.y));
  }
  public static PairType fogl_exp(PairType a, IntType b) {
  return new PairType(fogl_exp(a.x, b), fogl_exp(a.y, b));
  }
  public static PairType fogl_exp(PairType a, FloatType b) {
  return new PairType(fogl_exp(a.x, b), fogl_exp(a.y, b));
  }
  public static PairType fogl_exp(IntType a, PairType b) {
  return new PairType(fogl_exp(a, b.x), fogl_exp(a, b.y));
  }
  public static PairType fogl_exp(FloatType a, PairType b) {
  return new PairType(fogl_exp(a, b.x), fogl_exp(a, b.y));
  }

  public static IntType fogl_percent(IntType a,IntType b) {
  return new IntType(a.value % b.value);
}
  public static FloatType fogl_percent(FloatType a,IntType b) {
  return new FloatType(a.value % b.value);
}
  public static FloatType fogl_percent(IntType a,FloatType b) {
  return new FloatType(a.value % b.value);
}
  public static FloatType fogl_percent(FloatType a,FloatType b) {
  return new FloatType(a.value % b.value);
}
  public static PairType fogl_percent(PairType a, PairType b) {
  return new PairType(fogl_percent(a.x, b.x), fogl_percent(a.y, b.y));
  }
  public static PairType fogl_percent(PairType a, IntType b) {
  return new PairType(fogl_percent(a.x, b), fogl_percent(a.y, b));
  }
  public static PairType fogl_percent(PairType a, FloatType b) {
  return new PairType(fogl_percent(a.x, b), fogl_percent(a.y, b));
  }
  public static PairType fogl_percent(IntType a, PairType b) {
  return new PairType(fogl_percent(a, b.x), fogl_percent(a, b.y));
  }
  public static PairType fogl_percent(FloatType a, PairType b) {
  return new PairType(fogl_percent(a, b.x), fogl_percent(a, b.y));
  }

  public static BoolType fogl_eqeq(IntType a, IntType b) {
  return new BoolType(a.value == b.value);
```

```
}
   public static BoolType fogl_eqeq(FloatType a, IntType b) {
   return new BoolType(a.value == b.value);
}
   public static BoolType fogl_eqeq(IntType a, FloatType b) {
   return new BoolType(a.value == b.value);
}
   public static BoolType fogl_eqeq(FloatType a, FloatType b) {
   return new BoolType(a.value == b.value);
}

   public static BoolType fogl_neq(IntType a, IntType b) {
   return new BoolType(a.value != b.value);
}
   public static BoolType fogl_neq(FloatType a, IntType b) {
   return new BoolType(a.value != b.value);
}
   public static BoolType fogl_neq(IntType a, FloatType b) {
   return new BoolType(a.value != b.value);
}
   public static BoolType fogl_neq(FloatType a, FloatType b) {
   return new BoolType(a.value != b.value);
}

   public static BoolType fogl_gt(IntType a, IntType b) {
   return new BoolType(a.value > b.value);
}
   public static BoolType fogl_gt(FloatType a, IntType b) {
   return new BoolType(a.value > b.value);
}
   public static BoolType fogl_gt(IntType a, FloatType b) {
   return new BoolType(a.value > b.value);
}
   public static BoolType fogl_gt(FloatType a, FloatType b) {
   return new BoolType(a.value > b.value);
}

   public static BoolType fogl_lt(IntType a, IntType b) {
   return new BoolType(a.value < b.value);
}
   public static BoolType fogl_lt(FloatType a, IntType b) {
   return new BoolType(a.value < b.value);
}
   public static BoolType fogl_lt(IntType a, FloatType b) {
   return new BoolType(a.value < b.value);
}
   public static BoolType fogl_lt(FloatType a, FloatType b) {
   return new BoolType(a.value < b.value);
```

```
}

   public static BoolType fogl_gte(IntType a, IntType b) {
   return new BoolType(a.value >= b.value);
}
   public static BoolType fogl_gte(FloatType a, IntType b) {
   return new BoolType(a.value >= b.value);
}
   public static BoolType fogl_gte(IntType a, FloatType b) {
   return new BoolType(a.value >= b.value);
}
   public static BoolType fogl_gte(FloatType a, FloatType b) {
   return new BoolType(a.value >= b.value);
}

   public static BoolType fogl_lte(IntType a, IntType b) {
   return new BoolType(a.value <= b.value);
}
   public static BoolType fogl_lte(FloatType a, IntType b) {
   return new BoolType(a.value <= b.value);
}
   public static BoolType fogl_lte(IntType a, FloatType b) {
   return new BoolType(a.value <= b.value);
}
   public static BoolType fogl_lte(FloatType a, FloatType b) {
   return new BoolType(a.value <= b.value);
}

public static IntType fogl_neg(IntType a) {
   return new IntType(-a.value);
}

   public static FloatType fogl_neg(FloatType a) {
   return new FloatType(-a.value);
}

   public static BoolType fogl_not(BoolType a){
   return new BoolType(!a.value);
   }
   public static BoolType fogl_not(boolean a) {
   return new BoolType(!a);
   }

public static BoolType fogl_question(BoolType a) {
          return a;
   }
   public static BoolType fogl_question(ColorType a) {
          return new BoolType(true);
```

```
	}
	public static BoolType fogl_question(FunctionType a) {
		return new BoolType(true);
	}
	public static BoolType fogl_question(ListType a) {
		return new BoolType(true);
	}
	public static BoolType fogl_question(PairType a) {
		return new BoolType(true);
	}
	public static BoolType fogl_question(FloatType a) {
		if (a.value == 0) {
			return new BoolType(false);
		}
		else return new BoolType(true);
	}
	public static BoolType fogl_question(IntType a) {
		if (a.value == 0) {
			return new BoolType(false);
		}
		else return new BoolType(true);
	}
	public static BoolType fogl_question(StringType a) {
		if (a.value.equals("")) {
			return new BoolType(false);
		}
		else return new BoolType(true);
	}
}
// Type/BoolType.java
package Type;

public class BoolType extends AbstractType {
    public boolean value;

    public BoolType(boolean myValue) {
       value = myValue;
    }
    public BoolType() {
    value = false;
    }
}
// Type/ColorType.java
package Type;

import java.awt.Color;

public class ColorType extends AbstractType {
```

```
    public int[] value;
    public float alpha;

    public Color getColor() {
             return new Color(value[0], value[1], value[2], (int)
Math.floor(255*alpha));
       }

    public ColorType (int[] myValue, double myAlpha) {
       value = myValue;
       alpha = (float) myAlpha;
    }

    public ColorType(int a, int b, int c, double myAlpha) {
       value = new int[3];
       value[0] = a;
       value[1] = b;
       value[2] = c;
       alpha = (float)myAlpha;
    }

    public ColorType (IntType[] myValue, FloatType myAlpha) {
       value = new int[3];
       for (int i = 0; i < 3; ++i)      {
             value[i] = myValue[i].value;
       }
       alpha = myAlpha.value;
    }

    public ColorType(IntType a, IntType b, IntType c, FloatType myAlpha) {
       value = new int[3];
       value[0] = a.value;
       value[1] = b.value;
       value[2] = c.value;
       alpha = myAlpha.value;
    }

    public ColorType() {
    value = new int[3];
    value[0] = 0;
    value[1] = 0;
    value[2] = 0;
    alpha = (float) 1.0;
    }
}
// Type/FloatType.java
package Type;
```

```
public class FloatType extends AbstractType {

    public float value;

    public FloatType (float myValue) {
       value = myValue;
    }
    public FloatType (double myValue) {
       value = (float) myValue;
    }
    public FloatType(IntType other) {
       value = other.value;
    }
    public FloatType() {
    value = (float) 0.0;
    }
}
// Type/FunctionType.java
package Type;

public class FunctionType<T> extends AbstractType {
}
// Type/IntType.java
package Type;

public class IntType extends AbstractType {

    public int value;

    public IntType (int myValue) {
       value = myValue;
    }
    public IntType () {
    value = 0;
    }
}
// Type/ListType.java
package Type;

import java.util.ArrayList;
import java.util.List;

public class ListType<T> extends AbstractType {

    private List<T> list = new ArrayList<T>();

    private Class<T> type;
```

```
    private T createContents()
    {
        try {
            return type.newInstance();
        } catch (Exception e) {
            return null;
        }
    }

    public ListType () {

    }
    public ListType(T... args) {
       for(T t : args) {
           list.add(t);
       }
    }
       public int size() {
       return list.size();
       }
    public T get(int i) {
       return list.get(i);
    }
    public T get(IntType i) {
       return list.get(i.value);
    }
    public T set(IntType i, T elt) {
    if (i.value >= list.size()) {
        for (int j = list.size(); j <= i.value; j++) {
            list.add(createContents());
        }
    }
       return list.set(i.value, elt);
    }
}

// Type/PairType.java
package Type;

public class PairType extends AbstractType {
    public FloatType x;
    public FloatType y;

    public PairType (AbstractType myX, AbstractType myY) {
       if (myX instanceof IntType) {
           x = new FloatType(((IntType)myX).value);
       }else if (myX instanceof FloatType) {
           x = new FloatType(((FloatType)myX).value);
```

```
        } else {
            throw new Error("X value of ordered pair must be int or float.");
        }
        if (myY instanceof IntType)
            y = new FloatType(((IntType)myY).value);
        else if ( myY instanceof FloatType) {
            y = new FloatType(((FloatType)myY).value);
        } else {
            throw new Error("Y value of ordered pair must be int or float.");
        }
    }
    public PairType () {
    x = new FloatType(0.0);
    y = new FloatType(0.0);
    }

}

// Type/StringType.java
package Type;

public class StringType extends AbstractType {
    public String value;

    public StringType(String myValue) {
       value = myValue;
    }
    public StringType () {
    value = "";
    }
}

// Type/UpdateFunctionType.java
package Type;

public abstract class UpdateFunctionType extends FunctionType {
    public abstract void call(IntType ms);
}
```

8.10 - Print Tests

```
; arith.fogl
print(itos(3+4))
print(itos(1 + 2 * 3 - 4))
print(itos(3^4))
print(ftos(9^.5))
print(ftos(5/4))
print(itos(5//4))
```

```
; fib.fogl
var fib = func[int] (x:int) {
  if x < 2 { return 1 }
  else { return fib(x-1) + fib(x-2) }
}
print(itos(fib(0)))
print(itos(fib(1)))
print(itos(fib(2)))
print(itos(fib(3)))
print(itos(fib(4)))
print(itos(fib(5)))

; figure1.fogl
figure Foo {
	param bar = 5
}
var f1 = Foo(:),
f2 = Foo(bar:10)
print(itos(f1.bar))
print(itos(f2.bar))
f1.bar = 15
print(itos(f1.bar))

; figure2.fogl
figure Foo {
	param bar = 5,
	baz = func () {
		print(itos(bar))
	}
}
var f1 = Foo(:),
f2 = Foo(bar:10)
f1.baz()
f2.baz()

; figure3.fogl
figure Foo {
	param _visible = false,
	bar = int
}
var f1 = figure{}
f1 = Foo(bar:5)
if !(f1._visible) {
	var f2 = Foo(f1)
	print(itos(f2.bar))
}

; float.fogl
```

```
var f = float
f = 1.
f = 0.5e-15
f = .3e+3
f = .2
f = 1e5
f = 49
print(ftos(f))

; func1.fogl
var add = func[int] (a:int, b:int) { return a + b },
a = int
a = add(39, 3)
print(itos(a))

; func2.fogl
var printem = func (a:int, b:int, c:int, d:int) {
  print(itos(a))
  print(itos(b))
  print(itos(c))
  print(itos(d))
}
printem(49, 256, -10, 0)

; func3.fogl
var divisor = func[int(int)] (d:int) (func[int] (x:int) (x//d)),
half = divisor(2),
foo = half(8)
print(itos(foo))

; func4.fogl
var bar = 2,
changebar = func (x:int) { bar = x },
foo = func () {
  var bar = 6
  changebar(4)
}
foo()
print(itos(bar))

; func5.fogl
var foo = func[()] (n:int) {
  var bar = 5
  var baz = func () {
    bar = bar + n
    print(itos(bar))
  }
  print(itos(bar))
```

```
  return baz
}
var qux = foo(3)
qux()
qux()
foo(2)

; gcd.fogl
var gcd = func[int] (a:int, b:int) {
  while a != b {
    if a > b { a = a - b }
    else { b = b - a }
  }
  return a
}

print(itos(gcd(2,14)))
print(itos(gcd(3,15)))
print(itos(gcd(99,121)))


; helloworld.fogl
print("Hello World!")

; if.fogl
if true { print("1") }

if false { print("0") }
else { print("2") }

if true { print("3") }
elseif true { print("0") }

if false { print("0") }
elseif false { print("0") }
else { print("4") }

if true
{
print("5")
}
else
{
print("0")
}

; list.fogl
var foo = int[][][]
```

```
foo = [[], [[]]]
var bar = [[], [], [], [0, 1, 2], []]
bar[3][1] = 5
var baz = int[]
baz[2] = 2
print(itos(bar[3][1]))
print(itos(baz[2]))

; pair.fogl
var printPair = func (p:pair) {
  print("<<" + ftos(x(p)) + ", " + ftos(y(p)) + ">>")
}
printPair(<<3, 4>>)
printPair(<<3, 4>> + <<-2, 1>>)
printPair(<<3, 4>> * 2)
printPair(4 - <<3, 4>>)

; while.fogl
var i = int
i = 5
while i > 0 {
  print(itos(i))
  i = i - 1
}
print("42")
```

8.11 - Graphical Tests

```
; Arc1.fogl
; Draws arcs in two different directions, both of radius 50

Arc(loc:<<300, 300>> radius:50 end:PI)
Arc(loc:<<200, 400>> radius:50 end:PI clockwise:false)


; hellotimer.fogl
; Draws "Hello World!" in blue at <<10,10>> and changes to red/blue every second
figure Hello {
	param loc = <<10,10>>, col = color
	var cycle = int
	comp text = Text(text:"Hello World!" loc:loc col:col)

	var _update = func (ms:int) {
		cycle = (ms//1000)%2

		if cycle == 0 {
		col = #00f
		}
		else {
```

```
            col = #f00
            }
      }
}
Hello(loc:<<10,10>> col:#00f)


; helloworld.fogl
; Draws "Hello World!" in blue at <<10,10>> and in red (opacity .5) at <<100,100>>
figure Hello {
      param loc = <<10,10>>, col = color
      comp text = Text(text:"Hello World!" loc:loc col:col)
}
Hello(col:#00f)
Hello(col:#f00@.5 loc:<<100,100>>)


; Rect1.fogl
; Draws rectangles of different line widths

figure RectEdge {
      param loc = <<10, 10>>, stroke = 1
      comp r = Rect(loc:loc width:50 height:80 fill:false lineWidth:stroke)
}

RectEdge(:)
RectEdge(stroke:4 loc:<<100, 10>>)
RectEdge(stroke:16 loc:<<200, 10>>)


; Rect2.fogl
; Draws a rectangle at 0,0, width 50, height 80, colored in blue at .1 opacity
Rect(loc:<<0,0>> width:50 height:80 fill:true col:#00f@.1)

; Sierpinski.fogl
; Draws a Sierpinski fractal
figure Equilateral {
      param loc = <<0,0>>, sideLength = int, fill = true
      comp triangle = Poly(loc:loc points:[<<0,0>>, <<-sideLength/2,sideLength/
2*RT_3>>,
<<sideLength/2,sideLength/2*RT_3>>] fill:fill)
}

figure Sierpinski {
      param loc = <<0,0>>, sideLength = int
      comp triangle = Equilateral(loc:loc sideLength:sideLength fill:false),
      s1 = Sierpinski(loc:loc sideLength:sideLength//2),
```

```
	s2 = Sierpinski(loc:loc+<<-sideLength/4,sideLength/4*RT_3>>
sideLength:sideLength//2),
	s3 = Sierpinski(loc:loc+<<sideLength/4,sideLength/4*RT_3>>
sideLength:sideLength//2)
}
Sierpinski(loc:<<100,100>> sideLength:5)

; StopSign.fogl
; Draws a red octagon at <<200, 200>> and writes STOP on it in white Times New Roman

figure StopSign {
	param loc = <<200, 200>>, col = color
	comp octagon = Poly(loc:loc points:[<<100*cos(PI/8), 100*sin(PI/8)>>,
<<100*cos(3*PI/8), 100*sin(3*PI/8)>>, <<100*cos(5*PI/8), 100*sin(5*PI/8)>>,
<<100*cos(7*PI/8), 100*sin(7*PI/8)>>, <<100*cos(9*PI/8), 100*sin(9*PI/8)>>,
<<100*cos(11*PI/8), 100*sin(11*PI/8)>>, <<100*cos(13*PI/8), 100*sin(13*PI/8)>>,
<<100*cos(15*PI/8), 100*sin(15*PI/8)>>] col:col)
	comp text = Text(loc:loc-<<20, 0>> text:"STOP" col:#fff size:18 font:"Times
New Roman")
}

StopSign(col:#f00)


; Timer.fogl
; A timer that updates every second

figure Timer {
	var time = 0
	comp display = Text(text:(itos(time)) loc:<<200,200>> size:20)

	var _update = func (ms:int) {
		time = ms//1000
	}
}

var t = Timer(:)
```