Simple Image Generation Language (SIGL)

Phong Pham Abelardo Gutierrez Alketa Aliaj

Programming Languages and Translators
Spring 2007

Contents

Chapter 1

Introduction

In this project, we propose developing a new language for 2D image generation, Simple Image Generation Language (SIGL). SIGL can be seen as an extension of Virtual Reality Modeling Language (VRML), widely used for specifying 3D models. It deploys the familiar image specification methodology and syntax of VRML while providing more control and flexibility with conditional branching and loop similar to that of any general purpose programming language. Our designing goal for SIGL is to provide users with a simple, natural and yet flexible way to draw 2D images.

1.1 Motivation

Why a drawing language? With the availability of a lot of drawing software available, drawing images are just the matter of simple mouse clicks. However, not all images can be easily drawn by hand. For example, it would be very tedious to draw a grid, or a set of many co-center circles. A drawing programming language would make repetitive drawing task easy. It also allows drawing of complicated mathematical curves.

Why another drawing language? While VRML is a widely used standard, it is not suitable for programmers. In most cases, VRML files are generated by drawing softwares. In our language, we provide a drawing system using same OpenGL drawing methodology as VRML, empowered with additional flow control found in common programming languages.

Why a totally new language? Even though it is possible to build a similar library in general purpose languages such as Java/C/C++, the drawing syntax would be complicated by the parent language. The users also need to know the parent languages at the first place. In contrast, SIGL provides a natural and easy-to-understand syntax which a beginner can catch up in a short period of time.

1.2 Goal

The main goal of our language is to provide a simple, yet powerful, programming language that allows users to create simple 2D images in shortest time possible. While the choice of OpenGL drawing style might introduce a steep learning curve at the beginning for people who are not familiar with OpenGL drawing mechanism, we believe that this mechanism will allow much more flexibility once the users get used to the language.

1.2.1 Portability

Our interpreter is implemented in Java, and therefore can be run on any platform as long as Java is supported. For maximum compatibility, we have decided to deploy Java 1.4.2 instead of the latest version Java 6.0.

1.2.2 Ease-of-use

While the language has all powerful control flow found in general purpose language, users do not need to know at all about programming. Basic syntax for drawing is simple and intuitive. We have decided on the top-down execution flow instead of starting from a `main` method as commonly seen in C or Java, which might be confusing to non-programmer users.

1.2.3 Flexibility

With simplicity in mind, however, we do not limit our language to simple operations. We support all basic control flow available in any general purpose languages such as C or Java. Furthermore, we also integrate many different features that might not found in other programming languages: switching between dynamic and static scoping, switching between normal and applicative evaluation order. We believe that other than drawing purpose, this language might find its place in programming language courses where a demonstration for different language concepts can be done with ease.

1.3 Main language features

In this section, we highlight some of the key features of our language. While the purpose of the language is for drawing, since the course is about programming language rather than computer graphics, we concentrate much of our focus on enriching the language in programming language aspect rather than drawing aspect. Therefore, the drawing functionality only provides basic primitives. However, with the flexibility in the language aspect, there is nothing that will limit the users from creating fancy drawing using our language.

1.3.1 Drawing features

- OpenGL drawing mechanism
- Drawing primitives: lines, polygons, ellipses.
- RGB color space.
- Transformation: translation, rotation, scale.

1.3.2 Programming language features

C-like syntax

We choose to use C-like syntax as it is one of the most popular syntax which has been deployed in many other languages such as Java, C#, and also commonly used for pseudo-code in most programming language textbooks. We support nearly all C control flow constructions, including `if, while, for`, logical, arithmetic, relational operations, to name a few.

The only significant syntax that we don't support is `switch` which we find not really necessary as it can always be replaced with stacking `if`.

Data types

There are 4 main types in our language: int, real (equivalent to `double` in C), boolean, associative array, and function. Functions in our languages are first order entities and can be passed around just like variables.

Type system

The type system used in our language is dynamic type system. All type-checking is done at runtime

Scoping

By default, our language is static scoping. However, our interpreter provides an option to switch between static and dynamic scoping.

Evaluation order

By default, the evaluation order in our language is applicative order. Again, this can be switched to normal order evaluation by providing an extra option to our interpreter.

Built-in functions

Most functionalities in our language are introduced in form of built-in functions. Our built-in functions are automatically loaded at runtime.

Meaningful error message

Whenever something goes wrong, we try our best to provide a meaningful error message to users. Each error message is accompanied by the line and column of the code snippet causing the error, as well as the reason why the error is raised.

Chapter 2

Language Tutorial

SIGL, similar to VRML, deploys the OpenGL drawing mechanism. The result of a drawing command depends not only on the command itself, but also the transformation at the time of drawing. For example

```
ellipse (10, 10)
```

will draw a circle of diameter 10, with center at (0,0). If we would like to draw that same circle but with center at (50,100), we use the translation transformation

```
translate (50, 100);
ellipse (10, 10)
```

Other transformations available are rotation and scale, which rotates the object in 2D and scale it, respectively.

2.0.3 Localizing transformation effect

In the above code, every drawing command after `translate (50, 100);` will be moved in x-direction by 50 and y-direction by 100. However, sometimes we would like the effect to apply to a few commands only. We can do that with the following construction

```
:translate (50, 100):ellipse (10, 10)
ellipse (10, 10)
```

This will draws 2 circles of diameter 10, the first one with center at (50,100), and the center one with center still at (0,0).

2.0.4 Example code

```
polygon(                // we would like to draw a polygon
    0, 0,               // list of vertices of the polygon
    0, 10,               // here the vertices will form unit square
    10, 10,
    10, 0
```

```
);
:translate(5, 5):{   // draw a circle inside the square
    ellipse(10, 10);
}
// we'll draw the same square here
// except that this square would be rotated and translated
:translate(20,0):{         // translate it in horizontal direction by 2 units
    :rotate(45):{         // rotate it counter-clockwise by 45 degree
        polygon(
            0, 0,
            0, 10,
            10, 10,
            10, 0
        );
        :translate(0.5,0.5):{
            ellipse(10, 10);
        }
    }
}
```

will produce the image shown in Figure 2.1. For more complicated examples with control flow, please refer to Appendix D.

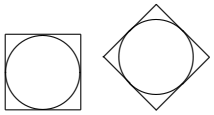

Figure 2.1: Result of the example code

Chapter 3

Language Reference Manual

3.1 Lexicon

SIGL uses a standard grammar and character set. The specific elements that comprise this grammar and character set are described in the following sections:

- Character set (Section 3.1.1)
- Rules for identifiers (Section 3.1.2)
- Use of comments in a program (Section 3.1.3)
- Keywords (Section 3.1.4)
- Operators (Section 3.1.5)
- Interpretation of constant values (Section 3.1.6)

SIGL compiler treats source code as a stream of characters. These characters are grouped into tokens, which can be punctuators, operators, identifiers, keywords, or constants. Tokens are the smallest lexical element of the language. The compiler forms the longest token possible from a given string of characters; the token ends when white space is encountered, or when the next character could not possibly be part of the token.

White space can be a space character, new-line character, tab character, form-feed character, or vertical tab character. Comments are also considered white space. Section 3.1.1 lists all the white space characters. White space is used only as a token separator, but is otherwise ignored in the character stream, and is used mainly for human readability.

In order to simplify the complexity of implementing the language, we choose not to support strings in SIGL. Being a drawing language, this limitation is acceptable in most cases.

3.1.1 Character set

SIGL supports only a small subset of ASCII characters as follows:

- The 26 lowercase Roman characters
  `a b c d e f g h i j k l m n o p q r s t u v w x y z`
- The 26 uppercase Roman characters
  `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

- The 10 decimal digits
  `0 1 2 3 4 5 6 7 8 9`
- The 23 graphic characters
  `! % & * ( ) - _ = + ; / | \ { } [ ] , . < >`
- White spaces: space, horizontal tab, new line, carriage return

3.1.2 Identifiers

An *identifier* in SIGL is a sequence of characters consisting of one or more uppercase or lowercase alphabetic characters, the digits from 0 to 9, and the underscore character (_). Identifiers must not starts with a digit. SIGL identifiers are case-sensitive, i.e. test1, TEST1, Test1 are 3 different identifiers. In general, any sequence of characters satisfying the previous rules can be used as identifier, except for some reserved words known as *keywords*(see Section 3.1.4). An identifier represents a name for variables and functions.

3.1.3 Comments

SIGL supports 2 commonly used comment styles:

- Conventional C multi-line comments: the `/*` (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the `*/` characters. This syntax is the same as ANSI C. Comments of this type are not allowed to be nested.
- C++ single-line comments: the // (two slashes) characters, followed by any sequence of characters. A new line not terminates this form of comment.

3.1.4 Keywords

Keywords are special sequences of characters reserved by compiler, and are not allowed to be used as identifiers. Table 3.1 shows all the keywords used in SIGL

break	else	return
continue	for	while
do	if	function

Table 3.1: SIGL keywords

3.1.5 Operators

SIGL supports almost all unary, binary, and ternary operators in C, both arithmetic and logical operators. Summary of all operators, their precedences and associativity are shown in Table 3.2.

3.1.6 Constants

There are 2 types of constants in SIGL, integer constants (such as $63, 0, 42$), and floating-point constants (such as $1.2, 0.00, 77E + 2$).

Integer constants are used to represent whole numbers. To specify a decimal integer constant, use a sequence of decimal digits $0 \ldots 9$. Value of a decimal constant is computed

Precedence	Operator	Description	Examples	Associativity
1	()	Grouping operator	(a + b) / 4	left to right
	[]	Array access	a[i]	
2	!	Logical negation	if (!done)	right to left
3	*	Multiplication	i = 2 * 4	left to right
	/	Floating point division	i = 9 / 4	
		Integer division	i = 9 2	
	%	Modulo	i = 9 % 2	
4	+	Addition	i = 3 + 4	left to right
	–	Subtraction	i = 4 – 3	
5	<	Comparison less-than	if (i < 42)	left to right
	>	Comparison greater-than	if (i > 42)	
	<=	Comparison less-than-or-equal	if (i <= 42)	
	>=	Comparison greater-than-or-equal	if (i >= 42)	
6	<	Comparison equal-to	if (i == 42)	left to right
	<	Comparison not-equal-to	if (i != 42)	
7	&&	Logical AND	if (a && b)	left to right
8	||	Logical OR	if (a || b)	left to right

Table 3.2: Summary of SIGL operators

in base 10. Integer constant values are always nonnegative; a preceding minus sign is interpreted as a unary operator, not as part of the constant.

A floating-point constant has a fractional or exponential part. Floating-point constants are always interpreted in decimal radix (base 10). Floating-point constants can be expressed with decimal point notation, signed exponent notation, or both. A decimal point without a preceding or following digit is not allowed (for example, .E1 is illegal).

The significand part of the floating-point constant (the whole number part, the decimal point, and the fractional part) may be followed by an exponent part, such as 32.45E2 . The exponent part (in the previous example, E2) indicates the power of 10 by which the significand part is to be scaled.

3.2 Declaration

3.2.1 Variables

SIGL deploys dynamic type system and does not require explicit variable declaration. Variable types are determined at runtime.

3.2.2 Functions

The syntax for declaring a function is as follows

```
function function-name (formal-list)
{
    statement*
}
```

The function will take formal list as its parameters (this can be empty). The type of the function depends on the type of the return value yielded from executing function body. Recursive function is also supported. Functions are not allowed to be nested.

3.3 Program execution

Program execution is from top to bottom. Function definitions are not part of the execution. However, a function must be defined before it can be called.

3.4 Expressions

Primary expressions

identifier

An identifier is a primary expression.

constant

An integer or floating-point constant is a primary expression. Its type is `int` in the former case, and is `double` in the latter.

(expression)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

primary-expression [expression]

A primary expression followed by an expression in square brackets is a primary expression. This is used to refer to an element of an associative array. Its type is the type of the object stored in the associative array.

primary-expression (expression-list)

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in SIGL is strictly by value. A function may change the values of its formal parameters, but these changes cannot possibly affect the values of the actual parameters. Recursive calls to any function are permissible. The type of the function call is the type of its return value.

3.4.1 Unary operators

– expression

The result is the negative of the expression, and has the same type. The type of the expression must be int, or double.

!expression

The result of the logical negation operator ! is `true` if the value of the `expression` is false, and `false` if the value of the `expression` is true. The type of the result is `boolean`. Expression must also have type `boolean`.

3.4.2 Multiplicative operators

expression * expression

The binary * operator indicates multiplication. If both operands are int, the result is int; if one is double, the other is converted to double, and the result is double; if both are double, the result is double. No other combination is allowed.

expression / expression

The binary / operator indicates floating-point division. Both operands, if not already of type double, are converted to double. The result is double.

expression \ expression

The binary \ operator indicates integer division. Both operands must be of type int. The result is of type int.

expression % expression

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be int, and the result is int. In the current implementation, the remainder has the same sign as the dividend.

3.4.3 Additive operators

expression + expression

The result is the sum of the expressions. If both operands are int, the result is int. If both are double, the result is double. If one is int, it is converted to double and the result is double. No other type combinations are allowed.

expression - expression

The result is the difference of the operands. If both operands are int, or double, the same type considerations as for + apply.

3.4.4 Relational operators

expression < expression

expression > expression

expression <= expression

expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield false if the specified relation is not valid and true if it is satisfied. Both operands must be of boolean type.

3.4.5 Equality operators

expression == expression

expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence (Thus `a < b == c < d` is true whenever a < b and c < d have the same truth value). However, they can be used to compare any 2 objects of same type, both must be boolean, or both must be numeric. Comparison between int and double or double and double might yield unexpected result.

3.4.6 expression && expression

The && operator returns `true` if both its operands are `true`, `false` otherwise. The second operand is not evaluated if the first operand is `false`. Both operands must have boolean type.

3.4.7 expression || expression

The || operator returns `true` if either of its operands is `true`, and `false` otherwise. The second operand is not evaluated if the value of the first operand is `true`. Both operands must have boolean type.

3.4.8 Assignment operator lvalue = expression

The value of the expression replaces that of the object referred to by the lvalue. The type of the expression is the type of expression. The type of lvalue is also changed to type of expression after the assignment.

3.5 Statements

3.5.1 Expression statement

Most statements are expression statements, which have the form

```
expression ;
```

Usually expression statements are assignments or function calls.

3.5.2 Compound statement

So that several statements can be used where one is expected, the compound statement is provided:

```
compoundstatement:
    { statement* }
```

3.5.3 Conditional statement

The two forms of the conditional statement are

```
if ( expression ) statement
if ( expression ) statement else statement
```

In both cases the `expression` is evaluated and if it is true, the first substatement is executed. In the second case the second substatement is executed if the expression is false. As usual the `else` ambiguity is resolved by connecting an else with the last encountered elseless `if`.

3.5.4 While statement

The while statement has the form

```
while ( expression ) statement
```

The substatement is executed repeatedly so long as the value of the expression remains true. The test takes place before each execution of the statement.

3.5.5 For statement

The for statement has the form

```
for ( expression1 ; expression2 ; expression3 ) statement
```

This statement is equivalent to

```
expression1;
while (expression2) {
    statement
    expression3;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes false; the third expression typically specifies an incrementation which is performed after each iteration. Any or all of the expressions may be dropped. A missing `expression2` makes the implied while clause equivalent to `while(true)`; other missing expressions are simply dropped from the expansion above.

3.5.6 Break statement

The statement

```
break ;
```

causes termination of the smallest enclosing `while` or `for` statement; control passes to the statement following the terminated statement.

3.5.7 Continue statement

The statement

```
continue ;
```

causes control to pass to the loop continuation portion of the smallest enclosing `while`, or `for` statement; that is to the end of the loop.

3.5.8 Return statement

A function returns to its caller by means of the return statement, which has one of the forms

```
return ;
return ( expression ) ;
```

In the first case no value is returned. In the second case, the value of the expression is returned to the caller of the function. Flowing off the end of a function is equivalent to a return with no returned value.

3.5.9 Transformation

Sometimes we would like to localize the effect of some transformation. This can be done using the following syntax

```
:transformation: statement
```

in which the transformation effect will only apply to the specified `statement`, and not any drawing command following it.

Chapter 4

Project Plan

4.1 Team responsibility

Member	Responsibility
Phong Pham	Team leader; overall architecture, evaluator; testing
Abelardo Gutierrez	Front-end, lexer, parser, walker; testing
Alketa Aliaj	Testing

4.2 Coding conventions

4.2.1 ANTLR coding style

If an ANTLR rule is short and contains only one choice without any action, then it will be written in one line. The colon ":" always starts at 1 tab unless the string in front of ":" is too long. In the one line mode. ";" is at the end of the line.

If an ANTLR rule has multiple choice or actions, the ";" is placed in a separated line at the 1 tab indent. "|" is also at 1 tab. Short actions are in the same line of its rule and at the right half of the screen. Long action lines start at 2 tabs column of the next line. Code in actions follows the Java coding style.

Lexem (token) names are in upper cases and syntactic items (non-terminals) are in lower cases, which may contain the underscore.

4.2.2 Java coding style

We mostly follow the standard code conventions available at http://java.sun.com/docs/codeconv/. We highlight some of the important points that we notice

- Indentation and spacing:
  - Indentation of each level is a tab size.
  - The left brace { occupies a full line and is at the same column as the first character of the previous text.
  - The right brace } occupies a full line and at the same column of its corresponding {.
  - For function calls and function definition, one space between function name and (. Arguments are separated by commas, and there must be a space after each comma.

  - Add spaces between operators and operands for outer expressions.

- Names:
  - Class names are English words whose first characters are capital.
  - Constant are in upper cases, and words are separated by underscore.
  - Always use meaningful names for functions and variable. Avoid using short form, except the short form is very commonly used (for example statement → stmt)
  - Variable and method names are English words whose first characters are capital except for the first word.

- Comments: follow JavaDoc standard for comments.

4.2.3 Test SIGL file coding style

Similar to Java coding style.

4.3 Project timeline

4.4 Developing environment

We allow each member to choose his/her own developing environment that he/she finds most comfortable with. Therefore we don't specify any specific IDE or operating system.

4.4.1 Java 1.4.2

For maximum compatibility, we decided to use Java version 1.4.2.

4.4.2 ANTLR

As required in project specification, we use the latest version ANTLR (version 3.0) to generate our scanner, lexer, parser and walker.

4.4.3 CVS

We use CVS as our version management system. Our repository is located on Columbia UNIX server `cunix.cc.columbia.edu`. It is backup everyday to many different servers that we have access to.

4.4.4 JUnit

We use JUnit (version 4.3.1) to automate unit testing.

4.4.5 Apache Ant

We automate all tasks done in the project by Apache Ant script.

4.5 Project Log

February 07	Discuss and choose the common working environment
February 09	Finish write-up common guideline on how to use tools (Ant, JUnit, etc)
February 21	Start working on some preliminary implementation of lexer/parser
March 12	Decide on overall design and architecture
March 13	Establish test drive, stubs
March 19	Finish most coding
March 21	Coding finished. Introduce some more features to the language.
March 26	Start peer-review testing.
April 31	Finalize all codes

Chapter 5

Architectural Design

5.1 Overview

The overall architecture of our interpreter is shown in Figure 5.1. First the SIGL source code will be scanned, and parsed into a default AST tree generated by ANTLR. This tree is then walked by a walker and build another tree in which each node is our own classes. This allows easier manipulation later on. This final tree is then fed into the evaluator. Even though this is a drawing language, we still provide extra console output besides main image output, for debugging and test purpose.

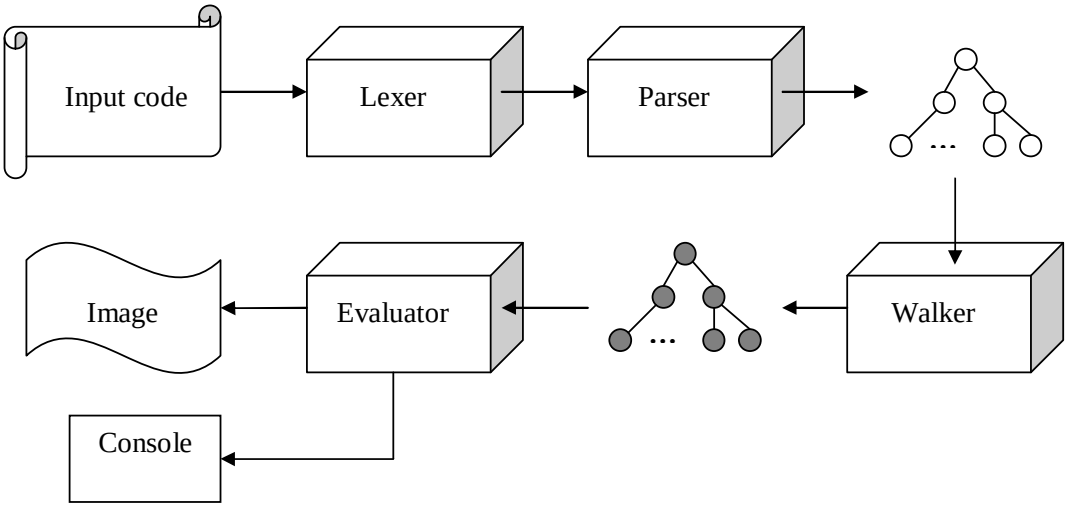

Figure 5.1: Overall design

5.2 Class hierarchy

Classes in our implementation are divided into 2 types: syntax representation classes, and evaluated value classes. The class hierarchy is shown in Figure 5.2.

Syntax representation classes all inherited from a single class call `Node`. `Node` is then split into 2 types, namely Expr and Stmt. The main difference between Expr and Stmt is that Expr has return value, and Stmt does not. Control flow constructions (such as `if`,

`while`) and different types of expressions are given their own classes. This allows maximum extendibility of the language in the future.

Evaluated value classes are the result of program execution and evaluation. All classes of this type inherits from `Value` class. Each value class represents a possible type in the system (except for `ThunkValue` which is used for normal-order evaluation purpose).

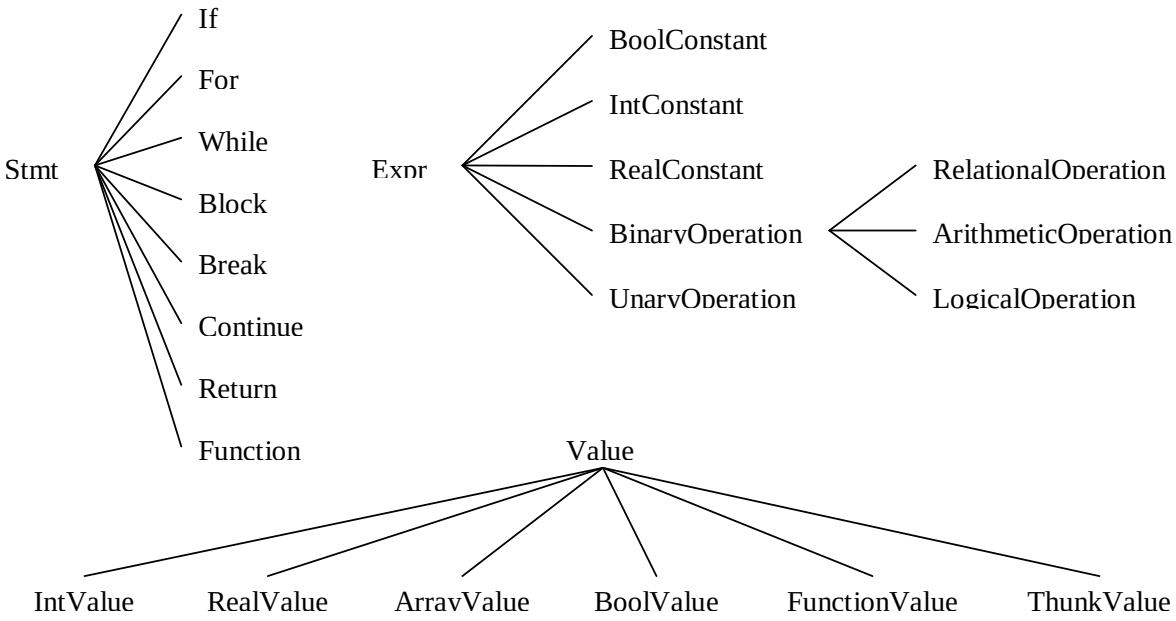

Figure 5.2: Summary class hierarchy

5.3 Symbol table

The following code shows the desired behavior of an SIGL program

```
x = 1; // x is bound to 1
{
    x = 5; // x is bound to 5
    y = 6; // x is bound to 5, y is bound to 6
}
// x is bound to 5, y is unbound
```

When a new block is created, the current environment is cloned and used in the new block. When the block execution terminates, the cloned environment is erased and we revert back to the original environment. However, the change to value of x in the cloned environment should be reflected in the original environment. This cannot be done with direct mapping from name to value in symbol table (it can be done but can be very complicated). We remedy this problem by introducing another layer of abstraction, which we call "stub". Figure 5.3 shows the layout as well as the operations on symbol table.

The extend operation is used then we want to bind a name to a new variable. Normal assignment only requires normal extend operation, which points the stub to the new value. However, there are cases in which we need to shadow the variable of the old environment. This is the case for argument binding in function call, and recursive function name

binding. This behavior requires a special operation on symbol table which we call extend destructively, in which a totally new stub is created, and point to the new value.

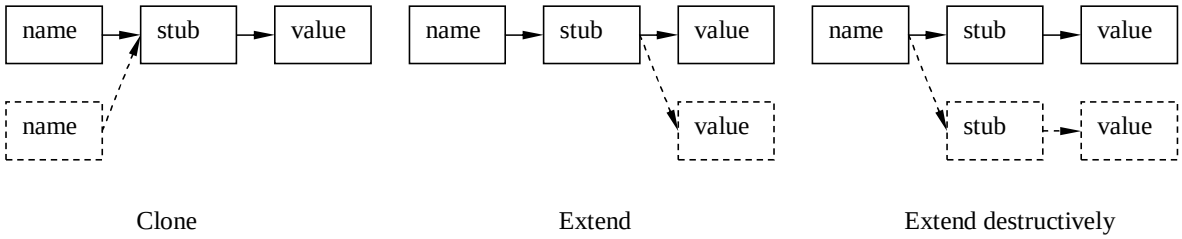

Figure 5.3: Operations on symbol table

Chapter 6

Testing Plan

6.1 Unit testing

Each member is responsible for performing unit-testing on his/her own code. We perform unit testing by using JUnit framework, which automate nearly the whole process. It also allows us to localize the testing to some specific parts of the project instead of running all unit tests at once.

6.2 Peer review

Since the purpose of the project is for the members to learn more about different programming language concept, even though each of each works on a different part of the project, we would like everyone to know what is going on in others' part. That's why we were much stressing on peer-review testing. Peer-review testing not only helps testing the code, but also helps identify mis-understanding (which cannot be found using unit testing). It also gives other team members an overall look at the whole project, what are the difficulties faced in different part of the project. To us this is very valuable, in the sense that it helps us learn more about many different aspect of a programming language.

6.3 Integrated testing

As the final stage, we test our implementation on many different programs with complicated implementation and make sure that our program still work well.

Chapter 7

Lessons Learned

During the course of the project, we ran into the following problems which we might try to overcome better in future projects

- Even though we have settled the overall design before hand, there are many implementation difficulties that we didn't see before hand. Many times we need to go back and revise some of the design decision we made in other to facilitate the implementation better.
- Team work overhead is a bit large. Different team members can be at different points, and sometimes mis-communication occurred.

Appendix A

Directory structure

```
sigl                              main project directory
|-- bin                           contains binary class files
|   `-- src                       contains source code generated by ANTLR
|-- conf                          configuration files
|-- docs                          documentations
|   `-- api                       JavaDoc API
|-- lib                           library used in the project, including ANTLR and JUnit
|-- src                           main source directory
|   |-- antlr                     ANTLR source files
|   |   |-- main                  contain ANTLR source codes for lexer/parser/walker
|   |   `-- test                  test code for ANTLR
|   `-- java                      Java source files
|       |-- main                  main code
|       |   `-- sigl              sigl package
|       |       |-- builtin       package for built-in functions
|       |       |   |-- draw      built-in drawing functions
|       |       |   |-- math      built-in maths functions
|       |       |   `-- transform built-in transformation
|       |       |-- core          core classes (AST nodes, Environment, Value, etc.)
|       |       |-- parser        utility class used during parsing
|       |       `-- util          additional utility classes
|       `-- test                  test codes
`-- test_files                    test files used during test
```

Appendix B

File organization

```
sigl
|-- bin
|   `-- src
|-- build.xml
|-- conf
|-- docs
|   `-- api
|-- lib
|   |-- antlr.jar
|   `-- junit.jar
|-- src
|   |-- antlr
|   |   |-- main
|   |   |   |-- grammar.g
|   |   |   `-- walker.g
|   |   `-- test
|   `-- java
|       |-- main
|       |   `-- sigl
|       |       |-- Evaluator.java
|       |       |-- builtin
|       |       |   |-- BuiltinFunction.java
|       |       |   |-- IsArrayFunction.java
|       |       |   |-- LengthFunction.java
|       |       |   |-- MyFunction.java
|       |       |   |-- PrintFunction.java
|       |       |   |-- draw
|       |       |   |   |-- ColorFunction.java
|       |       |   |   |-- Ellipse.java
|       |       |   |   |-- Line.java
|       |       |   |   `-- Polygon.java
|       |       |   |-- math
|       |       |   |   |-- AcosFunction.java
|       |       |   |   |-- AsinFunction.java
|       |       |   |   |-- AtanFunction.java
|       |       |   |   |-- CeilFunction.java
|       |       |   |   |-- CosFunction.java
|       |       |   |   |-- ExpFunction.java
|       |       |   |   |-- FloorFunction.java
```

```
|       |       |   |   |-- LogFunction.java
|       |       |   |   |-- RandomFunction.java
|       |       |   |   |-- RoundFunction.java
|       |       |   |   |-- SinFunction.java
|       |       |   |   |-- SqrtFunction.java
|       |       |   |   `-- TanFunction.java
|       |       |   `-- transform
|       |       |       |-- Rotate.java
|       |       |       |-- Scale.java
|       |       |       `-- Translate.java
|       |       |-- core
|       |       |   |-- ArithmeticOperation.java
|       |       |   |-- ArrayValue.java
|       |       |   |-- Assign.java
|       |       |   |-- BinaryOperation.java
|       |       |   |-- Block.java
|       |       |   |-- BoolConstant.java
|       |       |   |-- BoolValue.java
|       |       |   |-- Break.java
|       |       |   |-- Continue.java
|       |       |   |-- Environment.java
|       |       |   |-- Expr.java
|       |       |   |-- ExprStmt.java
|       |       |   |-- For.java
|       |       |   |-- Function.java
|       |       |   |-- FunctionCall.java
|       |       |   |-- FunctionValue.java
|       |       |   |-- If.java
|       |       |   |-- IntConstant.java
|       |       |   |-- IntValue.java
|       |       |   |-- LogicalOperation.java
|       |       |   |-- Node.java
|       |       |   |-- NotWellTypedError.java
|       |       |   |-- NumericValue.java
|       |       |   |-- RealConstant.java
|       |       |   |-- RealValue.java
|       |       |   |-- RelationalOperation.java
|       |       |   |-- Return.java
|       |       |   |-- Stmt.java
|       |       |   |-- ThunkValue.java
|       |       |   |-- Transform.java
|       |       |   |-- UnaryOperation.java
|       |       |   |-- Value.java
|       |       |   |-- Variable.java
|       |       |   `-- While.java
|       |       |-- parser
|       |       |   `-- CommonASTWithLine.java
|       |       `-- util
|       |           |-- CommandLineUtil.java
|       |           `-- InvalidOptionError.java
|       `-- test
`-- test_files
    |-- fractal_dragon_curve.sigl
    |-- fractal_hilbert_curve.sigl
```

```
|-- fractal_koch_curve.sigl
|-- simple.sigl
|-- test1.sigl
|-- test2.sigl
|-- test3.sigl
|-- dynamic.sigl
|-- firstorder.sigl
|-- passbyneed.sigl
|-- primivite.sigl
`-- recursive.sigl
```

Appendix C

Building the project

All tasks in the project, invoking ANTLR for generating lexer/parser/walker codes, compiling, testing using JUnit, generating JavaDoc, packaging, are done automatically using Apache Ant automated script. Table C.1. The source for `build.xml` Apache Ant script is included in Appendix D.

Apache Ant target	Task description	Dependency
init	prepare the directory structure	
compile	invoke incremental build	init,antlr
antlr	generate source codes for lexer/parser/walker	init
clean	clean all binary class files, ANTLR generated files	
build	start a clean build of the whole project	clean,compile
doc	generate JavaDoc documents	
jar	package the project into a single Jar file	build

Table C.1: Apache Ant tasks

Appendix D

Code listings

Please refer to the code listing index at the end of the report for fast pointers to specific files. The order of the files in this Appendix follows that of Appendix B rather than alphabetical order.

```
<?xml version="1.0" ?>
<project name="testproj" default="compile" basedir=".">

    <property name="src" location="src"/>
    <property name="bin" location="bin"/>
    <property name="gensrc" location="${bin}/src"/>
    <property name="lib" location="lib"/>
    <property name="dist" location="dist"/>
    <property name="doc" location="docs/api"/>
    <property name="jar" value="${dist}/sigl.jar"/>

    <target name="init">
        <tstamp/>
        <mkdir dir="${lib}"/>
        <mkdir dir="${bin}"/>
        <mkdir dir="${gensrc}"/>
    </target>

    <target name="compile" depends="init,antlr">
        <javac debug="on" deprecation="off" srcdir="${src}:${gensrc}"
            destdir="${bin}" source="1.4" target="1.4">
            <compilerarg value="-Xlint:unchecked"/>
            <compilerarg value="-Xlint:deprecation"/>
            <classpath>
                <pathelement path="${classpath}"/>
                <fileset dir="${lib}">
                    <include name="**/*.jar"/>
                </fileset>
            </classpath>
        </javac>
    </target>

    <target name="antlr" depends="init">
        <antlr
            target="${src}/antlr/main/grammar.g"
            outputdirectory="${gensrc}">
            <classpath>
                <pathelement location="${lib}/antlr.jar"/>
            </classpath>
        </antlr>
        <antlr
            target="${src}/antlr/main/walker.g"
            outputdirectory="${gensrc}">
            <classpath>
                <pathelement location="${lib}/antlr.jar"/>
            </classpath>
        </antlr>
    </target>

    <target name="clean">
        <delete dir="${bin}"/>
        <delete dir="${dist}"/>
        <delete dir="${doc}"/>
    </target>

    <target name="build" depends="clean,compile"/>

    <target name="jar" depends="build">
        <mkdir dir="${dist}"/>
        <jar destfile="${jar}">
            <fileset dir="${bin}">
                <exclude name="src/"/>
            </fileset>
        </jar>
```



```
    </target>

    <target name="doc">
        <javadoc destdir="${doc}"
            author="true"
            version="true"
            use="true"
            windowtitle="Simple Image Generation Language">
            <fileset dir="${src}" defaultexcludes="yes">
            </fileset>
        </javadoc>
    </target>
</project>
```

```
package sigl.builtin;

import java.util.*;

import sigl.core.*;

public abstract class BuiltinFunction
extends FunctionValue
{
    protected Vector deThunk (Vector arguments)
    {
        Vector args = new Vector();
        for (ListIterator iter = arguments.listIterator();iter.hasNext();)
        {
            Value value = (Value)iter.next();
            if (value instanceof ThunkValue)
            {
                args.add (
                    ((ThunkValue)value).getExpr().
                        evaluate (((ThunkValue)value).getEnvironment())
                );
            }
            else
            {
                args.add (value);
            }
        }
        return args;
    }

} // class PrintFunction
```

```
package sigl.builtin.draw;

import java.util.*;

import sigl.core.*;

public class ColorFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("color", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 3)
            throw new NotWellTypedError (
                "Function \"color\" requires 3 argument, found " +
                arguments.size());
        int colors[] = new int[3];
        for (int i = 0;i < 3;++i)
        {
            Value v = (Value)arguments.elementAt (i);
            if (v instanceof IntValue) colors[i] = ((IntValue)v).getValue();
            else throw new NotWellTypedError (
                "Function \"color\" only takes integer argument"
            );
        }
        _env.getGraphics().setColor (new java.awt.Color (colors[0], colors[1], c
olors[2]));
        return null;
    }

} // class ColorFunction
```

```
package sigl.builtin.draw;

import java.util.*;
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;

import sigl.core.*;

public class Ellipse
extends sigl.builtin.BuiltinFunction
{
    public Ellipse ()
    {
    }

    public Function getFunction ()
    {
        return new Function ("ellipse", null, null);
    }

    public Value apply (Vector arguments, Environment env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 2)
            throw new NotWellTypedError (
                "Function \"ellipse\" requires 2 arguments, found " +
                arguments.size());
        double size[] = new double[4];
        for (int i = 0;i < 2;++i)
        {
            Value v = (Value)arguments.elementAt (i);
            if (v instanceof IntValue) size[i] = ((IntValue)v).getValue();
            else if (v instanceof RealValue) size[i] = ((RealValue)v).getValue()
;
            else throw new NotWellTypedError (
                "Function \"ellipse\" only take numeric argument"
            );
        }
        Graphics2D g = env.getGraphics();
        g.setTransform (env.getTransform());
        g.draw (new java.awt.geom.Ellipse2D.Double(-size[0] / 2, -size[1] / 2, s
ize[0], size[1]));
        return null;
    }

} // class Ellipse
```

```
package sigl.builtin.draw;

import java.util.*;
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;

import sigl.core.*;

public class Line
extends sigl.builtin.BuiltinFunction
{
    public Line ()
    {
    }

    public Function getFunction ()
    {
        return new Function ("line", null, null);
    }

    public Value apply (Vector arguments, Environment env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 4)
            throw new NotWellTypedError (
                "Function \"line\" requires 4 arguments, found " +
                arguments.size());
        double coordinates[] = new double[4];
        for (int i = 0;i < 4;++i)
        {
            Value v = (Value)arguments.elementAt (i);
            if (v instanceof IntValue) coordinates[i] = ((IntValue)v).getValue()
;
            else if (v instanceof RealValue) coordinates[i] = ((RealValue)v).get
Value();
            else throw new NotWellTypedError (
                "Function \"line\" only take numeric argument"
            );
        }
        Graphics2D g = env.getGraphics();
        g.setTransform (env.getTransform());
        g.draw (
            new java.awt.geom.Line2D.Double(
                    (int)coordinates[0],
                    (int)coordinates[1],
                    (int)coordinates[2],
                    (int)coordinates[3])
        );
        return null;
    }

} // class Line
```

```
package sigl.builtin.draw;

import java.util.*;
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;

import sigl.core.*;

public class Polygon
extends sigl.builtin.BuiltinFunction
{
    public Polygon ()
    {
    }

    public Function getFunction ()
    {
        return new Function ("polygon", null, null);
    }

    public Value apply (Vector arguments, Environment env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() % 2 != 0)
            throw new NotWellTypedError (
                "Function \"polygon\" requires even number of arguments, found " +
                arguments.size());
        double coordinates[] = new double[arguments.size()];
        for (int i = 0;i < coordinates.length;++i)
        {
            Value v = (Value)arguments.elementAt (i);
            if (v instanceof IntValue) coordinates[i] = ((IntValue)v).getValue()
;
            else if (v instanceof RealValue) coordinates[i] = ((RealValue)v).get
Value();
            else throw new NotWellTypedError (
                "Function \"polygon\" only take numeric argument"
            );
        }
        Graphics2D g = env.getGraphics();
        for (int i = 0;i < coordinates.length;i+=2)
        {
            int j = i + 2;
            if (j >= coordinates.length) j = 0;
            g.setTransform (env.getTransform());
            g.draw (
                new java.awt.geom.Line2D.Double(
                        (int)coordinates[i],
                        (int)coordinates[i + 1],
                        (int)coordinates[j],
                        (int)coordinates[j + 1])
            );
        }
        return null;
    }

} // class Polygon
```

```
package sigl.builtin;

import java.util.*;

import sigl.core.*;

public class IsArrayFunction
extends FunctionValue
{
    public Function getFunction ()
    {
        return new Function ("isarray", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"isarray\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof ArrayValue) return new BoolValue (true);
        else return new BoolValue(false);
    }

} // class IsArrayFunction
```

```
package sigl.builtin;

import java.util.*;

import sigl.core.*;

public class LengthFunction
extends FunctionValue
{
    public Function getFunction ()
    {
        return new Function ("length", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"length\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof ArrayValue)
        {
            return new IntValue (((ArrayValue)value).size());
        }
        else
            throw new NotWellTypedError (
                "Function \"length\" only takes array argument"
            );
    }

} // class LengthFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class AcosFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("acos", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"acos\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new RealValue (Math.acos (((IntValue)value).getValue()));
        }
        else if (value instanceof RealValue)
        {
            return new RealValue (Math.acos (((RealValue)value).getValue()));
        }
        else
            throw new NotWellTypedError (
                "Function \"acos\" only take numeric argument"
            );
    }

} // class AcosFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class AsinFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("asin", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"asin\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new RealValue (Math.asin (((IntValue)value).getValue()));
        }
        else if (value instanceof RealValue)
        {
            return new RealValue (Math.asin (((RealValue)value).getValue()));
        }
        else
            throw new NotWellTypedError (
                "Function \"asin\" only take numeric argument"
            );
    }

} // class AsinFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class AtanFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("atan", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"atan\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new RealValue (Math.atan (((IntValue)value).getValue()));
        }
        else if (value instanceof RealValue)
        {
            return new RealValue (Math.atan (((RealValue)value).getValue()));
        }
        else
            throw new NotWellTypedError (
                "Function \"atan\" only take numeric argument"
            );
    }

} // class AtanFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class CeilFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("ceil", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"ceil\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new IntValue ((int)Math.ceil (((IntValue)value).getValue()));
        }
        else if (value instanceof RealValue)
        {
            return new IntValue ((int)Math.ceil (((RealValue)value).getValue()))
;
        }
        else
            throw new NotWellTypedError (
                "Function \"ceil\" only take numeric argument"
            );
    }

} // class CeilFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class CosFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("cos", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"cos\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new RealValue (Math.cos (((IntValue)value).getValue()));
        }
        else if (value instanceof RealValue)
        {
            return new RealValue (Math.cos (((RealValue)value).getValue()));
        }
        else
            throw new NotWellTypedError (
                "Function \"cos\" only take numeric argument"
            );
    }

} // class CosFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class ExpFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("exp", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"exp\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new RealValue (Math.exp (((IntValue)value).getValue()));
        }
        else if (value instanceof RealValue)
        {
            return new RealValue (Math.exp (((RealValue)value).getValue()));
        }
        else
            throw new NotWellTypedError (
                "Function \"exp\" only take numeric argument"
            );
    }

} // class ExpFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class FloorFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("floor", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"floor\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new IntValue ((int)Math.floor (((IntValue)value).getValue()))
;
        }
        else if (value instanceof RealValue)
        {
            return new IntValue ((int)Math.floor (((RealValue)value).getValue())
);
        }
        else
            throw new NotWellTypedError (
                "Function \"floor\" only take numeric argument"
            );
    }

} // class FloorFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class LogFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("log", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"log\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new RealValue (Math.log (((IntValue)value).getValue()));
        }
        else if (value instanceof RealValue)
        {
            return new RealValue (Math.log (((RealValue)value).getValue()));
        }
        else
            throw new NotWellTypedError (
                "Function \"log\" only take numeric argument"
            );
    }

} // class LogFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class RandomFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("random", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 0)
            throw new NotWellTypedError (
                "Function \"random\" requires no argument, found " +
                arguments.size());
        return new RealValue (Math.random ());
    }

} // class RandomFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class RoundFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("round", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"round\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new IntValue ((int)Math.round (((IntValue)value).getValue()))
;
        }
        else if (value instanceof RealValue)
        {
            return new IntValue ((int)Math.round (((RealValue)value).getValue())
);
        }
        else
            throw new NotWellTypedError (
                "Function \"round\" only take numeric argument"
            );
    }

} // class RoundFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class SinFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("sin", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"sin\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new RealValue (Math.sin (((IntValue)value).getValue()));
        }
        else if (value instanceof RealValue)
        {
            return new RealValue (Math.sin (((RealValue)value).getValue()));
        }
        else
            throw new NotWellTypedError (
                "Function \"sin\" only take numeric argument"
            );
    }

} // class SinFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class SqrtFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("sqrt", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"sqrt\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new RealValue (Math.sqrt (((IntValue)value).getValue()));
        }
        else if (value instanceof RealValue)
        {
            return new RealValue (Math.sqrt (((RealValue)value).getValue()));
        }
        else
            throw new NotWellTypedError (
                "Function \"sqrt\" only take numeric argument"
            );
    }

} // class SqrtFunction
```

```
package sigl.builtin.math;

import java.util.*;

import sigl.core.*;

public class TanFunction
extends sigl.builtin.BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("tan", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"tan\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue)
        {
            return new RealValue (Math.tan (((IntValue)value).getValue()));
        }
        else if (value instanceof RealValue)
        {
            return new RealValue (Math.tan (((RealValue)value).getValue()));
        }
        else
            throw new NotWellTypedError (
                "Function \"tan\" only take numeric argument"
            );
    }

} // class TanFunction
```

```
package sigl.builtin;

import java.util.*;

import sigl.core.*;

public class MyFunction
extends FunctionValue
{
    public Function getFunction ()
    {
        return new Function ("myfunction", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"myfunction\" requires 1 argument, found " +
                arguments.size());
        Value value = (Value)arguments.elementAt (0);
        if (value instanceof IntValue) return new IntValue (((IntValue)value).ge
tValue() * 2);
        throw new NotWellTypedError (
                "Function \"myfunction\" requires an integer argument");
    }

} // class MyFunction
```

```
package sigl.builtin;

import java.util.*;

import sigl.core.*;

public class PrintFunction
extends BuiltinFunction
{
    public Function getFunction ()
    {
        return new Function ("print", null, null);
    }

    public Value apply (Vector arguments, Environment _env)
    throws NotWellTypedError
    {
        arguments = deThunk(arguments);
        for (ListIterator iter = arguments.listIterator();
            iter.hasNext();)
        {
            Value value = (Value)iter.next();
            System.out.print (value);
            if (iter.hasNext()) System.out.print (" ");
        }
        System.out.println ();
        return null;
    }

} // class PrintFunction
```

```
package sigl.builtin.transform;

import java.util.*;
import java.awt.geom.AffineTransform;

import sigl.core.*;

public class Rotate
extends sigl.builtin.BuiltinFunction
{
    public Rotate ()
    {
    }

    public Function getFunction ()
    {
        return new Function ("rotate", null, null);
    }

    public Value apply (Vector arguments, Environment env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 1)
            throw new NotWellTypedError (
                "Function \"rotate\" requires 1 argument, found " +
                arguments.size());
        double angle = 0;
        Value v = (Value)arguments.elementAt (0);
        if (v instanceof IntValue) angle = ((IntValue)v).getValue();
        else if (v instanceof RealValue) angle = ((RealValue)v).getValue();
        else throw new NotWellTypedError (
            "Function \"rotate\" only take numeric argument"
        );
        env.getTransform().rotate(angle / 180 * Math.PI);
        return null;
    }

} // class Rotate
```

```
package sigl.builtin.transform;

import java.util.*;
import java.awt.geom.AffineTransform;

import sigl.core.*;

public class Scale
extends sigl.builtin.BuiltinFunction
{
    public Scale ()
    {
    }

    public Function getFunction ()
    {
        return new Function ("scale", null, null);
    }

    public Value apply (Vector arguments, Environment env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 2)
            throw new NotWellTypedError (
                "Function \"scale\" requires 2 argument, found " +
                arguments.size());
        double coordinates[] = new double[2];
        for (int i = 0;i < 2;++i)
        {
            Value v = (Value)arguments.elementAt (i);
            if (v instanceof IntValue) coordinates[i] = ((IntValue)v).getValue()
;
            else if (v instanceof RealValue) coordinates[i] = ((RealValue)v).get
Value();
            else throw new NotWellTypedError (
                "Function \"scale\" only take numeric argument"
            );
        }
        env.getTransform().scale(coordinates[0], coordinates[1]);
        return null;
    }

} // class Scale
```

```
package sigl.builtin.transform;

import java.util.*;
import java.awt.geom.AffineTransform;

import sigl.core.*;

public class Translate
extends sigl.builtin.BuiltinFunction
{
    public Translate ()
    {
    }

    public Function getFunction ()
    {
        return new Function ("translate", null, null);
    }

    public Value apply (Vector arguments, Environment env)
    throws NotWellTypedError
    {
        arguments = deThunk (arguments);
        if (arguments.size() != 2)
            throw new NotWellTypedError (
                "Function \"translate\" requires 2 argument, found " +
                arguments.size());
        double coordinates[] = new double[2];
        for (int i = 0;i < 2;++i)
        {
            Value v = (Value)arguments.elementAt (i);
            if (v instanceof IntValue) coordinates[i] = ((IntValue)v).getValue()
;
            else if (v instanceof RealValue) coordinates[i] = ((RealValue)v).get
Value();
            else throw new NotWellTypedError (
                "Function \"translate\" only take numeric argument"
            );
        }
        env.getTransform().translate(coordinates[0], coordinates[1]);
        return null;
    }

} // class Translate
```

```
package sigl.core;

/**
 * Represents arithmetic operation, including +, -, *, /, and modulo
 */
public class ArithmeticOperation
extends BinaryOperation
{
    /**
     * Default empty constructor
     */
    public ArithmeticOperation ()
    {
    }

    /**
     * Constructs an arithmetic operation given the operation in form of string,
     * the left hand side and right hand side of the operation
     * @param op the operation (can be either "+", "-", "*", "/", or "%")
     * @param lhs the left hand operand
     * @param rhs the right hand operand
     */
    public ArithmeticOperation (String op, Expr lhs, Expr rhs)
    {
        super (op, lhs, rhs);
    }

    /**
     * Evaluates the arithmetic expression
     * @param env the environment of the expression
     * @return the computed value
     */
    public Value evaluate (Environment env)
    {
        // evaluate both operands
        Value l = lhs.evaluate(env);
        Value r = rhs.evaluate(env);
        // type checking
        if (l == null || !(l instanceof NumericValue))
            throw new NotWellTypedError ("Invalid left hand side value", this);
        if (r == null || !(r instanceof NumericValue))
            throw new NotWellTypedError ("Invalid right hand side value", this);
        // if either of them is real value, then perform real operation
        if (l instanceof RealValue || r instanceof RealValue)
        {
            double lvalue = 0;
            double rvalue = 0;
            if (l instanceof IntValue) lvalue = ((IntValue)l).getValue();
            else lvalue = ((RealValue)l).getValue();
            if (r instanceof IntValue) rvalue = ((IntValue)r).getValue();
            else rvalue = ((RealValue)r).getValue();

            if (op.equals ("+")) return new RealValue (lvalue + rvalue);
            if (op.equals ("-")) return new RealValue (lvalue - rvalue);
            if (op.equals ("*")) return new RealValue (lvalue * rvalue);
            if (op.equals ("/")) return new RealValue (lvalue / rvalue);

            if (op.equals ("%"))
            {
                if (l instanceof RealValue)
                    throw new NotWellTypedError ("Left hand side of modulo operation must be
integer", this);
                if (r instanceof RealValue)
                    throw new NotWellTypedError ("Right hand side of modulo operation must b
```



```
e integer", this);
            }
        }
        else
        {
            // otherwise, perform integer operation
            int lvalue = ((IntValue)l).getValue();
            int rvalue = ((IntValue)r).getValue();
            if (op.equals ("+")) return new IntValue (lvalue + rvalue);
            if (op.equals ("-")) return new IntValue (lvalue - rvalue);
            if (op.equals ("*")) return new IntValue (lvalue * rvalue);
            if (op.equals ("/")) return new IntValue (lvalue / rvalue);
            if (op.equals ("%")) return new IntValue (lvalue % rvalue);
        }
        throw new NotWellTypedError ("Undefined operator", this);
    }

} // class ArithmeticOperation
```

```
package sigl.core;

import java.util.*;

/**
 * Represents the array type
 */
public class ArrayValue
implements Value
{
    /** hashtable for mapping between indices and value */
    private Hashtable bindings = new Hashtable();

    /**
     * Constructs an empty array value
     */
    public ArrayValue()
    {
    }

    /**
     * Sets value for a specific index
     * @param key the index
     * @param value the value
     */
    public void set (Value key, Value value)
    {
        bindings.put (key, value);
    }

    /**
     * Retrieves value for a specific index
     * @param key the index
     * @return the corresponding value
     */
    public Value get (Value key)
    {
        return (Value)bindings.get (key);
    }

    /**
     * Gets the number of elements in this array
     */
    public int size()
    {
        return bindings.size();
    }

    /**
     * Provides a meaningful toString() which will be used by built-in
     * print() command for debugging purpose
     */
    public String toString ()
    {
        StringBuffer s = new StringBuffer ("");
        s.append ("[");
        for (Iterator iter = bindings.keySet().iterator();
            iter.hasNext();)
        {
            Value key = (Value)iter.next();
            s.append (key + "=>" + bindings.get (key));
            if (iter.hasNext())
            {
                s.append (",");
```



```
            }
        }
        s.append ("]");
        return s.toString();
    }

} // class ArrayValue
```

```
package sigl.core;

import java.util.*;

/**
 * Represents an assignment in SIGL language. The left-hand-side of the
 * assignment must be a variable. When this assignment is evaluated, a binding
 * of the left-hand variable to the evaluated value of the right-hand-side
 * is created in the environment this assignment resides.
 */
public class Assign
extends Expr
{
    /** Left-hand-side expression */
    private Expr lhs = null;
    /** Right-hand-side expression */
    private Expr rhs = null;

    /**
     * Default empty constructor
     */
    public Assign ()
    {
    }

    /**
     * Constructs an assignment given the 2 side expressions
     * @param lhs the left-hand-side expression
     * @param rhs the right-hand-side expression
     */
    public Assign (Expr lhs, Expr rhs)
    {
        this.lhs = lhs;
        this.rhs = rhs;
    }

    /**
     * Evaluates the assignment. The right-hand-side is first evaluated. This
     * value is then bound to the left-hand-side variable in the environment.
     * @param env the current environment
     * @return the evaluated value of the right-hand-side expression
     */
    public Value evaluate (Environment env)
    {
        // check that the lhs must be a variable
        if (!(lhs instanceof Variable))
            throw new NotWellTypedError ("Left-hand-side of an assignment must be a variable",
this);
        // down cast for easier use
        Variable var = (Variable)lhs;
        // evaluate the right-hand-side expression
        Value value = rhs.evaluate (env);

        // if this is simply a variable, simply direct binding
        if (var.getIndices().isEmpty()) env.extend (var.getName(), value);
        else
        {
            // there are indices, then bind accordingly
            Value v = env.access (var.getName());
            if (!(v instanceof ArrayValue))
            {
                v = new ArrayValue();
                env.extend (var.getName(), v);
            }
```



```
            for (ListIterator iter = var.getIndices().listIterator();
                iter.hasNext();)
            {
                Expr expr = (Expr)iter.next();
                Value index = expr.evaluate (env);
                if (iter.hasNext())
                {
                    if (!(((ArrayValue)v).get(index) instanceof ArrayValue))
                        ((ArrayValue)v).set (index, new ArrayValue());
                    v = ((ArrayValue)v).get(index);
                }
                else
                {
                    ((ArrayValue)v).set (index, value);
                }
            }
        }
        return value;
    }

} // class Assign
```

```java
package sigl.core;

/**
 * An abstract representation for all binary operation (including arithmetic
 * as well as logical operations). Provides utility method for handling
 * set/get on left hand side, right hand side, and the operator.
 */
public abstract class BinaryOperation
extends Expr
{
    /** the operator */
    protected String op = null;
    /** left hand side expression */
    protected Expr lhs = null;
    /** right hand side expression */
    protected Expr rhs = null;

    /**
     * Default empty constructor
     */
    protected BinaryOperation ()
    {
    }

    /**
     * Constructs a binary operation given the operator and 2 side
     * expressions
     * @param op the operator, in string
     * @param lhs the left hand side expression
     * @param rhs the right hand side expression
     */
    protected BinaryOperation (String op, Expr lhs, Expr rhs)
    {
        this.op = op;
        this.lhs = lhs;
        this.rhs = rhs;
    }

    /**
     * Sets the operator
     * @param op the new operator
     */
    public void setOperator (String op)
    {
        this.op = op;
    }

    /**
     * Gets the operator
     * @return the current operator of this operation
     */
    public String getOperator ()
    {
        return op;
    }

    /**
     * Sets the left hand side expression
     * @param expr the new left-hand-side expression
     */
    public void setLHS (Expr expr)
    {
        this.lhs = expr;
    }
```



```java
    /**
     * Gets the left hand side expression
     * @return the left hand side expression
     */
    public Expr getLHS ()
    {
        return this.lhs;
    }

    /**
     * Sets the right hand side expression
     * @param expr the new right hand side expression
     */
    public void setRHS (Expr expr)
    {
        this.rhs = expr;
    }

    /**
     * Gets the right hand side expression
     * @return the right hand side expression
     */
    public Expr getRHS ()
    {
        return this.rhs;
    }

} // class BinaryOperation
```

```
package sigl.core;

import java.util.*;

/**
 * <p>Represents an SIGL block, the part of code surrounded by an opening
 * curly bracket "{" and a closing curly bracket "}".</p>
 * <p>When this block is executed, it clones the current environment, and
 * use the cloned environment to execute statements inside this block.
 */
public class Block
extends Stmt
{
    /** vector storing the statements in this block */
    private Vector stmts = new Vector();

    /**
     * Constructs an empty block
     */
    public Block ()
    {
    }

    /**
     * Adds a statement to the end of this block
     * @param stmt the new statement
     */
    public void addStatement (Stmt stmt)
    {
        stmts.add (stmt);
    }

    /**
     * Gets the number of statements in this block
     * @return the number of statements in this block
     */
    public int size ()
    {
        return stmts.size();
    }

    /**
     * Gets a specific statement from this block
     * @param index the 0-based index of the statement in this block
     * @return the requested statement
     */
    public Stmt getStatement (int index)
    {
        return (Stmt)stmts.elementAt (index);
    }

    /**
     * Execute the block. The current environment is first cloned, and the
     * cloned environment is then used to execute statements inside this block.
     * When the execution of the inside statements finish, continue flag,
     * break flag, return flag and return value will be propagated from the
     * cloned environment to the current environment.
     * @param env the current environment
     */
    public void execute (Environment env)
    {
        Environment dup = env.duplicate();
        for (ListIterator iter = stmts.listIterator();
            iter.hasNext();)
```



```
        {
            Stmt stmt = (Stmt)iter.next();
            stmt.execute (dup);
            if (dup.getContinueFlag())
            {
                env.setContinueFlag (true);
                env.setContinueStmt (dup.getContinueStmt());
                return;
            }
            if (dup.getBreakFlag())
            {
                env.setBreakFlag (true);
                env.setBreakStmt (dup.getBreakStmt());
                return;
            }
            if (dup.getReturnFlag())
            {
                env.setReturnFlag (true);
                env.setReturnValue (dup.getReturnValue());
                return;
            }
        }
    }

} // class Block
```

```
package sigl.core;

/**
 * Represent a boolean constant, either true or false. A BoolConstant
 * is evaluated into a BoolValue with the same boolean value.
 */
public class BoolConstant
extends Expr
{
    /** the value of the constant */
    private boolean value = false;

    /**
     * Default empty constructor
     */
    public BoolConstant ()
    {
    }

    /**
     * Constructs a boolean constant given the value
     * @param value the value of the constant
     */
    public BoolConstant (boolean value)
    {
        this.value = value;
    }

    /**
     * Gets the value of this boolean constant
     * @return the value of this boolean constant
     */
    public boolean getValue ()
    {
        return value;
    }

    /**
     * Sets the value of this boolean constant
     * @param value the new value
     */
    public void setValue (boolean value)
    {
        this.value = value;
    }

    /**
     * Evaluates this boolean constant. Evaluation is trivial, as a
     * BoolConstant is evaluated into a BoolValue with the same
     * boolean value
     */
    public Value evaluate (Environment e)
    {
        return new BoolValue (value);
    }

} // class BoolConstant
```

```
package sigl.core;

/**
 * Represents the boolean type in SIGL
 */
public class BoolValue
implements Value
{
    /** the boolean value */
    private boolean value = false;

    /**
     * Default empty constructor
     */
    public BoolValue ()
    {
    }

    /**
     * Constructs a BoolValue with provided boolean value
     * @param value the value of the newly constructed BoolValue
     */
    public BoolValue (boolean value)
    {
        this.value = value;
    }

    /**
     * Gets the value of this BoolValue
     * @return the boolean value of this BoolValue
     */
    public boolean getValue ()
    {
        return value;
    }

    /**
     * Sets the value of this BoolValue
     * @param the new boolean value of this BoolValue
     */
    public void setValue (boolean value)
    {
        this.value = value;
    }

    /**
     * Computes the hashCode of this BoolValue. This is needed because all Value
 types
     * can be used as indices for associative array.
     * @return the hash code, 1 if this value represents true, and 0 otherwise
     */
    public int hashCode ()
    {
        return (value)?1:0;
    }

    /**
     * Compares this BoolValue with another BoolValue. Both are equal if the int
ernal
     * boolean values are the same.
     * @param o the object to compare to
     */
    public boolean equals (Object o)
    {
```



```
        if (!(o instanceof BoolValue)) return false;
        return (value == ((BoolValue)o).value);
    }

    /**
     * Provides meaningful representation of this BoolValue, used for built-in p
rint
     * command
     */
    public String toString()
    {
        return "" + value;
    }

} // class BoolValue
```

```
package sigl.core;

/**
 * Represents a break statement. When executed, it will turn on the break flag
 * in the current environment.
 */
public class Break
extends Stmt
{
    public Break()
    {
    }

    public void execute (Environment env)
    {
        env.setBreakFlag (true);
        env.setBreakStmt (this);
    }

} // class Break
```

```
package sigl.core;

/**
 * Represents a continue statement. When executed, it will turn on the continue
 * flag in the current environment.
 */
public class Continue
extends Stmt
{
    public Continue()
    {
    }

    public void execute (Environment env)
    {
        env.setContinueFlag (true);
        env.setContinueStmt (this);
    }

} // class Continue
```

```
package sigl.core;

import java.util.*;
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;

/**
 * <p>
 * The environment in SIGL, containing the following
 * <ul>
 *  <li>The symbol table: a mapping between name and corresponding values</li>
 *  <li>The drawing environment, including the current drawing canvas, and the
 *      current transformation.</li>
 *  <li>Break, continue, return flag, return value</li>
 * </ul>
 * </p>
 * <p>
 * The symbol table is stored in form name -> stub -> value instead of
 * traditional name -> value binding. When the environment is clone, each name
 * will be cloned into the new environment. However, the cloned names in the
 * cloned environment still point to the original stubs stored in the original
 * environment.
 * </p>
 */
public class Environment
implements Cloneable
{
    public static boolean dynamicScoping = false;
    public static boolean passByNeed = false;

    /** break flag */
    private boolean breakFlag = false;
    /**
     * this is only meaningful if breakFlag is true, points to the break
     * statement which turned the break flag on
     */
    private Stmt breakStmt = null;
    /** continue flag */
    private boolean continueFlag = false;
    /**
     * this is only meaningful if continueFlag is true, points to the continue
     * statement which turned the continue flag on
     */
    private Stmt continueStmt = null;
    /** return flag */
    private boolean returnFlag = false;
    /**
     * this is only meaningful if returnFlag is true, contains the return value
     * provided in the return statement.
     */
    private Value returnValue = null;

    /** the drawing environment */
    private Graphics2D g = null;

    /** current transformation */
    private AffineTransform m = new AffineTransform();

    /** the symbol table */
    private Hashtable bindings = new Hashtable();

    public boolean getBreakFlag ()
    {
        return breakFlag;
    }
```



```
    public void setBreakFlag (boolean flag)
    {
        breakFlag = flag;
    }

    public boolean getContinueFlag ()
    {
        return continueFlag;
    }

    public void setBreakStmt (Stmt stmt)
    {
        this.breakStmt = stmt;
    }

    public Stmt getBreakStmt ()
    {
        return breakStmt;
    }

    public void setContinueFlag (boolean flag)
    {
        this.continueFlag = flag;
    }

    public boolean getReturnFlag ()
    {
        return returnFlag;
    }

    public void setContinueStmt (Stmt stmt)
    {
        this.continueStmt = stmt;
    }

    public Stmt getContinueStmt ()
    {
        return continueStmt;
    }

    public void setReturnFlag (boolean flag)
    {
        this.returnFlag = flag;
    }

    public Value getReturnValue ()
    {
        return returnValue;
    }

    public void setReturnValue (Value returnValue)
    {
        this.returnValue = returnValue;
    }

    /**
     * Clone this environment. Each name will be cloned into the new environment
.
     * However, the cloned names in the cloned environment still point to the
     * original stubs stored in this environment.
     * @return the cloned environment
     */
```

```
    public Environment duplicate ()
    {
        Environment env = new Environment();
        env.g = g;
        env.m = m;
        env.bindings = (Hashtable)bindings.clone();
        return env;
    }

    /**
     * Gets the bound value for a variable. If the bound value is a thunk,
     * then evaluate the thunk and replace the thunk with the evaluated value
     * inside the symbol table.
     * @param variableName name of the variable
     * @return the bound value for the specified variable name, null if
     *         there is no binding for that name
     */
    public Value access (String variableName)
    {
        ValueStub stub = (ValueStub)bindings.get (variableName);
        if (stub == null) return null;
        Value value = stub.getValue();
        if (value instanceof ThunkValue)
        {
            value = ((ThunkValue)value).getExpr().
                evaluate (((ThunkValue)value).getEnvironment());
            stub.setValue (value);
        }
        return value;
    }

    /**
     * Changes the binding of a name to value. If the name was previously
     * bound, the old value is replaced by the new one. Otherwise, a new stub
     * is created, and point to the new value
     * <pre>
     *  name -> (nothing)         ===> name -> new_stub -> new_value
     *  name -> stub -> old_value ===> name -> stub -> new_value
     * </pre>
     */
    public void extend (String variableName, Value value)
    {
        ValueStub stub = null;
        if ((stub = (ValueStub)bindings.get(variableName)) == null)
        {
            bindings.put (variableName, new ValueStub (value));
        }
        else
        {
            stub.setValue (value);
        }
    }

    /**
     * Changes the binding of a name to value destructively. A new stub is
     * always created no matter whether the variable is previously bound.
     * <pre>
     *  name -> (nothing)         ===> name -> new_stub -> new_value
     *  name -> stub -> old_value ===> name -> new_stub -> new_value
     * </pre>
     */
    public void extendDestructive (String variableName, Value value)
    {
        bindings.put (variableName, new ValueStub (value));
```



```
    }

    public void setGraphics (Graphics2D g)
    {
        this.g = g;
    }

    public Graphics2D getGraphics ()
    {
        return this.g;
    }

    public void setTransform (AffineTransform m)
    {
        this.m = m;
    }

    public AffineTransform getTransform()
    {
        return this.m;
    }

    /**
     * Gives a meaningful representation of the environment, for debugging
     * purpose
     */
    public String toString ()
    {
        StringBuffer s = new StringBuffer ("");
        s.append ("Break flag: " + breakFlag + "\n");
        s.append ("Continue flag: " + continueFlag + "\n");
        s.append ("Return flag: " + returnFlag + "\n");
        s.append ("Return value: " + returnValue + "\n");
        for (Iterator iter = bindings.keySet().iterator();
            iter.hasNext();)
        {
            String name = (String)iter.next();
            s.append (name + ":" + bindings.get(name) + "\n");
        }
        return s.toString();
    }

    private class ValueStub
    {
        private Value value = null;

        public ValueStub()
        {
        }

        public ValueStub(Value value)
        {
            this.value = value;
        }

        public void setValue (Value value)
        {
            this.value = value;
        }

        public Value getValue ()
        {
            return this.value;
        }
```

```
        public String toString ()
        {
            return value.toString();
        }

    } // class ValueStub

} // class Environment
```

```
package sigl.core;

public abstract class Expr
extends Node
{
    public abstract Value evaluate (Environment e);

} // class Expr
```

```
package sigl.core;

public class ExprStmt
extends Stmt
{
    private Expr expr = null;

    public ExprStmt ()
    {
    }

    public ExprStmt (Expr expr)
    {
        this.expr = expr;
    }

    public void execute (Environment env)
    {
        expr.evaluate (env);
    }

} // class ExprStmt
```

```
package sigl.core;

import java.util.*;

/**
 * Represents the "while" statement in SIGL language.
 *
 * @version $Revision$
 */
public class For
extends Stmt
{
    private Vector initialization = null;
    private Expr condition = null;
    private Vector update = null;
    private Stmt body = null;

    public For ()
    {
    }

    public For (Vector initialization,
                Expr condition,
                Vector update,
                Stmt body)
    {
        this.initialization = initialization;
        this.condition = condition;
        this.update = update;
        this.body = body;
    }

    public void setCondition (Expr condition)
    {
        this.condition = condition;
    }

    public Expr getCondition ()
    {
        return this.condition;
    }

    public void setInitialization (Vector initialization)
    {
        this.initialization = initialization;
    }

    public Vector getInitialization ()
    {
        return this.initialization;
    }

    public void setUpdate (Vector update)
    {
        this.update = update;
    }

    public Vector getUpdate ()
    {
        return update;
    }

    public void setBody (Stmt body)
    {
```



```
        this.body = body;
    }

    public Stmt getBody ()
    {
        return this.body;
    }

    public void execute (Environment env)
    {
        // run the initialization
        for (ListIterator iter = initialization.listIterator();
            iter.hasNext();)
        {
            Expr expr = (Expr)iter.next();
            expr.evaluate(env);
        }
        while (true)
        {
            if (condition != null)
            {
                // evaluate the condition
                Value cond = condition.evaluate (env);
                // type check
                if ((cond == null) || (!(cond instanceof BoolValue)))
                    throw new NotWellTypedError ("Condition to \"for\" must be a boolean expre
ssion", condition);
                // branch according to the return boolean value
                if (!((BoolValue)cond).getValue()) return;
            }
            body.execute (env);
            // run the update
            for (ListIterator iter = update.listIterator();
                iter.hasNext();)
            {
                Expr expr = (Expr)iter.next();
                expr.evaluate(env);
            }
            // check for all possible flags to catch
            if (env.getContinueFlag()) env.setContinueFlag (false);
            if (env.getBreakFlag())
            {
                env.setBreakFlag (false);
                return;
            }
            if (env.getReturnFlag()) return;
        }
    }

} // class For
```

```
package sigl.core;

import java.util.*;

public class Function
extends Stmt
{
    private String functionName = null;
    private Vector formals = new Vector();
    private Stmt body = null;

    public Function ()
    {
    }

    public Function (String functionName, Vector formals, Stmt body)
    {
        this.functionName = functionName;
        this.formals = formals;
        this.body = body;
    }

    public void setFunctionName (String functionName)
    {
        this.functionName = functionName;
    }

    public String getFunctionName ()
    {
        return this.functionName;
    }

    public void setFormals (Vector formals)
    {
        this.formals = formals;
    }

    public Vector getFormals ()
    {
        return this.formals;
    }

    public void setBody (Stmt body)
    {
        this.body = body;
    }

    public Stmt getBody ()
    {
        return this.body;
    }

    public void execute (Environment env)
    {
        Environment functionEnvironment = (Environment)env.duplicate();
        FunctionValue funcValue = new FunctionValue (this, functionEnvironment);
        functionEnvironment.extendDestructive (functionName, funcValue);
        env.extend (functionName, funcValue);
    }

    public Value apply (Vector arguments, Environment _env)
    {
        if (arguments.size() != formals.size())
        {
```



```
            throw new NotWellTypedError ("Function \"" + functionName +
                                         "\" requires " + formals.size() +
                                         " argument(s) but found " + arguments.size(),
                                         this);
        }
        Environment env = _env.duplicate();
        for (int i = 0;i < formals.size();++i)
        {
            String formalName = (String)formals.elementAt (i);
            Value value = (Value)arguments.elementAt (i);
            env.extendDestructive (formalName, value);
        }
        body.execute (env);
        if (env.getBreakFlag())
            throw new NotWellTypedError ("Break without a loop", env.getBreakStmt());
        if (env.getContinueFlag())
            throw new NotWellTypedError ("Continue without a loop", env.getContinueStm
t());
        if (env.getReturnFlag()) return env.getReturnValue();
        return null;
    }

} // class Function
```

```
package sigl.core;

import java.util.*;

public class FunctionCall
extends Expr
{
    private String funcName = null;
    private Vector arguments = new Vector();

    public FunctionCall ()
    {
    }

    public FunctionCall (String funcName)
    {
        this.funcName = null;
    }

    public void setFunctionName (String funcName)
    {
        this.funcName = funcName;
    }

    public String getFunctionName ()
    {
        return this.funcName;
    }

    public void setArguments (Vector arguments)
    {
        this.arguments = arguments;
    }

    public Vector getArguments ()
    {
        return this.arguments;
    }

    public Value evaluate (Environment env)
    {
        Value value = env.access (funcName);
        if (value == null)
            throw new NotWellTypedError ("Function \"" + funcName + "\" not found", thi
s);
        if (!(value instanceof FunctionValue))
            throw new NotWellTypedError ("\"" + funcName + "\" is not a function", this)
;
        FunctionValue funcValue = (FunctionValue)value;
        Vector args = new Vector();
        for (ListIterator iter = arguments.listIterator();
            iter.hasNext();)
        {
            if (!Environment.passByNeed)
                args.add (((Expr)iter.next()).evaluate (env));
            else
                args.add (new ThunkValue((Expr)iter.next(), env));
        }
        try
        {
            return funcValue.apply (args, env);
        }
        catch (NotWellTypedError e)
        {
```



```
            // this catch is for built-in function
            // where the function body is not available
            // the error location will be pointed to
            // the function application itself
            if (e.getNode() == null) e.setNode(this);
            throw e;
        }
    }

} // class FunctionCall
```

```
package sigl.core;

import java.util.*;

public class FunctionValue
implements Value
{
    private Function func = null;
    private Environment env = null;

    public FunctionValue ()
    {
    }

    public FunctionValue (Function func, Environment env)
    {
        this.func = func;
        this.env = env;
    }

    public void setEnvironment (Environment env)
    {
        this.env = env;
    }

    public Environment getEnvironment ()
    {
        return env;
    }

    public Function getFunction ()
    {
        return func;
    }

    public void setFunction (Function func)
    {
        this.func = func;
    }

    public Value apply (Vector arguments, Environment _env)
    {
        if (!Environment.dynamicScoping)
            return func.apply (arguments, env);
        else
            return func.apply (arguments, _env);
    }

} // class FunctionValue
```

```
package sigl.core;

/**
 * Represents the "if" statement in SIGL language. Each if statement consists
 * of a conditional expression, an if-part, and an else-part. If there is no
 * else-part, it is set to null
 *
 * @version $Revision$
 */
public class If
extends Stmt
{
    private Expr condition = null;
    private Stmt ifPart = null;
    private Stmt elsePart = null;

    public If ()
    {
    }

    public If (Expr condition, Stmt ifPart)
    {
        this (condition, ifPart, null);
    }

    public If (Expr condition, Stmt ifPart, Stmt elsePart)
    {
        this.condition = condition;
        this.ifPart = ifPart;
        this.elsePart = elsePart;
    }

    public void setCondition (Expr condition)
    {
        this.condition = condition;
    }

    public Expr getCondition ()
    {
        return this.condition;
    }

    public void setIfPart (Stmt ifPart)
    {
        this.ifPart = ifPart;
    }

    public Stmt getIfPart ()
    {
        return this.ifPart;
    }

    public void setElsePart (Stmt elsePart)
    {
        this.elsePart = elsePart;
    }

    public Stmt getElsePart ()
    {
        return this.elsePart;
    }

    public void execute (Environment env)
    {
```



```
        // evaluate the condition
        Value cond = condition.evaluate (env);
        // type check
        if ((cond == null) || (!(cond instanceof BoolValue)))
            throw new NotWellTypedError ("Condition to \"if\" must be a boolean expression", co
ndition);
        // branch according to the return boolean value
        if (((BoolValue)cond).getValue()) ifPart.execute(env);
        else if (elsePart != null) elsePart.execute(env);
    }

} // class If
```

```
package sigl.core;

public class IntConstant
extends Expr
{
    private int value = 0;

    public IntConstant ()
    {
    }

    public IntConstant (int value)
    {
        this.value = value;
    }

    public int getValue ()
    {
        return value;
    }

    public void setValue (int value)
    {
        this.value = value;
    }

    public Value evaluate (Environment e)
    {
        return new IntValue (value);
    }

} // class IntConstant
```

```
package sigl.core;

public class IntValue
implements NumericValue
{
    private int value = 0;

    public IntValue ()
    {
    }

    public IntValue (int value)
    {
        this.value = value;
    }

    public int getValue ()
    {
        return value;
    }

    public void setValue (int value)
    {
        this.value = value;
    }

    public int hashCode ()
    {
        return value;
    }

    public boolean equals (Object o)
    {
        if (!(o instanceof IntValue)) return false;
        return (value == ((IntValue)o).value);
    }

    public String toString ()
    {
        return "" + value;
    }

} // class IntValue
```

```
package sigl.core;

public class LogicalOperation
extends BinaryOperation
{
    public LogicalOperation ()
    {
    }

    public LogicalOperation (String op, Expr lhs, Expr rhs)
    {
        super (op, lhs, rhs);
    }

    public Value evaluate (Environment env)
    {
        Value l = lhs.evaluate(env);
        if (l == null || !(l instanceof BoolValue))
            throw new NotWellTypedError ("Left hand side is not boolean", this);
        if (op.equals ("||"))
        {
            if (((BoolValue)l).getValue()) return l;
            Value r = rhs.evaluate (env);
            if (r == null || !(r instanceof BoolValue))
                throw new NotWellTypedError ("Right hand side is not boolean", this);
            return r;
        }
        else if (op.equals ("&&"))
        {
            if (!((BoolValue)l).getValue()) return l;
            Value r = rhs.evaluate (env);
            if (r == null || !(r instanceof BoolValue))
                throw new NotWellTypedError ("Right hand side is not boolean", this);
            return r;
        }
        else throw new NotWellTypedError ("Unknown operator", this);
    }

} // class LogicalOperation
```

```
package sigl.core;

public abstract class Node
{
    private int line = -1;
    private int column = -1;

    public void setLine (int line)
    {
        this.line = line;
    }

    public int getLine ()
    {
        return line;
    }

    public void setColumn (int col)
    {
        this.column = col;
    }

    public int getColumn ()
    {
        return column;
    }

} // class Node
```

```
package sigl.core;

public class NotWellTypedError
extends Error
{
    private Node errorNode = null;

    public NotWellTypedError ()
    {
        super ();
    }

    public NotWellTypedError (String message)
    {
        super (message);
    }

    public NotWellTypedError (Node cause)
    {
        this.errorNode = cause;
    }

    public NotWellTypedError (String message, Node cause)
    {
        super (message);
        this.errorNode = cause;
    }

    public Node getNode ()
    {
        return errorNode;
    }

    public void setNode (Node node)
    {
        this.errorNode = node;
    }

} // class NotWellTypedError
```

```
package sigl.core;

public interface NumericValue
extends Value
{
} // interface NumericValue
```

```
package sigl.core;

public class RealConstant
extends Expr
{
    private double value = 0;

    public RealConstant ()
    {
    }

    public RealConstant (double value)
    {
        this.value = value;
    }

    public double getValue ()
    {
        return value;
    }

    public void setValue (double value)
    {
        this.value = value;
    }

    public Value evaluate (Environment e)
    {
        return new RealValue (value);
    }

} // class RealConstant
```

```
package sigl.core;

public class RealValue
implements NumericValue
{
    private double value = 0;

    public RealValue ()
    {
    }

    public RealValue (double value)
    {
        this.value = value;
    }

    public double getValue ()
    {
        return value;
    }

    public void setValue (double value)
    {
        this.value = value;
    }

    public String toString ()
    {
        return "" + value;
    }

} // class RealValue
```

```
package sigl.core;

public class RelationalOperation
extends BinaryOperation
{
    public RelationalOperation ()
    {
    }

    public RelationalOperation (String op, Expr lhs, Expr rhs)
    {
        super (op, lhs, rhs);
    }

    public Value evaluate (Environment env)
    {
        Value l = lhs.evaluate(env);
        Value r = rhs.evaluate(env);
        if (l == null) throw new NotWellTypedError ("Invalid value for left hand side", thi
s);
        if (r == null) throw new NotWellTypedError ("Invalid value for right hand side", th
is);

        if (l instanceof BoolValue)
        {
            if (op.equals ("==") || op.equals ("!="))
            {
                if (!(r instanceof BoolValue))
                    throw new NotWellTypedError("Types mismatched between left and right han
d side", this);
                return new BoolValue (
                    (op.equals("=="))
                    ?(
                        ((BoolValue)l).getValue() ==
                        ((BoolValue)r).getValue()
                    )
                    :(
                        ((BoolValue)l).getValue() !=
                        ((BoolValue)r).getValue()
                    )
                );
            }
            else
            {
                throw new NotWellTypedError("Left hand side is boolean, numeric type expected",
 this);
            }
        }

        if (!((l instanceof IntValue) || (l instanceof RealValue)))
            throw new NotWellTypedError("Left hand side is not numeric", this);
        if (!((r instanceof IntValue) || (r instanceof RealValue)))
            throw new NotWellTypedError("Right hand side is not numeric", this);

        double lvalue = 0;
        double rvalue = 0;
        if (l instanceof IntValue) lvalue = ((IntValue)l).getValue();
        else lvalue = ((RealValue)l).getValue();
        if (r instanceof IntValue) rvalue = ((IntValue)r).getValue();
        else rvalue = ((RealValue)r).getValue();

        if (op.equals ("==")) return new BoolValue (lvalue == rvalue);
        if (op.equals ("!=")) return new BoolValue (lvalue != rvalue);
        if (op.equals ("<")) return new BoolValue (lvalue < rvalue);
```



```
        if (op.equals ("<=")) return new BoolValue (lvalue <= rvalue);
        if (op.equals (">")) return new BoolValue (lvalue > rvalue);
        if (op.equals (">=")) return new BoolValue (lvalue >= rvalue);

        throw new NotWellTypedError ("Unknown operator", this);
    }

} // class RelationalOperation
```

```
package sigl.core;

public class Return
extends Stmt
{
    private Expr expr = null;

    public Return()
    {
    }

    public Return (Expr expr)
    {
        this.expr = expr;
    }

    public void setReturnExpression (Expr expr)
    {
        this.expr = expr;
    }

    public Expr getReturnExpression ()
    {
        return this.expr;
    }

    public void execute (Environment env)
    {
        env.setReturnFlag (true);
        if (expr != null) env.setReturnValue (expr.evaluate (env));
        else env.setReturnValue (null);
    }

} // class Return
```

```
package sigl.core;

public abstract class Stmt
extends Node
{
    public abstract void execute (Environment e);

} // class Stmt
```

```
package sigl.core;

/**
 * Represents a pair of expression and the environment in which the expression
 * is defined. This allows late computation in pass-by-need.
 */
public class ThunkValue
implements Value
{
    private Expr expr = null;
    private Environment env = null;

    /**
     * Default empty constructor
     */
    public ThunkValue ()
    {
    }

    /**
     * Creates a thunk given the expression and the environment
     * @param expr the expression
     * @param env the environment
     */
    public ThunkValue (Expr expr, Environment env)
    {
        this.expr = expr;
        this.env = env;
    }

    public Expr getExpr ()
    {
        return this.expr;
    }

    public void setExpr (Expr expr)
    {
        this.expr = expr;
    }

    public Environment getEnvironment ()
    {
        return env;
    }

    public void setEnvironment (Environment env)
    {
        this.env = env;
    }

} // ThunkValue
```

```
package sigl.core;

import java.awt.geom.AffineTransform;

public class Transform
extends Stmt
{
    private Expr transformation = null;
    private Stmt body = null;

    public Transform ()
    {
    }

    public Transform (Expr transformation, Stmt body)
    {
        this.transformation = transformation;
        this.body = body;
    }

    public void execute (Environment env)
    {
        // backup the transformation matrix
        AffineTransform m = (AffineTransform)env.getTransform().clone();
        // perform the transformation
        transformation.evaluate (env);
        // and execute the body
        body.execute (env);
        // now we've finish, restore the old transformation matrix
        env.getTransform().setTransform(m);
    }

} // class Transform
```

```
package sigl.core;

public class UnaryOperation
extends Expr
{
    private String op = null;
    private Expr expr = null;

    public UnaryOperation ()
    {
    }

    public UnaryOperation (String op, Expr expr)
    {
        this.op = op;
        this.expr = expr;
    }

    public void setOperator (String op)
    {
        this.op = op;
    }

    public String getOperator ()
    {
        return this.op;
    }

    public void setExpression (Expr expr)
    {
        this.expr = expr;
    }

    public Expr getExpression ()
    {
        return expr;
    }

    public Value evaluate (Environment env)
    {
        Value e = expr.evaluate (env);
        if (op.equals ("!"))
        {
            if (e == null || !(e instanceof BoolValue))
                throw new NotWellTypedError ("This operator is only applicable to boolean value"
, this);
            return new BoolValue (!((BoolValue)e).getValue());
        }
        else if (op.equals ("-"))
        {
            if (e == null)
                throw new NotWellTypedError ("Invalid value", this);
            if (e instanceof IntValue)
                return new IntValue (-((IntValue)e).getValue());
            else if (e instanceof RealValue)
                return new RealValue (-((RealValue)e).getValue());
            else
                throw new NotWellTypedError ("This operator is only applicable to numeric value"
, this);
        }
        else throw new NotWellTypedError ("Unknown operator", this);
    }

} // class UnaryOperation
```

```
package sigl.core;

public interface Value
{
} // interface Value
```

```
package sigl.core;

import java.util.*;

public class Variable
extends Expr
{
    private String name = null;
    private Vector indices = new Vector();

    public Variable ()
    {
    }

    public Variable (String name)
    {
        this.name = name;
    }

    public void setName (String name)
    {
        this.name = name;
    }

    public String getName ()
    {
        return this.name;
    }

    public void setIndices (Vector indices)
    {
        this.indices = indices;
    }

    public Vector getIndices ()
    {
        return this.indices;
    }

    public Value evaluate (Environment env)
    {
        Value v = env.access (name);
        StringBuffer s = new StringBuffer();
        for (ListIterator iter = indices.listIterator();
            iter.hasNext();)
        {
            if (!(v instanceof ArrayValue))
            {
                throw new NotWellTypedError ("Trying to index a non-array value " + name +
 s.toString(), this);
            }
            Expr expr = (Expr)iter.next();
            Value index = expr.evaluate (env);
            v = ((ArrayValue)v).get (index);
            s.append("[" + index + "]");
        }
        if (v == null)
        {
            if (indices.isEmpty())
                throw new NotWellTypedError ("Unbound variable \"" + name + "\"", this
);
            else
                throw new NotWellTypedError ("Array index out of bound " + name + s.toS
tring(), this);
```



```
        }
        return v;
    }

} // class Variable
```

```
package sigl.core;

/**
 * Represents the "while" statement in SIGL language.
 *
 * @version $Revision$
 */
public class While
extends Stmt
{
    private Expr condition = null;
    private Stmt body = null;

    public While ()
    {
    }

    public While (Expr condition, Stmt body)
    {
        this.condition = condition;
        this.body = body;
    }

    public void setCondition (Expr condition)
    {
        this.condition = condition;
    }

    public Expr getCondition ()
    {
        return this.condition;
    }

    public void setBody (Stmt body)
    {
        this.body = body;
    }

    public Stmt getBody ()
    {
        return this.body;
    }

    public void execute (Environment env)
    {
        while (true)
        {
            // evaluate the condition
            Value cond = condition.evaluate (env);
            // type check
            if ((cond == null) || (!(cond instanceof BoolValue)))
                throw new NotWellTypedError ("Condition to \"while\" must be a boolean expressio
n", condition);
            // branch according to the return boolean value
            if (!((BoolValue)cond).getValue()) return;
            body.execute (env);
            if (env.getContinueFlag()) env.setContinueFlag (false);
            if (env.getBreakFlag())
            {
                env.setBreakFlag (false);
                return;
            }
            if (env.getReturnFlag()) return;
        }
```



```
    }

} // class While
```

```
package sigl;

import java.io.*;
import java.util.*;
import java.net.*;

import java.awt.*;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;

import sigl.core.*;
import sigl.parser.*;
import sigl.util.*;

/**
 * The main interpretor for SIGL language.
 */
public class Evaluator
{
    public static void main (String args[])
    throws Exception
    {
        // get the possible options
        int imageWidth = Integer.parseInt (CommandLineUtil.getOption ("--width",
args, "300"));
        int imageHeight = Integer.parseInt (CommandLineUtil.getOption ("--height"
, args, "300"));
        Environment.dynamicScoping = CommandLineUtil.getFlag ("--dynamic", args);
        Environment.passByNeed = CommandLineUtil.getFlag ("--pbn", args);

        args = CommandLineUtil.removeBlanks (args);

        // the source file
        String filename = args[0];
        // the output image file, default to be output.jpeg
        String outputFilename = "output.jpeg";
        if (args.length >= 2) outputFilename = args[1];

        // lexer
        SIGLLexer lexer = new SIGLLexer(
            new BufferedReader (
                new FileReader (filename)
            )
        );

        // parser
        SIGLParser parser = new SIGLParser (lexer);
        parser.setASTNodeClass ("sigl.parser.CommonASTWithLine");
        parser.program();
        antlr.CommonAST ast = (antlr.CommonAST) parser.getAST();

        // walk the parse tree and convert it to our format
        SIGLWalker walker = new SIGLWalker();
        Stmt s = walker.stmt(ast);

        // create an empty environment
        Environment env = new Environment();
        // we use java reflection to automatically add all functions
        // defined in sigl.builtin package into the default environment
        // as built-in in functions
        Class[] classes = getClasses ("sigl.builtin");
        for (int i = 0;i < classes.length;++i)
        {
            FunctionValue funcValue = (FunctionValue)classes[i].newInstance();
```



```
            String functionName = funcValue.getFunction ().getFunctionName();
//          System.out.println ("Loading built-in function \"" +
//              functionName + "\" (" + classes[i].getName() + ")");
            env.extend (functionName, funcValue);
        }

        // set up the drawing environment
        BufferedImage image = new BufferedImage (imageWidth, imageHeight, Buffer
edImage.TYPE_INT_RGB);
        Graphics2D g = (Graphics2D)image.getGraphics();
        g.setColor (Color.WHITE);
        env.setGraphics (g);
        // now we can start execution
        try
        {
            s.execute (env);
            if (env.getBreakFlag())
                throw new NotWellTypedError ("Break without a loop", env.getBreakStmt
());
            if (env.getContinueFlag())
                throw new NotWellTypedError ("Continue without a loop", env.getContinu
eStmt());
        }
        catch (NotWellTypedError e)
        {
            System.err.print (e.getMessage());
            if (e.getNode() != null)
                System.err.print (" at " +
                e.getNode().getLine() + ":" +
                e.getNode().getColumn());
            System.err.println ();
            System.err.flush ();
        }
        // output the final image
        ImageIO.write (image, "JPEG", new File (outputFilename));
    }

    private static Class[] getClasses(String packageName)
    throws ClassNotFoundException
    {
        ArrayList classes = new ArrayList();
        // get a file object for the package
        File directory = null;
        try
        {
            ClassLoader cld = Thread.currentThread().getContextClassLoader();
            if (cld == null)
            {
                throw new ClassNotFoundException("Can't get class loader.");
            }
            String path = packageName.replace('.', '/');
            URL resource = cld.getResource(path);
            if (resource == null)
            {
                throw new ClassNotFoundException("No resource for " + path);
            }
            if (resource.getProtocol().equals ("file"))
                directory = new File(URLDecoder.decode(resource.getFile(), "UTF
-8"));
            else
                directory = new File(resource.getFile());
        }
        catch (UnsupportedEncodingException x)
        {
```

```
            throw new ClassNotFoundException(packageName + "(" + directory
                    + ") does not appear to be a valid package");
        }
        if (directory.exists())
        {
            // Get the list of the files contained in the package
            File[] files = directory.listFiles();
            for (int i = 0; i < files.length; i++)
            {
                String filename = files[i].getName();
                // we are only interested in .class files
                if (filename.endsWith(".class") && !filename.equals ("BuiltinFunction.cl
ass"))
                {
                    // removes the .class extension
                    classes.add(Class.forName(packageName + '.'
                            + filename.substring(0, filename.length() - 6)));
                }
                if (files[i].isDirectory())
                {
                    Class[] children = getClasses(packageName + "." + filename);
                    for (int j = 0;j < children.length;++j)
                        classes.add(children[j]);
                }
            }
        }
        else
        {
            throw new ClassNotFoundException(packageName
                    + " does not appear to be a valid package");
        }
        Class[] classesA = new Class[classes.size()];
        classes.toArray(classesA);
        return classesA;
    }


} // class Evaluator
```

```
package sigl.parser;

import antlr.*;
import antlr.collections.*;

public class CommonASTWithLine extends CommonAST
{
    private int col = -1;
    private int line = -1;

    public void initialize(Token tok)
    {
        super.initialize(tok);
        line = tok.getLine();
        col = tok.getColumn();
    }
    public void initialize (AST ast)
    {
        super.initialize(ast);
        if (ast instanceof CommonASTWithLine)
        {
            col = ((CommonASTWithLine)ast).getColumn();
            line = ((CommonASTWithLine)ast).getLine();
        }
    }
    public int getLine()
    {
        return line;
    }
    public int getColumn()
    {
        return col;
    }
}
```

```
package sigl.util;

import java.util.*;

public class CommandLineUtil
{
    public static String getOption (String optionFlag,
                                    String [] options)
    throws InvalidOptionError
    {
        return getOption (optionFlag, options, null, null);
    }

    public static String getOption (String optionFlag,
                                    String [] options,
                                    String defaultValue)
    throws InvalidOptionError
    {
        return getOption (optionFlag, options, defaultValue, null);
    }

    /**
     * Gets the value of an option from option list.
     * @param optionFlag the flag of the option being looked for
     * @param options the option list
     * @param defaultValue the default value to be returned if the option
     *          is missing (put null to omit);
     * @param valueConversion the hash to convert from the input to internal
     *          representation. This value can also be used to put constraints
     *          on possible values for the option: when this parameter is
     *          supplied (not null), the method checks if the input value
     *          is one of the keys of valueConversion value and output the
     *          value of that key accordingly
     * @return defaultValue if the option is not found,<br>
     *          the input value if the option is found and the valueConversion
     *          is not provided,<br>
     *          and the converted value if the option is found and the
     *          valueConversion is provided
     * @throws InvalidOptionError if the option is found but the input value is
     *          not found or an input value is not one of the key of
     *          valueConversion hash if valueConversion hash is provided
     */
    public static String getOption (String optionFlag,
                                    String [] options,
                                    String defaultValue,
                                    Properties valueConversion)
    throws InvalidOptionError
    {
        String value = null;
        for (int i = 0;i < options.length;++i)
        {
            if (options[i].equals (optionFlag))
            {
                // the first type of option
                // the option value come as a separate argument
                // right after the option flag argument
                options[i] = "";
                if (i == options.length - 1)
                {
                    throw new InvalidOptionError
                        ("Missing value for option \"" + optionFlag + "\"");
                }
                value = options[i + 1];
                options[i + 1] = "";
```



```
            }
            else
            {
                // the second type of option
                // the option value is put in the same argument
                // as the option flag, separated by a =
                if (options[i].startsWith (optionFlag + "="))
                {
                    value = options[i].substring (optionFlag.length () + 1);
                    options[i] = "";
                }
            }

            if (value != null)
            {
                if (valueConversion != null)
                {
                    String convertedValue = null;
                    if ((convertedValue = valueConversion.getProperty (value))
                        != null)
                    {
                        return convertedValue;
                    }
                    else
                    {
                        // construct a string consists of all possible
                        // values for the option
                        // for error outputing purpose
                        StringBuffer possibleValue = new StringBuffer();
                        for (Enumeration e = valueConversion.keys ();
                            e.hasMoreElements();)
                        {
                            possibleValue.append (",");
                            possibleValue.append ((String)e.nextElement ());
                        }
                        possibleValue.delete (0, 1);

                        // throw error
                        throw new InvalidOptionError ("Invalid option value. " +
                            "Possible values for \"" + optionFlag + "\" are [" +
                            possibleValue.toString () + "]");
                    }
                }
                else
                {
                    return value;
                }
            }
        }

        return defaultValue;
    }

    public static boolean getFlag (String optionFlag,
                                   String [] options)
    {
        for (int i = 0;i < options.length;++i)
        {
            if (options[i].equals (optionFlag))
            {
                options[i] = "";
                return true;
            }
        }
```

```
        return false;
    }

    public static String [] removeBlanks (String [] options)
    {
        int count = 0;
        for (int i = 0;i < options.length;++i)
        {
            if (options[i].length () > 0) ++count;
        }
        String [] result = new String[count];
        for (int i = 0, j = 0;i < options.length;++i)
        {
            if (options[i].length () > 0)
            {
                result[j++] = options[i];
            }
        }
        return result;
    }

    public static Properties propertiesFromStringArray (String [] values)
    {
        Properties p = new Properties ();
        for (int i = 0;i < values.length;i += 2)
        {
            if (i + 1 >= values.length) break;
            p.setProperty (values[i], values[i + 1]);
        }
        return p;
    }

} // class CommandLineUtil
```

```
package sigl.util;

public class InvalidOptionError extends Error
{
    public InvalidOptionError ()
    {
        super ();
    }

    public InvalidOptionError (String message)
    {
        super (message);
    }

    public InvalidOptionError (String message, Throwable cause)
    {
        super (message, cause);
    }

    public InvalidOptionError (Throwable cause)
    {
        super (cause);
    }

} // class InvalidOptionError
```

```
header
{
package sigl.parser;
}

{
import java.io.*;
}

class SIGLLexer extends Lexer;

options
{
    k = 2;
    charVocabulary = '\3'..'\377';
    exportVocab = SIGL;
    testLiterals = false;
}

tokens
{
    INT_CONSTANT;
    REAL_CONSTANT;
    NEGATE;
}

{
    public static void main (String args[])
    throws Exception
    {
        String filename = args[0];
        BufferedReader input = new BufferedReader (
            new FileReader (filename));
        SIGLLexer lexer = new SIGLLexer (input);
        while (true)
        {
            Token siglToken = (Token)lexer.nextToken ();
            if (siglToken.getType() == Token.EOF_TYPE) break;
            System.out.println (siglToken);
            System.out.flush ();
        }
        input.close ();
    }
}

/* Operators: */

ASSIGN          : '=' ;
COLON           : ':' ;
COMMA           : ',' ;
QUESTION        : '?' ;
SEMI            : ';' ;

LPAREN          : '(' ;
RPAREN          : ')' ;
LBRACKET        : '[' ;
RBRACKET        : ']' ;
LCURLY          : '{' ;
RCURLY          : '}' ;

EQUAL           : "==" ;
NOT_EQUAL       : "!=" ;
LTE             : "<=" ;
LT              : "<" ;
```



```
GTE             : ">=" ;
GT              : ">" ;

DIV             : '/' ;
DIV_ASSIGN      : "/=" ;
PLUS            : '+' ;
PLUS_ASSIGN     : "+=" ;
MINUS           : '-' ;
MINUS_ASSIGN    : "-=" ;
MUL             : '*' ;
MUL_ASSIGN      : "*=" ;
MOD             : '%' ;
MOD_ASSIGN      : "%=" ;

LAND            : "&&" ;
LNOT            : '!' ;
LOR             : "||" ;

WHITESPACE
        :       ( ( '\003'..'\010' | '\t' | '\013' | '\f' | '\016'.. '\037' | '\
177'..'\377' | ' ' )
                    | "\r\n"                { newline(); }
                    | ( '\n' | '\r' )       { newline(); }
                )                           { $setType(Token.SKIP); }
        ;


COMMENT
        :       "/*"
                (
                    options
                    {
                        generateAmbigWarnings = false;
                    }
                    :
                    { LA(2) != '/' }? '*'
                    | "\r\n"                { newline(); }
                    | ( '\r' | '\n' )       { newline(); }
                    | ~( '*'| '\r' | '\n' )
                )*
                "*/"                        { $setType(Token.SKIP); }
        ;

CPPCOMMENT
        :
                "//" ( ~('\n') )*           { $setType(Token.SKIP); }
        ;

/* Numeric Constants: */

protected
Digit
        :       '0'..'9'
        ;

protected
Exponent
        :       ( 'e' | 'E' ) ( '+' | '-' )? ( Digit )+
        ;

NUMBER
        :       ( ( Digit )+ ( '.' | 'e' | 'E' ) )=> ( Digit )+
                ( '.' ( Digit )* ( Exponent )?
                | Exponent
```

```
                )                               { $setType(REAL_CONSTANT); }
        |       '.'
                ( ( Digit )+ ( Exponent )?
                                                { $setType(REAL_CONSTANT); }
                )

        |       (Digit)+     { $setType(INT_CONSTANT); }
        ;


ID
        options
                {
                    testLiterals = true;
                }
        :       ( 'a'..'z' | 'A'..'Z' | '_' )
                ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' )*
        ;


{
import java.io.*;
import antlr.debug.misc.*;

import sigl.parser.*;
}

class SIGLParser extends Parser;
options
{
    k = 2;
    buildAST = true;
    exportVocab = SIGL;
}

tokens
{
    PROGRAM;
    STMT_LIST;
    EXPR_LIST;
    FORMAL_LIST;
    FUNC_CALL;
    TRANSFORM;
    BREAK="break";
    CONTINUE="continue";
    RETURN="return";
}

{
    public static void main (String args[])
    throws Exception
    {
        String filename = args[0];
        SIGLLexer lexer = new SIGLLexer(
            new BufferedReader (
                new FileReader (filename)
            )
        );
        SIGLParser parser = new SIGLParser (lexer);
        parser.program();
        antlr.CommonAST ast = (antlr.CommonAST) parser.getAST();

        ASTFrame frame = new ASTFrame("AST", ast);
        frame.setVisible(true);
```



```
    }
}

program
    : (stmt | funcdecl)*
        { #program = #([PROGRAM,"PROGRAM"], program); }
    ;

funcdecl
    : "function"^ ID LPAREN! formal_list RPAREN! LCURLY! stmt_list RCURLY!
    ;

formal_list
    : ID (COMMA! ID)*
        { #formal_list = #([FORMAL_LIST,"FORMAL_LIST"], formal_list); }
    | { #formal_list = #([FORMAL_LIST,"FORMAL_LIST"]); }
    ;

stmt_list
    : (stmt)*
        { #stmt_list = #([STMT_LIST,"STMT_LIST"], stmt_list); }
    ;

stmt
    : if_stmt
    | for_stmt
    | while_stmt
    | break_stmt
    | continue_stmt
    | return_stmt
    | expr_stmt
    | LCURLY! stmt_list RCURLY!
    | transformation_stmt
    | empty_stmt
    ;

if_stmt
    : "if"^ LPAREN! expr RPAREN! stmt
        (options {greedy = true;}: "else"! stmt )?
    ;

for_stmt
    : "for"^ LPAREN! expr_list SEMI! (expr)? SEMI! expr_list RPAREN! stmt
    ;

while_stmt
    : "while"^ LPAREN! expr RPAREN! stmt
    ;

break_stmt
    : BREAK SEMI!
    ;

continue_stmt
    : CONTINUE SEMI!
    ;

return_stmt
    : RETURN^ (expr)? SEMI!
    ;

expr_stmt
    : expr SEMI!
    ;
```

```
transformation_stmt
    : COLON! func_call COLON! stmt
        { #transformation_stmt = #([TRANSFORM,"TRANSFORM"], transformation_stmt)
; }
    ;

empty_stmt
    : SEMI!
    ;

expr
    : assignment_expr
    ;

assignment_expr
    : conditional_expr
        (
            (
                ASSIGN^
                | PLUS_ASSIGN^
                | MINUS_ASSIGN^
                | MUL_ASSIGN^
                | DIV_ASSIGN^
                | MOD_ASSIGN^
            )
            assignment_expr
        )?
    ;

conditional_expr
    : conditional_term (LOR^ conditional_term)*
    ;

conditional_term
    : equality_expr (LAND^ equality_expr)*
    ;

equality_expr
    : relational_expr ((EQUAL^ | NOT_EQUAL^) relational_expr)*
    ;

relational_expr
    : arithmetic_expr ((LT^ | LTE^ | GT^ | GTE^) arithmetic_expr)*
    ;

arithmetic_expr
    : term ((PLUS^ | MINUS^) term)*
    ;

term
    : unary ((MUL^ | DIV^ | MOD^) unary)*
    ;

unary
    : MINUS^ unary
    | LNOT^ unary
    | factor
    ;

factor
    : LPAREN! expr RPAREN!
    | l_value
    | func_call
```



```
    | INT_CONSTANT
    | REAL_CONSTANT
    | "true"
    | "false"
    ;

l_value
    : ID^ (LBRACKET! expr RBRACKET!)*
    ;

func_call
    : ID LPAREN! expr_list RPAREN!
        { #func_call = #([FUNC_CALL,"FUNC_CALL"], func_call); }
    ;

expr_list
    : expr (COMMA! expr)*
        { #expr_list = #([EXPR_LIST,"EXPR_LIST"], expr_list); }
    | { #expr_list = #([EXPR_LIST,"EXPR_LIST"]); }
    ;
```

```
header
{
package sigl.parser;
}

{
import java.io.*;
import java.util.*;

import sigl.core.*;
}

class SIGLWalker extends TreeParser;
options
{
    importVocab = SIGL;
}

stmt returns [Stmt s]
{
    Expr e1 = null, e2 = null;
    s = null;
    Stmt s1 = null, s2 = null;
    Vector v = null, v1 = null;
}
    : #("if" e1=expr s1=stmt
       ( s2=stmt { s = new If(e1, s1, s2); } | /* nothing */ { s = new If(e1, s1
); } )
    )
    | #("while" e1=expr s1=stmt { s = new While(e1, s1); } )
    | #("for" v=expr_list (e1=expr)? v1=expr_list s1=stmt { s = new For(v,e1,v1,
s1); })
    | #(STMT_LIST { Block block = new Block (); s = block; }
       (s1 = stmt { block.addStatement (s1); })*
    )
    | #(PROGRAM { Block block = new Block (); s = block; }
       (s1 = stmt { block.addStatement (s1); })*
    )
    | BREAK { s = new Break(); s.setLine(#BREAK.getLine()); s.setColumn(#BREAK.g
etColumn()); }
    | CONTINUE { s = new Continue(); s.setLine(#CONTINUE.getLine()); s.setColumn
(#CONTINUE.getColumn()); }
    | #(RETURN (e1 = expr { s = new Return(e1); } | /* nothing */ { s = new Retu
rn(); })
       { s.setLine(#RETURN.getLine()); s.setColumn(#RETURN.getColumn()); }
    )
    | #("function" ID v=formal_list s1=stmt { s = new Function(#ID.getText(), v,
 s1); })
    | #(TRANSFORM e1=func_call s1=stmt { s = new Transform (e1,s1); } )
    | e1=expr { s = new ExprStmt (e1); }
    ;

formal_list returns [Vector v]
{
    v = new Vector();
    Stmt s;
}
    : #(FORMAL_LIST
       (ID { v.add (#ID.getText()); })*
    );

expr_list returns [Vector v]
{
    v = new Vector();
```



```
    Expr e;
}
    : #(EXPR_LIST
       (e=expr { v.add (e); })*
    );

expr returns [Expr e]
{
    Expr a, b;
    e = null;
}
    : #(LOR             a=expr b=expr   { e = new LogicalOperation("||", a, b);
e.setLine(#LOR.getLine()); e.setColumn(#LOR.getColumn()); } )
    | #(LAND            a=expr b=expr   { e = new LogicalOperation("&&", a, b);
e.setLine(#LAND.getLine()); e.setColumn(#LAND.getColumn()); } )
    | #(EQUAL           a=expr b=expr   { e = new RelationalOperation("==", a, b
); e.setLine(#EQUAL.getLine()); e.setColumn(#EQUAL.getColumn()); } )
    | #(NOT_EQUAL       a=expr b=expr   { e = new RelationalOperation("!=", a, b
); e.setLine(#NOT_EQUAL.getLine()); e.setColumn(#NOT_EQUAL.getColumn()); } )
    | #(LT              a=expr b=expr   { e = new RelationalOperation("<", a, b)
; e.setLine(#LT.getLine()); e.setColumn(#LT.getColumn()); } )
    | #(LTE             a=expr b=expr   { e = new RelationalOperation("<=", a, b
); e.setLine(#LTE.getLine()); e.setColumn(#LTE.getColumn()); } )
    | #(GT              a=expr b=expr   { e = new RelationalOperation(">", a, b)
; e.setLine(#GT.getLine()); e.setColumn(#GT.getColumn()); } )
    | #(GTE             a=expr b=expr   { e = new RelationalOperation(">=", a, b
); e.setLine(#GTE.getLine()); e.setColumn(#GTE.getColumn()); } )
    | #(PLUS            a=expr b=expr   { e = new ArithmeticOperation("+", a, b)
; e.setLine(#PLUS.getLine()); e.setColumn(#PLUS.getColumn()); } )
    | #(MINUS           a=expr          { e = new UnaryOperation("-", a); }
                        (b=expr         { e = new ArithmeticOperation("-", a, b)
; } )?
                        { e.setLine(#MINUS.getLine()); e.setColumn(#MINUS.getCol
umn()); }
    )
    | #(MUL             a=expr b=expr   { e = new ArithmeticOperation("*", a, b)
; e.setLine(#MUL.getLine()); e.setColumn(#MUL.getColumn()); } )
    | #(DIV             a=expr b=expr   { e = new ArithmeticOperation("/", a, b)
; e.setLine(#DIV.getLine()); e.setColumn(#DIV.getColumn()); } )
    | #(MOD             a=expr b=expr   { e = new ArithmeticOperation("%", a, b)
; e.setLine(#MOD.getLine()); e.setColumn(#MOD.getColumn()); } )
    | #(LNOT            a=expr          { e = new UnaryOperation("!", a); e.setL
ine(#LNOT.getLine()); e.setColumn(#LNOT.getColumn()); } )
    | INT_CONSTANT                      { e = new IntConstant(Integer.parseInt (
#INT_CONSTANT.getText())); e.setLine(#INT_CONSTANT.getLine()); e.setColumn(#INT_
CONSTANT.getColumn()); }
    | REAL_CONSTANT                     { e = new RealConstant(Double.parseDoubl
e (#REAL_CONSTANT.getText())); e.setLine(#REAL_CONSTANT.getLine()); e.setColumn(
#REAL_CONSTANT.getColumn()); }
    | "true"                            { e = new BoolConstant (true); }
    | "false"                           { e = new BoolConstant (false); }
    | #(ASSIGN          a=expr b=expr   { e = new Assign (a, b); e.setLine(#ASSI
GN.getLine()); e.setColumn(#ASSIGN.getColumn());} )
    | #(PLUS_ASSIGN     a=expr b=expr   { e = new Assign (a, new ArithmeticOpera
tion ("+", a, b));  e.setLine(#PLUS_ASSIGN.getLine()); e.setColumn(#PLUS_ASSIGN.
getColumn());} )
    | #(MINUS_ASSIGN    a=expr b=expr   { e = new Assign (a, new ArithmeticOpera
tion ("-", a, b));  e.setLine(#MINUS_ASSIGN.getLine()); e.setColumn(#MINUS_ASSIG
N.getColumn());} )
    | #(MUL_ASSIGN      a=expr b=expr   { e = new Assign (a, new ArithmeticOpera
tion ("*", a, b));  e.setLine(#MUL_ASSIGN.getLine()); e.setColumn(#MUL_ASSIGN.ge
tColumn());} )
    | #(DIV_ASSIGN      a=expr b=expr   { e = new Assign (a, new ArithmeticOpera
tion ("/", a, b));  e.setLine(#DIV_ASSIGN.getLine()); e.setColumn(#DIV_ASSIGN.ge
```

```
tColumn());} )
    | #(MOD_ASSIGN      a=expr b=expr   { e = new Assign (a, new ArithmeticOpera
tion ("%", a, b));  e.setLine(#MOD_ASSIGN.getLine()); e.setColumn(#MOD_ASSIGN.ge
tColumn());} )
    | e=func_call
    | #(ID
        {
            Variable v = new Variable(#ID.getText());
            v.setLine(#ID.getLine()); v.setColumn(#ID.getColumn());
            Vector indices = new Vector();
            v.setIndices (indices);
            e = v;
        }
        (a=expr
            {
                indices.addElement (a);
            }
        )*
        )
    ;

func_call returns [Expr e]
{
    e = null;
}
    :#(FUNC_CALL
        {
            e = new FunctionCall ();
            Vector args = null;
        }
        ID { ((FunctionCall)e).setFunctionName (#ID.getText()); e.setLine(#ID.ge
tLine()); e.setColumn(#ID.getColumn()); }
        args=expr_list { ((FunctionCall)e).setArguments (args); }
    )
    ;
```

```
// convention:
// F => 1
// + => 2
// − => 3
// X => 4
// Y => 5

// X −> X+YF+, Y −> −FX−Y
function dragon_curve(a)
{
    i = 0;
    size = length(a);
    for (i=0;i < size;i+=1)
    {
        if (isarray(a[i]))
        {
            dragon_curve(a[i]);
        }
        else if (a[i] == 4)
        {
            // found an X
            a[i][0]=4;
            a[i][1]=2;
            a[i][2]=5;
            a[i][3]=1;
            a[i][4]=2;
        }
        else if (a[i] == 5)
        {
            // found an Y
            a[i][0]=3;
            a[i][1]=1;
            a[i][2]=4;
            a[i][3]=3;
            a[i][4]=5;
        }
    }
}

function draw(a)
{
    i=0;
    while (i < length(a))
    {
        if (isarray(a[i])) draw(a[i]);
        else
        {
            if ((a[i] == 1) ||
                (a[i] == 4) ||
                (a[i] == 5))
            {
                line(0,0,2,0);
                translate(2,0);
            }
            else if (a[i] == 2) rotate(90);
            else rotate(−90);
        }
        i+=1;
    }
}

total_depth=11;
a[0]=1;
a[1]=4;
```



```
for (i=0;i < total_depth;i+=1) dragon_curve(a);
j=myfunction(2);
print(j);
:translate(200,50): draw(a);
```

```
while( true ) {
    while( a[i] < v) i = i+1;
    while( a[j] > v) j = j-1;
    if ( i >= j ) break;
    x = a[i]; a[i] = a[j]; a[j] = x;
}

{
    if((a+b)>(b+c)&&(f1<f2)){
        array [a][b] = f1;
    }
    else{
        array [b][a] = f2;
    }
    while(f1!=f2||f2!=f3){
        b1=true;
        a=b+c;
    }
    while(array[a][b]==2.5){
        b1=false;
        a=b+c/250;
    }
}
```

```
function sum(a)
{
    i=0;
    total=0;
    while (i<length(a))
    {
        total+=a[i];
        i+=1;
    }
    return total;
}

function recursive(a,size)
{
    if (size == 0) return 0;
    return a[size − 1] + recursive(a,size − 1);
}

a[0]=5;
a[1]=4;
a[2]=3;
a[3]=2;
a[4]=1;
i=3;
count1=sum(a);
count2=recursive(a,5);
print(i, count1, count2, 8 % 3, length(a));
```

```
n=99999;
function fac(n)
{
    if (n==0) return 1;
    else return n * fac(n − 1);
}
print(n);
print(fac(5));
print(n);
{
    b=5;
    n=364;
}
print(n);
print(b);
```

```
function test()
{
    print(a);
}
a = 10;
test();
```

```
function add (a, b) { return a + b; }
function sub (a, b) { return a - b; }
function mul (a, b) { return a * b; }
function div (a, b) { return a / b; }

function apply (f,a,b)
{
    return f(a,b);
}

a = 10;
b = 5;
print(apply(add,a,b));
print(apply(sub,a,b));
print(apply(mul,a,b));
print(apply(div,a,b));
```

```
function runs_forever (i)
{
    return i + runs_forever(i + 1);
}

function trivial (a, b)
{
    print(b);
}

trivial (runs_forever(0), 10);
```

```
function wheel(size)
{
    ellipse(size,size);
    for (i = 0;i < 360;i += 45)
    {
        :rotate(i):line(0,-size/2,0,size/2);
    }
}

function rect(x1,y1,x2,y2)
{
    polygon(x1,y1,x1,y2,x2,y2,x2,y1);
}

function truck()
{
    back_length = 200;
    back_height = 100;
    head_length = 40;
    head_height = 50;

    // draw the body
    rect(0,0,back_length,back_height);
    :translate(back_length,back_height - head_height):rect(0,0,head_length,head_
height);

    // draw 2 wheels
    wheel_size = 20;
    :translate(20,back_height):wheel(wheel_size);
    :translate(back_length + head_length / 2,back_height):wheel(wheel_size);

    // decoration
    angle = 45;
    size = 25;
    :translate(20,back_height / 2):
        for (i = 0;i < 3;i+=1)
        {
            :rotate(angle):ellipse(size, size / 2);
            translate(size * 1.5,0);
            size *= 2;
            angle = -angle;
        }
}

:translate(50,50):truck();
```

```
function factorial (n)
{
    if (n == 0) return 1;
    else return n * factorial (n − 1);
}

print(factorial(5));
```

Table of Contents