IML

An Image Manipulation Language

Steven Chaitoff • Eric Hu • Cindy Liao • Zach van Schouwen

Columbia University
COMS W4115 Programming Languages and Translators
Spring 2007

Professor Stephen Edwards

Table of Contents

Introduction

Overview

Description

IML is a programming language designed for easy image manipulation and batch processing. With constructs for flow control and mathematical operations, IML is a strong procedural language for mathematical computations. In addition, it has constructs for manipulating pixel data and opening and saving images. Combining the two, IML may serve to make complex image transformations and effects.

Usability

The syntax of IML is meant to be familiar so users may code speedily and learn the language quickly. For this reason, it uses a C like syntax. To aid beginning programmers, particularly for constructs unique to IML, a principal design goal was for the language to be simple and unambiguous: interactions with the file system are diminished to commands for opening and saving. Opaque constructs like typecasting are forbidden, supplanted by a duck typing principle that evaluates expressions in a way the makes sense. In general, we intend that programs in IML do what the user expects.

Image formats that IML can open are GIFs, JPEGs, PNGs and BMPs. It can save images as JPEG, PNGs and BMPs. Inside IML programs, images are converted to bitmaps – two dimensional arrays of pixels – to be manipulated easily, meaning images can be opened in one format and saved in another. In this respect, IML can be used as a format conversion tool.

Portability

IML is an interpreted language that is converted to Java at runtime. As Java is platform-independent, IML can be run on any machine architecture that supports Java 1.5 or higher.

Language Features

Data Types

IML has five primitive data types: `Int`, `Float`, `String`, `Pixel` and `Image`. An `Image` is comprised of `Pixel`s. In addition, arrays of each of these types are permitted, as are arrays of arrays, hence any type can have any degree of dimensionality.

Keywords

Keywords distinguish language constructs for flow-control and variable manipulation. All keywords in IML are lowercase, and they may not be used as variable names.

Operators

Operators are symbols used to perform computations on variables. IML has operators for low-level logical and arithmetic calculations and comparisons, as well as an operator for assigning values to variables. The symbols for operators are standard i.e. `+` denotes addition; `/` denotes equality in comparison, etc. Operators conform to standard order of operations and parenthesis can be used to compute lower precedence operations before higher precedence ones. In addition, operators may be chained, e.g. `1+2+3+4+5`. Operators are overloaded; that is, their behavior is dependent upon the operands on which they act, and operators may not be used at all with some operands.

Control Flow Statements

Three main constructs comprise control-flow: the *for statement* repeats a section of code (a block) until a condition is met. The *while statement* repeats a block while a condition is met, and will stop repetition until that condition becomes untrue. Finally, the *if/else statement* involves two blocks, executing the first and not the second if a condition is met, and the second but not the first if that condition is not met. *If* clauses may be used without a corresponding *else* clause. In addition, the *break* statement immediately stops repetition of a *while* or *for* loop as if the qualifying condition had suddenly not been met.

Functions

Functions are named subroutines within a program. An IML program is a series of functions, meaning all computations must be done inside functions. Functions may not be nested, and each function is denoted by the word “function” followed by its name and parameters. The last function must be named “main”, which is where program execution begins.

Variables

Variables are storage containers for values. Every variable has an associated data type, restricting the values that it can represent. Variables must be declared exactly once before assignment, and assignment cannot be combined with declaration, e.g. `Int a = 5;` is forbidden.

Scope

All variables must be placed inside of a scope, defined by curly brackets. Scopes themselves can be nested, and a variable is only defined within the scope that it was declared, and its entire child scopes.

Function scopes are considered child scopes, meaning that the scope of a function is a child of the scope of the parent function that called it, and all bound variables in the parent function are accessible in the child function, unless redefined.

Preprocessor directives

IML supports preprocessor directives of the form # *filename* written at the beginning of the interpreted iml file. In this way, it is possible to separate IML programs into separate files or include the IML standard library.

Sample Syntax

Variables and Functions

```
Int a;

Float number;
number  = 3.14;

String str;
str = "string assignment";

Image a;
Image b;
a = b;

function pi() {
        return 3.14159265;
}

function add_numbers(int a, int b) {
        return a + b;
}
```

Arrays

```
Pixel p[20][30];

function get_pixel(Pixel p[][]) {
        return p[5][5];
}
```

Control Flow

```
Int i;
for (i = 0; i < 10; i = i + 1) {
        print(i);
}

Int j;
j = 2048;
while (j % 2 == 0) {
        print (j = j / 2);
}

Pixel k;
if (red k == blue k && blue k == green k) {
        print ("shade of grey");
}
else {
        print ("color");
}
```

Complete Program

```
function name_1(arg_1, arg_2, …) {
        statement_1;
        statement_2;
        …
}

function name_2(arg_1, arg_2, …) {
        statement_1;
        statement_2;
        …
}

…

function main() {
        name_1(expression_1, expression_2);
        statement_1;
        statement_2;
        …
}
```

Language Tutorial

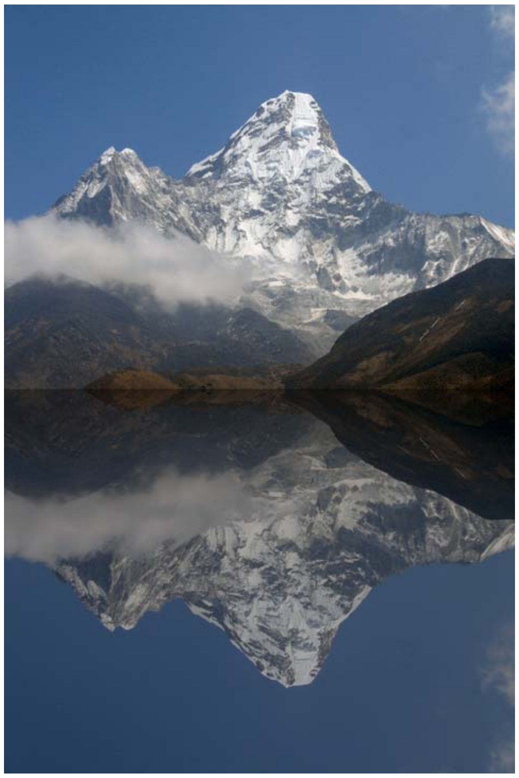
Description

Presented below is an IML program that demonstrate the main attributes of the language.

Mirror Tutorial

Mirror Code

```
/*
* File: mirror.iml
* opens an image, attaches a reflected, translucent
* copy of it underneath and saves this new image
*/

function image_mirror(Image s) {
        Int h;  h = rows s;     // h is height of image
        Int w;  w = cols s;

        Pixel mirror[w][2 * h];

        Int i;  Int j;
        for ( ; i < w; i = i + 1) {
                for (j = 0; j < h; j = j +1) {
                        mirror[i][j] = mirror[i][2 * h - j - 1] =s[i][j];
                }
        }

        return mirror;
}


function image_translucent(Image r) {
        Int i; Int j;
        Float translucency;
        translucency = 0.8 * 255;
        for (i = rows r / 2; i < rows r; i = i + 1) {
                for (j = 0; j < cols r; j = j + 1) {
                        alpha r[j][i] = translucency;
                }
        }
        return r;
}


function main() {
        Image scene;
        "landscapes/mountain.jpg" open scene;

        scene = image_mirror(scene);
        scene = image_translucent(scene);

        scene save "landscapes/mountain_lake.jpg";
        print("Finished execution");
}

/*
* end of mirror.iml
*/
```



Mirror Explanation

Program execution begins with the main function, which must be the last function in the program. Main does not take arguments and does not return a value. `Image scene` declares a new `Image` type named `scene`. The `open` command is used to load pixel data into the image. `open` always has operators `String` on the left and `Image` on the right, where `String` is the pathname to the image file to be stored in image. `scene` now holds the pixel data of the image file opened.

`scene = image_mirror(scene)` passes the `scene` image into the `image_mirror` function, which modifies scene and returns it, and the resulting modified image is reassigned to `scene`. Upon calling `image_mirror`, execution jumps the the `image_mirror` function, whose `Image` argument `s` takes the place of `scene`. Two `Int` types are declared, that default to zero. The `rows` operator in `h = rows s` retrieves the number of rows of pixels in `Image s`. The number of rows is then stored in `h`. The `cols` operator similarly retrieves the number of columns of pixels in an `Image`. `Pixel mirror[w][2 * h]` declares a two dimensional array of `Pixel`s named `mirror`, the same width as the `scene` image but twice the height.

The nested for-loops are a common structure in IML that iterates through every pixel in an image based on its width and height (the number of rows and columns). Inside the loops is `mirror[i][j] = mirror[i][2 * h - j - 1] = s[i][j]`, which is a chained equality equivalent to the two lines:

```
    mirror[i][j] = s[i][j];
    mirror[i][2 * h - j - 1] = s[i][j];
```

All pixels in the input image `s` are copied to the same location in the `mirror` image. Further, all pixels in the input image are copied *from the bottom of `mirror` up*. Because `mirror` is twice the height of `s`, `mirror` contains a copy of `s` in its top half and an upside-down copy of in `s` its bottom half. After copying all the pixels, `mirror` is returned to the main function.

In main, the `image_translucent()` function is called with the modified `scene`. `image_translucent` iterates through the image in the same fashion as `image_mirror`, except it begins iteration halfway down the image, only affecting pixels below the middle row of pixels. The `alpha` operator in `alpha r[j][i] = translucency` retrieves the alpha channel from the the (j, i)th pixel in the image, and it is assigned the value `translucency`. The effect is to make all pixels in the bottom half of the image somewhat translucent. Alpha values, as well as red, green, and blue values for a pixel range between 0 and 255. A value of 255 for alpha denotes a completely opaque pixel. Setting `translucency = 0.8 * 255` then indicates that alpha should be set to 80 percent of the maximum. Note that `translucency` is a `Float` type, but alpha must be an integer. IML implicitly casts the `Float` to an `Int` by flooring.

After returning to main, the modified `scene` image is saved to a new location with a save operator. The `print()` function prints a string to the console to indicate that program execution is complete. `print()` is built into IML and does not need to be written.

Language Reference Manual

Lexical Description

Tokens

There are six types of tokens: identifiers, keywords, constants, string literals, operators, and separators. Comments are ignored. Any white space (defined as any number of spaces, tabs, newlines or feeds) is also ignored, except as to separate adjacent tokens where no white space separation would be ambiguous.

Comments

There are two types of comments. Single-line comments begin with `//` characters and continue until a newline character. Multi-line comments may begin and end with `/*` and `*/`, respectively. Comments do not nest.

Identifiers

Identifiers are any sequence of one or more letters, numbers, and underscores. The first character of an identifier must not be a number. Upper and lower case letters are different, and identifiers may be any length.

Keywords

The following words are reserved and may not be used as identifiers.

```
alpha    blue    break   cols    else   function
green    for     if      length  open   red
return   rows    save    while
```

Constants

<u>Integer Constants</u>
Integer constants consist of a sequence of one or more digits assumed to be in base 10.

<u>Floating Point Constants</u>
Floating points constants consist of an integer part, a decimal point, and a fraction part. Both the integer and fraction parts must consist of a sequence of one or more digits.

String Literals

String literals are any sequence of characters beginning and ending with a `"` character. Any `"` characters excluding those the beginning or end of the sequence must be preceded by a `\` character. Any `\` characters that are not part of an escape sequence must be preceded by a `\` character as well.

Separators

Separators are tokens that serve to delimit the position of other tokens.

<u>Delineators</u>
`(` and `)` characters surround expressions and argument lists for function calls. `{` and `}` characters delineate the scope of identifiers inside of them. The `,` character separates both arguments in a function call and parameters in the argument list of a function

declaration.

Terminators
Every statement that does not precede a statement-block must end with the `;` character.

Syntax Notation

Basic Types

Pixel Type
The Pixel type stores one pixel in memory, which consists of red, green, blue and alpha (RGBA) byte values whose intensity values range from 0 to 255. Any RGBA value set greater than 255 will revert to 255, and any value set to less than 0 will revert to 0. The default RGBA value of a Pixel upon declaration is 0 red, 0 green, 0 blue, and 0 alpha, which is a black and transparent pixel.

Image Type
The Image type stores an image in memory, which consists of a two-dimensional array of Pixel types. The default value of an Image upon declaration is a 1x1 array composed of one Pixel.

Integral Type
The Integral type (Int) stores a 32 bit signed integer in memory. Values can range from -2147483648 to +2147483647. The default value of an integer upon declaration is 0.

Floating Point Type
The Floating Point type (Float) stores a 64 bit double precision floating point number in memory. The default value for a floating point upon declaration is 0.0.

String Type
The String type is used to store character strings in memory. The default value of a string upon declaration is the empty string, "". A string can be initialized by setting it to a literal:

```
String hello;     // hello is the empty string
hello = “hello world!\n”;
```

Derived Types

Array Types
It is possible to create an array of any type, including an array itself (multi-dimensional arrays). Array types are denoted with square brackets ([ ]) following an identifier. The number of elements in an array is static; this number must be specified during declaration. Array indices begin at 0.

Function Types

Functions are denoted by the word “function” followed by its identifier, an optional sequence of input arguments, and a series of statements, possibly including other function calls and recursive calls. Functions may or may not return a value, and multiple return statements are permitted. Function return types are unspecified, meaning a function may return different types depending on the context in which it is called.

LValues

An *lvalue* is an expression referring to an *object*, or a named region of storage. All basic types and array types yield an lvalue equal to its stored value. This allows assignments like the following:

```
      Int a;      Int b;
      a = b = 5; // b is assigned 5 and also returns a value of 5
```

Conversions

A variable always maintains its type throughout program execution. Conversions are permitted and implicitly attempted to maintain the consistency of a type. Any illogical conversions generate a syntax error.

Casting

No explicit casting is allowed.

Arithmetic Conversions

Ints, when assigned to Float types, will convert to a Float type. Float types, when assigned to Ints, will have any digits following the decimal point truncated and then converted to an Int:

```
      Int a;      a = 4.5;    // a is converted 4
```

Numerical and String Conversions

Ints and Floats, when assigned to strings, will be converted to a string containing the value of the number. Strings cannot be assigned to numbers and doing so will result in a syntax error.

Pixel and Image Conversions

Pixel and Image types cannot be directly converted. Only a two-dimensional array of pixels, which is an Array type, can be converted to an Image.

Expressions

The operators defined in the following expressions appear in decreasing precedence.

Primary Expressions

A primary expression is any identifier, constant or parenthesized expression, which may itself be a primary expression. Therefore, a primary expression consists of any sequence of identifiers, constants and operator expressions ordered by the associativity and precedence rules of the operators therein.

Postfix Operators

Array References

Array references are indicated by square brackets ([ ]) following the array identifier on which they operate. They are in postfix form and are left-to-right associative, meaning multi-dimensional references are expanded outwardly:

```
// a is a three dimensional array of size >= 1x2x3
( a[0][1][2] == (((a[0]) [1]) [2]) ) // evaluates true
```

Array references on Image types allow access to Pixel types.

```
Image myimage;    Pixel mypixel;
mypixel = myimage[0][0]; // upper left most pixel of image
```

Array references on String types return a string containing the character referenced:

```
String h; h = “hello”;
h = h[0]; // h has value “h”;
```

String character references are not lvalues:

```
h = “hello”;      h[0] = “y”; // not permitted
```

Function Calls

Function calls are postfix expressions consisting of an optional comma-separated list of argument expressions:

```
Int a;    Int b;
a = 1;      b = 1;
sum (1, 1) ;      // function call
```

Unary Operators

Unary Minus Operator

The unary minus (-) is a prefix operator that negates integer and floating-point values. It has right-to-left associativity.

Sizing Operators

Sizing operators are prefix operators and have right to left associativity. Sizing operators do not return lvalues, as arrays are fixed in size.

Length Operator
The length operator can only be applied to array types. It returns the number of elements in an array. Although the Image type behaves like a two-dimensional array and the array reference operators can be applied to it, the length operator cannot because it is not an array type.

```
// a is an array of 10 String types
String a[10];     (length a == 10)  // evaluates true
```

Rows Operator
The rows operator can only be applied to Image types. It returns the height of the image in pixels.

Cols Operator
The cols operator can only be applied to Image types. It returns the width of the image in pixels.

Channel Operators

Channel operators are prefix operators and have right to left associativity. Channel operators can only be applied to Pixel types, and they return lvalues, therefore channel properties of a Pixel can be modified.

Red Channel Operator
The red operator returns the red intensity value of a Pixel:

```
Pixel a;    red a = 0;  // remove the red hue from a pixel
```

Green Channel Operator
The green operator returns the green intensity value of a Pixel.

Blue Channel Operator
The blue operator returns the blue intensity value of a Pixel.

Alpha Channel Operator
The alpha operator returns the alpha transparency value of a Pixel.

Multiplicative Operators

Multiplicative operators are multiplication (+), division (/) and modulus (%). They are infix operators with left-to-right associativity. Multiplication demands arithmetic operands. If both operands are Int types, the result is an Int type. If either operand is a Float type, the result is a float type. Division also demands arithmetic operands. The result is always a Float type. Modulus demands Int type operands, and the result is always an Int type.

Additive Operators

Additive operators are addition (+) and subtraction (-). They are infix operators with left-to-right associativity. In addition of arithmetic operands, the result is the sum of both operands. If both operands are Int types, the result is an Int type. If either operand is a Float type, the result is a float type. In addition of String operands, the result is the concatenation of the right-hand String to the left-hand String. In addition of an Int operand to a String operand (String + Int), the Int operand is converted to a String, and the two Strings are concatenated.

The subtraction operator demands arithmetic operands. If both operands are Int types, the result is an Int type. If either operand is a Float type, the result is a Float type.

Relational Operators

Comparison Operators
Comparison operators are less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=). Comparison operators are infix operators and have left-to-right associativity. They demand arithmetic operands. They return the value 1 if the relationship between their operands is true, and 0 if it is false.

Equality Operators
Equality operators are equal to (==), and unequal to (!=). They behave in the same way as comparison operators except with lower precedence.

Logical Operators

And Operator
The logical and operator (&&) returns 0 if any of its operands are false. Otherwise it returns 1. False operands consist of 0 (Int), 0.0 (Float), and "" (an empty String). True operands consist of all non-false operands.

Or Operator
The logical or operator (||) returns 0 if all of its operands are false. Otherwise it returns 1.

Assignment Operator

The assignment operator (=) is an infix operator with right-to-left associativity. Its left-hand operand must be an lvalue. Assignment puts the value of the right-hand operand into the storage space pointed at by the left-hand operand.

File I/O Operators

Open Operator
The open operator is an infix operator that opens an image file on disk and stores it in an Image type. It must only be applied with a String operand on the left and an Image operand on the right. The String operand indicates the file path of the image file:

```
Image a;
"/path/to/photo.jpg" open a;
```

Save Operator
The save operator is an infix operator that saves an image to disk. It must only be applied with an Image operand on the left and a String operand on the right. The String

indicates the file path of the image file:

```
Image a = "/path/to/photo.jpg";
a save "/path/to/copy.jpg";
```

Declarations

Type Specifiers

Variables are defined by prepending a type specifier to the name of the new variable:

type-specifier identifier ;

All variables must be forward-declared; doing otherwise will result in a runtime error.

Declarators

Array Declarators

Arrays are declared in a similar manner to basic types, but a length specification is required.

type-specifier identifier **[** *expression* **]** *;*

Function Declarators

Functions are defined and declared at once. A function definition is properly formed as:

type-specifier postfix-expression **(** *argument-expression-list?* **)**
statement-block

The argument expression list takes the form of a comma-delimited sequence of variable declarations:

parameter-list := *type-specifier identifier* **[,** *type-specifier identifier* **] ***

Function names are subject to the same language constraints as variable names, as they are identifiers. A function and variable within the same scope may not share the same name, as they are both identifiers.

Initialization

Variables need not be initialized as declaration implies a default initialization. Int types default to 0; floats to 0.0; strings to “”; Pixels to a black, transparent pixel; Images to a 1x1 black, transparent image. Array types initialize all of their elements to the default:

```
Pixel[5][5];      // a 2D array of black, transparent pixels
```

Explicit initialization cannot be combined with the declaration. It must be handled in a separate statement:

```
Int n;      // declaration (n=0)
n = 3;      // initialization
```

Image Initialization
An image is initialized with data using the open keyword and a string literal of the path of a supported image file:

```
Image myimage;    "/path/to/photo.jpg" open myimage;
```

A path to a non-image file or an unopenable image file is a runtime error.

Scoping

Variables have a stack of bindings within a lexical scope. Variable bindings are pushed onto a stack when a new scope is created, and they are popped off when a scope ends. The value of variable is the topmost value in the stack:

```
function child() {
      {     // inner scope
            Int i;
            i = 2;
      }     // end of inner scope; i unbound from 2
      print (i);  // prints 1
}

function main() {
      Int i;
      i = 1;      // i bound to 1
      child();
}
```

Statements

Statements are generally expressions followed by a semicolon to represent the end of a statement. Statements are executed in sequence. Statement-blocks are sequences of zero or more statements.

Selection Statements

Selection statements evaluate conditions and direct control flow appropriately.
Valid selection statement forms are:

if (*expression) statement-block*
if (*expression) statement-block* **else** *statement-block*

Iteration Statements

Basic iteration structures are supported for looping control flow.

For Loops

A valid for statement form is:

> **for (** *expression-statement expression-statement expression-statement* **)**
> *statement-block*

The first expression statement is evaluated before the loop begins; the second is evaluated at the beginning of each iteration (and, if 0, ends loop execution); finally, the last statement is evaluated at the end of each iteration.

<u>While Loops</u>
A valid while statement form is:

> **while (** *expression* **)** *statement-block*

At the beginning of each iteration, if the qualifying expression evaluates to 1, statement-block is executed.

<u>Break</u>
The break keyword must only be placed in the statement-block of an iteration statement. When encountered, execution immediately breaks out of the iterative loop as if the qualifying expression had returned false. The current loop of iteration is not completed. That is, the statements following the break statement in that block are not executed.

Compound Statements

Nested statements are permitted, such that selection and iteration statements can appear inside of a statement block. All statement blocks must begin with an open bracket and end with a close bracket.

<u>Identifer Scope</u>
The scope of a variable is determined by the deepest statement block in which it appears.

Grammar

Nonterminal symbols are written in *italics*. Terminals are written in **boldface**.
A question mark (*?*) character following a non-terminal indicates that it is optional.

function-definition:
> *identifer* **(** *type-expression-list* **)** *statement-block*

statement:
> *statement-block*
> *selection-statement*
> *iteration-statement*

expression-statement
declaration-statement
break-statement
return-statement
io-statement

statement_block:
(*compound-statement* **)**

compound-statement:
statement
compound-statement statement

type-specifier:
Int | Float | String | Pixel | Image

selection-statement:
if (*expression* **)** *statement-block*
if (*expression* **)** *statement-block* **else** *statement-block*

iteration-statement:
while (*expression* **)** *statement-block*
for (*expression-statement expression-statement expression-statement* **)** *statement-block*

break-statement:
break

return-statement:
return

io-statement:
print (*expression* **)**

size-indicator:
[*expression*? **]**
size-indicator **[** *expression*? **]**

type-expression:
type-specifier identifier dimension-idenitifer?

dimension-idenitifer:
[ ]
dimension-idenitifer **[ ]**

type-expression-list:
type-expression
type-expression-list **,** *type-expression*

declaration-statement:
type-specifier identifier size-indicator? **;**

expression-statement:
expression? **;**

expression:

fileio-expression

fileio-expression:
assignment-expression
assignment-expression **save** *assignment-expression*
assignment-expression **open** *assignment-expression*

assignment-expression:
logical-or-expression
logical-or-expression **=** *assignment-expression*

logical-or-expression:
logical-and-expression
logical-or-expression **||** *logical-and-expression*

logical-and-expression:
equality-expression
logical-and-expression **&&** *equality-expression*

equality-expression:
relational-expression
equality-expression **==** *relational-expression*
equality-expression **!=** *relational-expression*

relational-expression:
additive-expression
relational-expression **>** *additive-expression*
relational-expression **<** *additive-expression*
relational-expression **>=** *additive-expression*
relational-expression **<=** *additive-expression*

additive-expression:
multiplicative-expression
additive-expression **+** *multiplicative-expression*
additive-expression **-** *multiplicative-expression*

multiplicative-expression:
unary-expression
multiplicative-expression ***** *unary-expression*
multiplicative-expression **/** *unary-expression*
multiplicative-expression **%** *unary-expression*

unary-expression:
channel-expression
! *unary-expression*
+ *unary-expression*
- *unary-expression*
length *channel-expression*
rows *channel-expression*
cols *channel-expression*

channel-expression:
postfix-expression
red *postfix-expression*
green *postfix-expression*
blue *postfix-expression*

alpha *postfix-expression*

argument-expression-list:
expression
argument-expression-list expression

postfix-expression:
(*expression* **)**
constant-expression
identifier size-indicator
identifier **(** *argument-expression-list?* **)**

constant-expression:
string-constant
integer-constant
floating-constant

Project Plan

Plan Overview

Our group held weekly meetings throughout the semester on Tuesday nights. During meetings we reviewed each team member's individual work, and, if necessary, merged documents and/or programs together. We generally split our team into pairs, each focusing on a specific task. After each major checkpoint in our project, we met with our advisor to evaluate our progress and discuss any questions or concerns.

Development

Our workflow changed considerably over time. At the beginning of the project we found it easiest to meet each week and complete a piece of the interpreter together. We built the Lexer and Parser this way, for example. As the interpreter became more complicated it became apparent that older code needed to be modified, and our linear workflow was not efficient. We began assigning roles to team members such that every person always maintained a large piece of the project. By the end of the project, our dynamic had well defined roles: Zach and Cindy us of wrote code for the front end and back end, respectively, and Eric and Steven did testing and debugging. Still however, we attempted that each member was informed about the different parts of the project. We tended to have overlapping roles, since working on the front end or back end occasionally required making changes to the other, and debugging often required rewriting code. As we completed different phases of the project, we maintained a test suite for that phase, which served as a foundation for our regression test suite.

ANTLR, ANother Tool for Language Recognition

We wrote the lexer, parser, and tree walker in ANTLR, which constructed an abstract syntax tree and a tree walker to traverse it. ANTLR code comprised the interpreter front end.

Java

The back end of the project was written in Java. We used the Java Runtime Environment version 1.6.0 for our project, but Java 1.5.0 suffices to interpret IML. The back end uses a variety of built in Java data structures, including stacks, linked lists, and arraylists. We often consulted the Java 2D API Specification (http://java.sun.com/j2se/1.4.2/docs/guide/2d/spec.html) to ensure our implementation.

Operating System

We used several of different systems over the development of the project. All testing, however, was done on Mac OS X and Red Hat Linux.

IDE

We used Eclipse SDK 3.2.1 when developing code for our project and the ANTLR Studio Eclipse plug-in, version 1.1.0.

Subversion

Our project was under version control throughout its entire development. We used Subversion, which allowed us to access code for our project remotely from both our personal computers and the CLIC lab machines. Subversion was extraordinarily helpful for managing our code.

Documentation

Each of our team members worked on documentation equally.

Programming Style

ANTLR

The most important part of our ANTLR programming style was to keep it spaced well and very readable, since ANTLR tends to have a difficult syntax (particularly in the tree walker).

For this reason, all rules begin with the rule name and any declarations, and all rules end with a single semicolon and two carriage returns. All rule matches (except the first) begin with an or (|) symbol, and the right side of the production has a margin just larger than the |, so it is very easy to see individual matches, even if each match contains a lot of code.

Symbols are written in all capital letters in the lexer, but in all lowercase letters in the parser. All labels in the parser use underscores for spaces.

Java

For the Java back end used a convention that matched many universal Java conventions:

- All lowercase letters for variable names, but capitalize the first letter when separating a variable name into two or more logical words.
- Left braces '{' are at the end of the previous line, and corresponding right braces '}' are on a new line, unless it is a very short block of code.
- Spaces between operators and operands

IML

After writing numerous IML programs for testing, we also developed conventions for writing it:

- Short lowercase variable names
- Long, descriptive lowercase function names with underscores for spaces.
- Tabs between declarations and initializations, or several declarations, e.g.
  ```
  Int x;      Int y;      x = 5;      y = 5;
  ```

In any language, we always attempted to keep line lengths below 80 characters for readability.

Timelines

Project Timeline

Date	Task
1/23/07	Brainstorm ideas and prepare proposals
1/30/07	Pick one idea and finalize the design details
2/7/07	Submit Project Proposal
2/27/07	Create a tentative Lexer and Parser
3/5/07	Submit Language Reference Manual
3/20/07	Create a Walker, modifying the Lexer and Parser as needed
3/27/07	Test basic functionality of the walker, maintain test cases for the regression test suite
4/3/07	Begin running test programs and sample code
5/2/07	Project Completion
5/5/07	Prepare for Presentation
5/7/07	Final Presentation

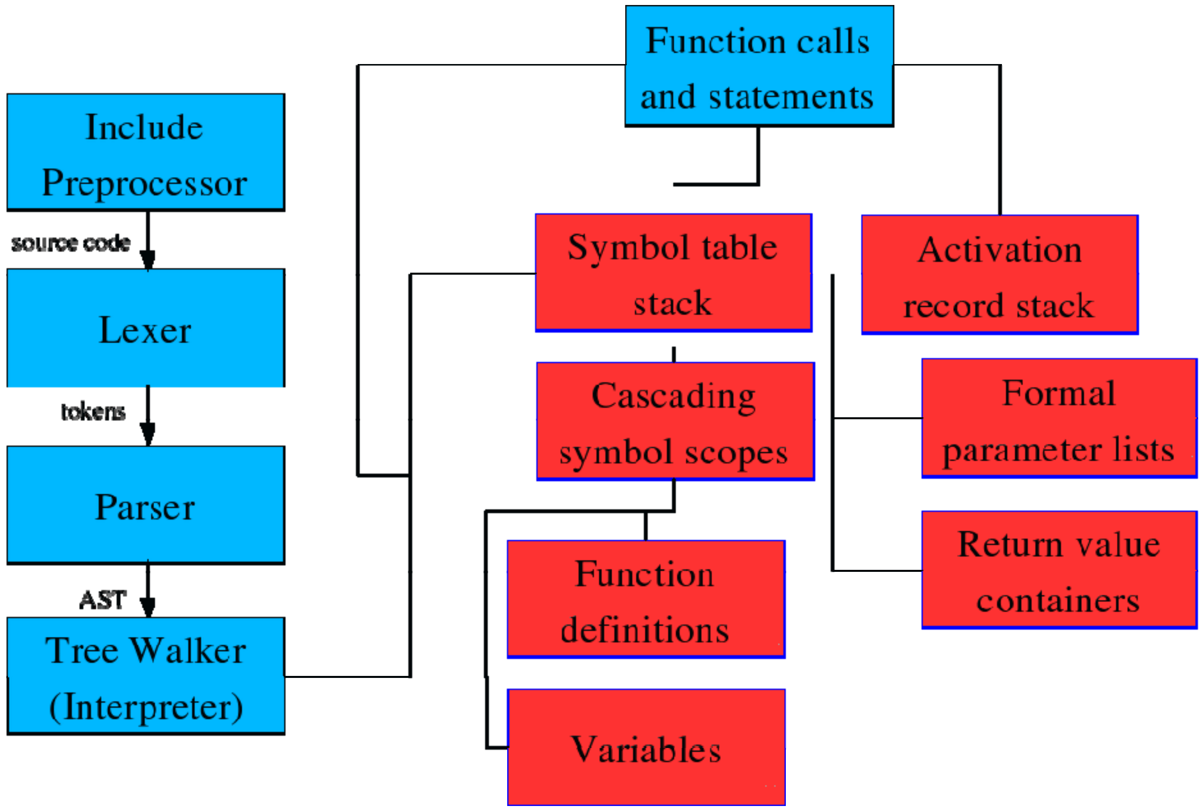
Project Log

Date	Task
1/23/07	Brainstorm designs for a language

1/30/07	Voted on an idea and solidified designs
2/6/07	Finalize Project Proposal
2/7/07	Submit Project Proposal
2/13/07	Reviewed comments on the Proposal, learned basic of ANTLR
2/20/07	Divided up work on Lexer and Parser
2/27/07	Worked on Language Reference Manual
3/6/07	Completed tentative Lexer and Parser
3/13/07	Spring Break
3/20/07	Create a Walker, modifying the Lexer and Parser as needed, verified the validity of the AST structure
3/27/07	Started working on symbol tables, activation records, and recursion handling
3/29/07	Data types, operators, scoping, loops, if/else
4/3/07	Basic testing, code revision
4/10/07	Arrays, arguments, Image datatype, error handling, split up documentation tasks
4/17/07	Writing images, break, return
4/24/07	Tested sample programs, clean up code, finalize comments
5/2/07	Finished all work on the project
5/6/07	Prepare for Presentation
5/7/07	Final Presentation

Architectural Design

Component Diagram



Interpreter Components

IML works within the general framework for a compiler laid out by ANTLR. However, since we opted to write an interpreter, rather than generating Java code, the mechanism by which the tree walker works is changed substantially. The component flow is thereby made somewhat more complicated.

Preprocessor

Code is initially fed through a minimal preprocessor. Include directives (implemented with a pound sign) refer to arbitrary locations in the file system. The files pointed to by these directives are combined with the main IML file to create the ".preprocessor" file, which is then used as the build target.

Lexer

Once the preprocessor completes, the lexer takes the combined file as input and converts it to a stream of tokens, using an extension of the standard ANTLR lexer class.

The result of this operation is stored in the lexer object, which is then used to construct the parser.

Parser

The parser, given this sequence of tokens, generates an abstract syntax tree, according to the rules specified in the grammar. In many cases, we instruct the grammar to create new subtrees to ease the parsing process (for example, in the formal parameter list of the function). The ability to refer to sub-ASTs within the scope of the walker proves very useful later.

Tree Walker

The returned AST is then passed to the tree walker, which is an extension of another ANTLR standard class. The tree walker processes the code in stages, and contains the bulk of the "interpreter" within it. The tree walker begins by instantiating the symbol table. The symbol table is a local variable in the walker, and can be accessed by the various tree functions. We implemented it as a customized stack of hash tables, which allows us to very simply implement dynamic scoping. When the tree walker encounters an identifier, it queries the hash stack, which then checks each hash table, starting from the top, until it finds the innermost scope definition of the specified identifier (or throws a runtime exception if it is not defined).

The tree walker also prepares a empty stack of activation records. Each activation record contains the function parameters and a reference to the symbol where its return value will be placed. The symbol is defined at function call time, before being evaluated in the expression containing the function.

The program is, fundamentally, parsed as a series of function definitions. Upon encountering each definition, the tree walker creates a new entry in the symbol table, containing a reference to the AST subtree corresponding to the function definition. The function code is not walked yet -- any non-parsing errors will be delayed until the function is actually called, a behavior similar to that seen in PHP or JavaScript. The duck typing system that we used precludes most obvious semantic type checking, making an initial semantic pass largely unnecessary.

Interpretation

When the program finds the definition for `main()`, it does not follow the usual function definition procedure -- instead, it instantiates a new activation record with a blank argument list and a container for the return value of main, if any. Once the record has been created, main is executed as a block.

At the beginning of each block's execution, `enter_scope()` is called, creating a new scoping layer (that is, a new hash table is added to the stack). Blocks are then treated as a mass of declarations and statements, which are executed individually.

When a statement hits a function call, the function definition is retrieved from the symbol table. A new activation record is created, and the interpreter attempts to copy the parameters into the local scope. We implemented this operation in such a way that it is subject to the same constraints as the "=" operator. As such, limited type casting is supported (integers to floats, all objects to strings), but egregious abuse of the duck typing system generates runtime errors. A local symbol is then created, and a reference to it is placed in the activation record. `return` statements in the function attempt to assign a value to this symbol. Since IML does not require functions to specify a return type, this symbol may be overwritten, and is not type-checked until it is evaluated after the function exits.

What we managed to compose was a Turing-complete, multi-file, duck-typed interpreter. The interpreter is the final module of the program – it uses the image toolkit to read and write from image files on the disk, a runtime operation that occurs during regular program execution.

Component Development

Preprocessor

The preprocessor was implemented by Zach.

Lexer and Parser

The ANTLR grammars that implement the lexer and parser were initially written by the entire team. Steven and Eric improved the parser significantly and added numerous modifications to allow for more predictable behavior in the walker. Zach and Cindy made small modifications during the final project stages.

Tree Walker

We implemented the tree-walker pairs – Eric and Steven did the initial implementation; Zach and Cindy did initial debugging; all four members added many features during the enrichment phase; Steven and Zach did the final round of debugging. Eric and Steven also wrote a battery of regression tests.

Interpreter

The interpreter code was split among group members: Eric implemented the operations modules for mathematical and logical operations, and arrays; Cindy implemented the image module for file i;o and wrote pixel operations; Zach implemented the symbol tables, activation records, parameter storage, and scoping; Steven wrote the structure and the early implementation of the function table, which was later agglomerated into the symbol table.

Final Stages

The regression test module was written by Cindy. All members of the team composed regression tests, modified, and debugged extensively.

Test Plan

Automation

Automation was achieved by a simple Java program that essentially ran all the tests within folder through the interpreter and compared the output to the expected result for that test. All tests were given an ".iml" extension and all results were given ".result" extension. To simplify testing, we temporarily redirected stdout and stderr to a separate file for each test and compared the contents of that file to the expected result. If a test failed, the test module would print out the name of the test, the expected result, and the actual result.

Test Suite

Most of the test cases were written to the test the more basic aspects of the language, such as mathematical operations, variable declarations, function declarations and calling, array access, loops, conditionals, and recursion. Test cases were also written to test pixel generation and manipulation by printing out and comparing pixel values.

To test image manipulation, the test case would use small resolution images and save to bmp to avoid possible issues from compression. Comparing the resulting image's pixel values with the expected image's pixel values would determine if the test passed or not.

To test the rigor of the language, a separate set of "failure" cases were also written designed specifically to test if the language would catch errors. If any of these cases passed, then it meant something in the language needed fixing.

The test suites, including expected results, and regression test module source code can be found in the Appendix.

Source and Target Language

Since our language is interpreted, no target language is actually generated.

Test Suite Development

The regression test program was written by Cindy. Most of the initial tests checking the basic language operations were written by Steven and Eric. Zach and Cindy wrote most of the tests checking pixel and image manipulations. Zach also wrote the preprocessor test. Everyone was involved in heavy debugging of the tests and expected results.

Lessons Learned

Steven Chaitoff

It is been a fantastic experience for me building IML. Creating the interpreter could be infuriatingly troublesome at times, but it was this difficulty that has made the project worthwhile in the end. With time and perseverance my team and I were able to create our own language, when four months ago I hadn't any idea how to approach this sort of project.

In my estimation, two key factors facilitated our project. Firstly, our team met frequently, and we often wrote code during those meetings instead of using them to check up on each other's progress. Diagrammatically, a translator can be split into a "front end" and "back end," but as it turns out, these two development processes have much to do with each other. All the members of our team understood what each part of the interpreter did, which proved very useful in the end.

Secondly, I found it extremely helpful to read documentation before coding, particularly about ANTLR. I was unfamiliar with ANTLR and, at times, coding in it without knowing exactly what it did. If I were to give one piece of advice to future groups, it is to make sure you know what your code does! (Even if it seems to be doing the right thing).

Eric Hu

Organization: We should have spread out our work more throughout the semester; we were somewhat behind by the spring break checkpoint. We should have been stricter about establishing a set interface specification so that we could split up the work without interfering with each other.

Testing: Writing test code for the sections or functionality you coded yourself is not enough, because chances are, if you wrote a test case for it, then you have already accounted for it in the code. It is helpful to have someone else try writing test code for your section, because the new approach has a better chance at finding a bug. This being said, it is still important for you to test the basic functionality before the more rigorous tests.

Working Together: Teamwork is both good and bad. One programmer working 4 times as long could write more code than 4 programmers working together on the same code. It is important to split up the work so that team members don't interfere with each other's sections and slow each other down. However, it is helpful to brainstorm and design together as a team. It is also helpful when stuck on a bug or logical problem, because a fresh set of eyes could be really helpful.

Finally, version control is wonderful when working on the same files. Having a log of changes, and being able to see who made the changes is useful if a problem occurs, or for reverting changes.

Cindy Liao

Modularization: Classes are extremely helpful for division of labor. More code that can modularized into a separate class means less people will be working on the same file. Also, it is very important to predefine the set of interfaces that each member of the team would strictly adhere to. We had some problems initially where new pieces of code from one member were breaking another member's code, thus making the merging process more difficult.

Testing Early: Regression testing wasn't properly set up until we had some of the basic functions of the interpreter working (i.e. print). It may have saved us a lot of the time if we started regression testing when the AST was written to catch a lot of stupid mistakes from the parser *before* seriously working on the walker.

Lastly, some advice for future teams: Start early and have regular, face-to-face meetings. Even though everyone was assigned their parts, it would still very helpful to actually meet in person to discuss how each part worked and ultimately how they can be combined together.

Zach van Schouwen

Version control is only useful if everyone uses it. Make sure all team members commit at least every working version of their code to the server so it's easier to keep track of who did what, and if necessary, which version to revert to. Print statements can be very useful for debugging but should be used with caution and should also be easily turned off. It can greatly slow down the interpreter if a string was printed out for every pixel of an image.

Also, it was very beneficial, especially during the latter parts of the coding, to have one person typing and another looking on to catch mistakes. Moreover, a lot of simple mistakes were caught simply by having one person attempt to explain what he or she was doing to another.

Appendix

Interpreter Front End

Grammer.g

```
/*
* IMLGrammar.g
*
*/

/*
* Parser
* Written by Steven C.  Zach S.
*/
```

```
class IMLParser extends Parser;
options { buildAST = true; }
tokens {
      NEG; POS; ARGS; ARG; FUNCDEF; BLOCK; DECL; RET; FOREXPR; WHILEEXPR; FUNCALL; ELEM; FLOAT;
BRACKPAIR;
}

program:
      (function_definition)+
;

function_definition:
      "function"! ID LPAREN! (type_expression_list) RPAREN! statement_block
      { #function_definition = #([FUNCDEF, "function_definition"], #function_definition); }
;

statement:
      statement_block
      | selection_statement
      | iteration_statement
      | expression_statement
      | declaration_statement
      | break_statement
      | return_statement
      | io_statement
      ;

statement_block:
      LBRACE! compound_statement RBRACE!
      { #statement_block = #([BLOCK, "block"], #statement_block); }
      ;

compound_statement:
      (statement)*
      ;

type_specifier:
      "Int" | "Float" | "String" | "Pixel" | "Image"
      ;

selection_statement:
      "if"^ LPAREN! expression RPAREN! statement_block (options {greedy = true;} : "else"!
statement_block)?
      ;

iteration_statement:
      "while"^ LPAREN! while_expression RPAREN! statement_block
      | "for"^ LPAREN! for_expression SEMI! for_expression SEMI! for_expression RPAREN!
statement_block
      ;

while_expression:
      expression
      { #while_expression = #([WHILEEXPR, "while_expression"], #while_expression); }
;

for_expression :

      (expression)?
      { #for_expression = #([FOREXPR, "for_expression"], #for_expression); }
;

break_statement:
      "break"^ SEMI!
      ;

return_statement:
      "return"^ expression SEMI!
      ;
```

```
io_statement:
      "print"^ LPAREN! expression RPAREN! SEMI!
      ;

size_indicator:
      (LBRACK! (expression)? RBRACK!)+
      ;

type_expression:
      type_specifier ID (dimensionality)?
      { #type_expression = #([ARG, "arg"], #type_expression); }
      ;

dimensionality:
    (brackpair)+
    ;

brackpair:
    LBRACK! RBRACK!
     { #brackpair = #([BRACKPAIR, "brackpair"]); }
    ;

type_expression_list:
      (type_expression (COMMA! type_expression)*
      | /*no argument list in this case but AST node is still built */)
      { #type_expression_list = #([ARGS, "args"], #type_expression_list); }
      ;

declaration_statement:
      type_specifier ID (size_indicator)? SEMI!
      { #declaration_statement = #([DECL, "variable_declaration"], #declaration_statement); }
      ;

expression_statement:
      (expression)? SEMI!
      ;

expression:
      fileio_expression
      ;

fileio_expression:
      assignment_expression (("save"^ | "open"^) assignment_expression)?
      ;

assignment_expression: /* right associative here */
      logical_or_expression (ASSIGN^ assignment_expression)?
      ;

logical_or_expression:
      logical_and_expression (OR^ logical_and_expression)*
      ;

logical_and_expression:
      equality_expression (AND^ equality_expression)*
      ;

equality_expression:
      relational_expression ((EQ^|NE^) relational_expression)*
      ;

relational_expression:
      additive_expression ((LT^|GT^|GE^|LE^) additive_expression)*
      ;

additive_expression:
      multiplicative_expression ((PLUS^|MINUS^) multiplicative_expression)*
      ;

multiplicative_expression:
      unary_expression ((MULT^|DIV^|MOD^) unary_expression)*
      ;

unary_expression:
      NOT^ unary_expression
      | PLUS^ unary_expression { #unary_expression.setType(POS); }
      | MINUS^ unary_expression { #unary_expression.setType(NEG); }
      | ("length"^ | "rows"^ | "cols"^)? channel_expression
      ;

channel_expression:
      ("red"^ | "green"^ | "blue"^ | "alpha"^)? postfix_expression
      ;

argument_expression_list:
      COMMA! expression (argument_expression_list)?
      ;

postfix_expression:
      ( ID | constant_expression )
            ( (size_indicator { #postfix_expression = #([ELEM, "array_element"],
#postfix_expression); })
             |  LPAREN!
                 (RPAREN! { #postfix_expression = #([FUNCALL, "function_call"],
#postfix_expression); }
                 | expression (argument_expression_list)? { #postfix_expression = #([FUNCALL,
"function_call"], #postfix_expression); } RPAREN!
                 )
            )?
      | LPAREN! expression RPAREN!
      ;

constant_expression:
      string_constant
      | numeric_constant
      ;

string_constant:
      LITERAL
      ;

numeric_constant:
      INTEGER | FLOAT
      ;



/*
* Lexer
* Written by Cindy L.  Eric H.  Steven C.
*/

class IMLLexer extends Lexer;
options {
    k = 2;
    charVocabulary = '\u0000'..'\u007F';
}

DOT: '.';
AND : "&&" ;
LE : "<=" ;
SEMI : ';' ;
COMMA : ',' ;
OR : "||" ;
GT : '>' ;
LPAREN : '(' ;
ASSIGN : '=' ;
GE : ">=" ;
RPAREN : ')' ;
EQ : "==" ;
LBRACE : '{' ;
PLUS : '+' ;
NOT : '!' ;
RBRACE : '}' ;
MINUS : '-';
NE : "!=" ;
LBRACK : '[' ;
MOD : '%' ;
MULT : '*' ;
LT : '<' ;
RBRACK : ']' ;
DIV : '/' ;
QUOTE : '"' ;

protected DIGIT : ('0'..'9') ;
protected MINISC : 'a'..'z' ;
protected MAJUSC : 'A'..'Z' ;

INTEGER : (DIGIT)+ (DOT (DIGIT)+ { $setType(FLOAT); })? ;
ID : ('_' | MINISC | MAJUSC) ('_' | MINISC | MAJUSC | DIGIT)* ;
LITERAL : QUOTE! ( ESCAPE | ~('\\'|'"') )* QUOTE!;
ESCAPE : '\\'
      ( 'n' { $setText("\n"); }
      | 't' { $setText("\t"); }
      | 'b' { $setText("\b"); }
      | 'f' { $setText("\f"); }
      | '\"' { $setText("\""); }
      | '\'' { $setText("\'"); }
      | '\\' { $setText("\\"); } )
;
WHITESPACE : (' ' | '\t' | '\n' { newline(); } )+ { $setType(Token.SKIP); } ;
LINECMNT : "//" (~'\n')* { $setType(Token.SKIP); } ;
FULLCMNT : "/*" ( options { greedy = false; } : . )* "*/" { $setType(Token.SKIP); } ;
```

Tree Walker

```
/*
* IMLWalker.g
* Written by Steven C.  Zach S.  Cindy L.
*/


{ import java.util.Stack;
  import java.util.LinkedList;
  import java.util.Iterator;
  import java.util.ArrayList;
  import antlr.CommonAST;
   }

class IMLTreeParser extends TreeParser;


options { importVocab = IMLParser; }

{
  public static void debug(String d) {
      //System.out.println(d);
  }

      // Single symbol table stores stacks of hash tables; functions are
      // defined in here alongside variables.
      SymbolTable symbolTable = new SymbolTable();

      // We add a record to the activation stack each time a function is
      // actually called. Each record points to a subtree where we resume
      // execution.
      Stack<ActivationRecord> activationStack = new Stack();

      Symbol exprResult;
}
```

```
// At press time, there are no global variables or declarations -- just a
// series of functions, one of which is main().
program throws Exception :
      (functions)*
;


functions throws Exception {
            ArrayList<TypePair> alist;
    }:
      #(FUNCDEF ID alist=arguments body:.) {
            // Note where this function lies in the AST, then add a
            // reference to that position to the symbol table.
            if (#ID.getText().equals("main")) {
              Symbol ret = new Symbol(Symbol.SymType.INT);

              ActivationRecord outer = new ActivationRecord(ret, new ArrayList());
              activationStack.push(outer);
                  block(body, false);
            } else {
              debug("Storing function: " + #ID.getText());
                  Symbol definition = new Symbol(Symbol.SymType.FUNCDEF,
                                          body);
                  symbolTable.define(#ID.getText(), definition);
                  definition.setArgumentList(alist);
            }
      }
;

return_type returns [Symbol.SymType r = Symbol.SymType.INT] :
      #(RET r=data_type) // add array support
;

// Return the data type as a member of our SymType enum.
data_type returns [Symbol.SymType r = Symbol.SymType.INT] :
        ("Int"    { r = Symbol.SymType.INT; })
      | ("Float"  { r = Symbol.SymType.FLOAT; })
      | ("String" { r = Symbol.SymType.STRING; })
      | ("Pixel"  { r = Symbol.SymType.PIXEL; })
      | ("Image"  { r = Symbol.SymType.IMAGE; })
;

arguments returns[ArrayList<TypePair> alist = new ArrayList<TypePair>()] { TypePair a; }:
      #(ARGS { debug("arg list"); } (#(ARG a=argument { alist.add(a); debug("adding: " + a); }))*
)
;

argument returns[TypePair name = null] :
      {int count = 0; Symbol.SymType type; }
      type=data_type ID (. {count++;})* // add array support (for arguments its [][]...)
      { name = new TypePair(#ID.getText(), type, count); }
;

block [boolean useRecord] returns [Result r = new Result()] throws Exception :
      {

        debug("block");
            r.type = Result.ResultType.IS_NEITHER;
            Symbol.SymType type;   // For storage

            symbolTable.enterScope();

            if (useRecord) {
              debug("checking activation rec'd");
              ActivationRecord rec = activationStack.peek();

              // Define each argument from rec.
              Iterator i = rec.args.iterator();
              while (i.hasNext()) {
                  Argument s = (Argument) i.next();
```

```
                 //debug("defining " + s.name + " as " + s.value.toString() +
                 //       " in local scope");
                 symbolTable.define(s.name, s.value);
               }
             debug("stop args");
           }
       }
       #(BLOCK (#(DECL type=data_type ID
                             { LinkedList specs = new LinkedList<Integer>(); }
                             (size_spec:. {
                                   Symbol s = expression(size_spec);
                                   specs.add(s.toInt());
                             })*)
             {
               debug("Declaration: " + #ID.getText());
                 // define the identifier from DECL
                 // determine the type from "specs"
                 Symbol s;
                 if (specs.size() > 0) s = Symbol.initArray(type, specs);
                 else s = new Symbol(type);
                 symbolTable.define(#ID.getText(), s);
             }
             | stmt:. {
               debug("stmt");
               if (!r.isReturn() && !r.isBreak()) {
                 r = statement(stmt);
               }
             })*)
       {
         debug("leaving block");
       if (useRecord) {
             debug("removing activation rec'd");
             activationStack.pop(); /* our record is no longer useful */
       }
         symbolTable.leaveScope();
       }
;

statement returns [Result r = new Result()] throws Exception { int v; Symbol e;
debug("statement");} :
       r = block [false]
       |  expression
       | #("print" {debug("not stuck");} e=expression) {
                 if (e != null)
                       //System.out.println((e.getClass().toString()) + " " + e.toString());
                       System.out.println(e.toString());
                 debug("print called.");
             }
       | "break" { debug("BREAKING"); r.type = Result.ResultType.IS_BREAK; }
       | #("return" e=expression ) {
             ActivationRecord rec = activationStack.peek();
             rec.retval.copy(e);
             r.type = Result.ResultType.IS_RETURN;
       }
       | #("if" i_qualifier:. if_block:. (else_block:.)?)
             {
                 debug("IF statement");
                 if (Ops.truth(expression(i_qualifier)))
                       { r = block(if_block, false); }
                 else if (else_block != null)
                       { r = block(else_block, false); }

             }
       | #("while" #(WHILEEXPR w_qualifier:.) #(BLOCK (w_loop:.)?))
             {
                 while (#w_qualifier == null || Ops.truth(expression(w_qualifier)))
                 {
                       Result r2;
                       if (w_loop != null) {
                             r2 = block(#BLOCK, false);
                             if (r2.isBreak()) break;
```

```
                              if (r2.isReturn()) {
                                    r.type = Result.ResultType.IS_RETURN;
                                    return r;
                              }
                        }
                  }

            }
      | #("for" #(FOREXPR (f_initializer:.)?) #(FOREXPR (f_qualifier:.)?)
                #(FOREXPR (f_iterator:.)?) #(BLOCK (f_loop:.)?))
            {
                  if (#f_initializer != null) expression(f_initializer);
                  while (#f_qualifier == null || Ops.truth(expression(f_qualifier)))
                  {
                        Result r2;
                        if (f_loop != null) {  // don't execute empty blocks
                              r2 = block(#BLOCK, false);
                              if (r2.isBreak()) break;
                              if (r2.isReturn()) {
                                    r.type = Result.ResultType.IS_RETURN;
                                    return r;
                              }
                        }
                        if (#f_iterator != null) expression(f_iterator);
                  }
            }
;

expression returns [Symbol r = new Symbol(Symbol.SymType.INT)] throws Exception { Symbol a, b,
sym; } :
        #(PLUS     a=expression b=expression)  { debug("plus"); r = Ops.plus(a, b); debug("after
plus"); }
      | #(MINUS    a=expression b=expression)  { r = Ops.minus(a, b); }
      | #(MULT     a=expression b=expression)  { r = Ops.multiply(a, b); }
      | #(DIV      a=expression b=expression)  { r = Ops.divide(a, b); }
      | #(MOD      a=expression b=expression)  { r = Ops.modulus(a, b); }
      | #(NEG      a=expression)              { r = Ops.negation(a); }
      | #(POS      a=expression)         { r = Ops.pos(a); }
      | #(EQ       a=expression b=expression)  { r = Ops.equality(a, b); }
      | #(NE       a=expression b=expression) { r = Ops.not(Ops.equality(a, b)); }
      | #(LT       a=expression b=expression) { r = Ops.lessthan(a, b); }
      | #(GT       a=expression b=expression) { r = Ops.greaterthan(a,b); }
      | #(LE       a=expression b=expression) { r = Ops.lessequal(a,b); }
      | #(GE       a=expression b=expression) { r = Ops.greaterequal(a,b); }
      | #(NOT      a=expression)              { r = Ops.not(a); }
      | #(OR       a=expression b=expression) { r = Ops.or(a, b); }
      | #(AND      a=expression b=expression) { r = Ops.and(a, b); }
      | #("red"    a=expression)         { r = PixelOps.red(a); }
      | #("blue"   a=expression)         { r = PixelOps.blue(a); }
      | #("green"  a=expression)         { r = PixelOps.green(a); }
      | #("alpha"  a=expression)         { r = PixelOps.alpha(a); }
      | #("length" a=expression)         { r = ArrayOps.length(a); }
      | #("rows"   a=expression)         { r = ArrayOps.rows(a); }
      | #("cols"   a=expression)         { r = ArrayOps.cols(a); }
      | #("save"   a=expression b=expression)  { IMLImageOps.save(a,b); r=a; }
      | #("open"   a=expression b=expression) { IMLImageOps.open(a,b); r = b;     }
      | i:INTEGER { r = new Symbol(Symbol.SymType.INT, new Integer(#i.getText())); }
      | f:FLOAT   { r = new Symbol(Symbol.SymType.FLOAT, new Float(#f.getText())); }
      | s:LITERAL { r = new Symbol(Symbol.SymType.STRING,
                                 new String(#s.getText())); }
    | #(ELEM ID { r = (Symbol) symbolTable.identifier(#ID.getText()); } (index:. {
      Symbol pos = expression(index);
      debug("Indexing to: " + pos.toString());
      r = ArrayOps.at(r, (Integer) pos.getValue());
      })*)
      | #(FUNCALL ID {
            sym = (Symbol) symbolTable.identifier(#ID.getText());
            if (!sym.isFunction())
                  throw new Exception("Function call to non-function object " +
                                               #ID.getText() + "!");
            ArrayList<Argument> args = new ArrayList<Argument>();

            ActivationRecord rec = new ActivationRecord(r, args);
            activationStack.push(rec);

            Iterator it = sym.argumentList().iterator();
      } (a=expression {
                  // foreach arg
                  if (!it.hasNext())
                        throw new Exception("Too many parameters!");
                  // Assign local parameters to the argument pairs
                  TypePair pair = (TypePair) it.next();
                  String name = pair.name;
                  debug("Param: " + name);
                  // force *rough* type checking using assign()
                  // TODO dimensionality
                  Symbol pass = new Symbol((pair.dimensionality > 0) ?
                        Symbol.SymType.ARRAY : pair.type);
                  Ops.assign(pass, a);
                  Argument arg = new Argument(name, pass);
                  args.add(arg);
             })*
        {
            // after arglist
            // Jump to the tree position indicated by sym
            AST block = (AST) sym.getValue();
            block(block, true);
        } )
      | #(ASSIGN b=expression a=expression) {
            debug("b=" + b.toString());
            debug("a=" + a.toString());
            Ops.assign(b, a);
            debug("after assign: b=" + b.toString());
            r = b;
        }
      | identifier:ID {
        r = (Symbol) symbolTable.identifier(identifier.getText());
        //debug("at id level: " + r.toString());
      }
;
```

Interpreter Back End

Formal Argument Object

```
/*
* TypePair.java
* Written by Zach S.
*/

public class TypePair {
      public TypePair(String n, Symbol.SymType t, int dim) {
            name = n;
            type = t;
            dimensionality = dim;
      }

      public String name;
      public Symbol.SymType type;
      public int dimensionality;
}
```

Actual Argument Object

```
/*
* Argument.java
* Written by Zach S.
*/

public class Argument {
       public Argument(String n, Symbol v) {
              name = n;
              value = v;
       }

       public String name;
       public Symbol value;
}
```

Function Exit Type

```
/*
* Result.java
* Written by Zach S.
*/

public class Result {
       public Result() {
              type = ResultType.IS_NEITHER;
       }

       public enum ResultType { IS_BREAK, IS_RETURN, IS_NEITHER };
       public ResultType type;

       public Boolean isReturn() { return (type == ResultType.IS_RETURN); }
       public Boolean isBreak() { return (type == ResultType.IS_BREAK); }
}
```

Activation Record

```
/*
* ActivationRecord.java
* Written by Zach S.
*/

import java.util.*;

public class ActivationRecord {
       public ActivationRecord(Symbol r, ArrayList<Argument> a) {
              retval = r;
              args = a;
       }

       public Symbol retval;
       public ArrayList<Argument> args;
}
```

Symbol Table

```
/*
* SymbolTable.java
* Written by Zach S.  Steven C.
*/
```

```
import java.util.*;

/* Stack of hash tables for implementing simple scope. */
public class SymbolTable extends Stack {
        /* Initializes the global scope. */
        public SymbolTable() { enterScope(); }

        /* Start a new, empty scope. */
        public void enterScope() {
                Hashtable tbl = new Hashtable();
                push(tbl);
        }

        /* Destroy the topmost scope layer. */
        public void leaveScope() { pop(); }

        /* Get the value of an identifier by cascading through the stack.
         * Throws an exception if the identifier is not found. */
        public Object identifier(String id) throws NoSuchFieldException {
                Stack search = (Stack) clone();
                while (!search.empty()) {
                        Hashtable scope = (Hashtable) search.pop();
                        if (scope.containsKey(id)) return scope.get(id);
                }
                /* Didn't find it... */
                throw new NoSuchFieldException("I couldn't find " + id);
        }

        /* Add a new variable -- puts it in the topmost scope. */
        public void define(String id, Object obj) throws RuntimeException {
                Hashtable scope = (Hashtable) peek();
                if (scope.containsKey(id))
                        throw new RuntimeException("Can't redefine " + id.toString() + "!");
                scope.put(id, obj);
        }
}
```

Variable Bindings

```
/*
* Symbol.java
* Written by Cindy L.  Eric H.  Steven C.  Zach S.
*/

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

import antlr.collections.AST;

public class Symbol
{
        public enum SymType { INT, FLOAT, STRING, PIXEL, IMAGE, ARRAY, FUNCDEF };

        private SymType type;
        private Object value;
        public char channel = 0;
        public int imageType = 0;


        public Symbol (Symbol.SymType type) {
                this.type = type;
                if (type == SymType.INT) value = new Integer(0);
                if (type == SymType.FLOAT) value = new Float(0.0f);
                if (type == SymType.STRING) value = new String("");
                if (type == SymType.PIXEL) value = new Pixel();
```

```
              if (type == SymType.ARRAY) value = new ArrayList<Symbol>();
              if (type == SymType.IMAGE) value = new ArrayList<Symbol>();
       }

       public Symbol (SymType type, Object value) {
              this.type = type;
    try {
       this.setValue(value);
    } catch (Exception e) {
       e.printStackTrace();
       System.err.println("Uh-oh"); }
       }

       public Symbol (Boolean value) {
    this.type = SymType.INT;
    Integer val = (value == Boolean.TRUE ? 1 : 0);
    this.value = val;
  }

  public static Symbol initArray(Symbol.SymType type, LinkedList<Integer> specs) {
       if (specs.size() == 0) return new Symbol(type);
       Symbol s = new Symbol(SymType.ARRAY);

       int len = ((Integer) specs.peek()).intValue();
       LinkedList<Integer> newspecs = new LinkedList<Integer>();
       for (int i = 1; i < specs.size(); i++)
              newspecs.add(i - 1, specs.get(i));

       for (int i = 0; i < len; i++)
              /* ignore this stupid warning: */
              ((ArrayList) s.value).add(i, initArray(type, newspecs));
       return s;
  }

       public void setRef(Integer val) {
              value = val;
              //ref = true;
       }

       public void setRefArray(ArrayList<Symbol> val) {
              value = val;
       }

       public void copy(Symbol s) {
         type = s.type;
         value = s.value;
       }

       public void setValue(Object value) throws NoSuchMethodException {
              if (value instanceof Boolean) {
                     Integer val = (value == Boolean.TRUE ? 1 : 0);
                     this.value = val;
              }
              else if (type == SymType.INT) {
                     Integer val = (Integer)value;
                     this.value = val;
              }
              else if (type == SymType.FLOAT) {
                     Float val = (Float)value;
                     this.value = val;
              }
              else if (type == SymType.STRING) {
                     String val = (String)value;
                     this.value = val;
              }
              else if (type == SymType.PIXEL) {
                     if (channel != 0) {
                            setChannel(value);
                            return;
                     }
                     Pixel val = new Pixel((Pixel) value);
```

```
			this.value = val;
		}
		else if (type == SymType.ARRAY || type == SymType.IMAGE) {
			SymType preserve = findType();

			/* CAREFUL! You don't want to call deepCopy() with type Image, or
			 * stupid casting disasters will happen. */
			if (preserve == SymType.IMAGE) preserve = SymType.PIXEL;

			this.value = new ArrayList<Symbol>();
			deepCopy(preserve, (ArrayList<Symbol>) value);
		}
		else if (type == SymType.FUNCDEF) {
			AST val = (AST) value;
			this.value = val;
		}
		else throw new NoSuchMethodException("Object type not set!");
	}

	public SymType findType()
	throws NoSuchMethodException {
		Symbol s = this;
		while (s.type == SymType.ARRAY) s = ArrayOps.at(s, 0);
		return s.type;
	}

	public int findDims()
	throws NoSuchMethodException {
		Symbol s = this;
		int c = 0;
		while (s.type == SymType.ARRAY) { s = ArrayOps.at(s, 0); c++; }
		return c;
	}

	public void deepCopy(Symbol.SymType type, ArrayList<Symbol> value)
	throws NoSuchMethodException {
		ArrayList<Symbol> myArrayList = (ArrayList<Symbol>) this.value;
		for (int i=0; i<value.size(); i++) {
			Symbol v = (Symbol) value.get(i);
			Symbol s = new Symbol((v.getType() == SymType.ARRAY ||
					v.getType() == SymType.IMAGE) ? v.getType() : type);

			// Don't pass in a null array
			if (s.getType() == Symbol.SymType.ARRAY) {
				LinkedList<Integer> specs = new LinkedList<Integer>();
				specs.add(0, ((ArrayList) v.getValue()).size());
				s = initArray(type, specs);
			}
			Ops.assign(s, v);
			myArrayList.add(i, s);
		}
	}

	public void setChannel(Object value) throws NoSuchMethodException {
		Pixel p = (Pixel) this.value;
		Integer i = (Integer) value;

		if (i < 0 || i > 255)
		throw new NoSuchMethodException("Invalid channel value");

		switch(channel) {
		case 'r': p.red = i; break;
		case 'g': p.green = i; break;
		case 'b': p.blue = i; break;
		case 'a': p.alpha = i; break;
		default: throw new NoSuchMethodException("Invalid channel");
		}
	}

	/* Time-saver constructors for anonymous symbols. */
	public Symbol (Integer value) {
		this.type = SymType.INT;
		this.value = value;
	}
	public Symbol (String value) {
		this.type = SymType.STRING;
		this.value = value;
	}
	public Symbol (Float value) {
		this.type = SymType.FLOAT;
		this.value = value;
	}
	public Symbol (Pixel value) {
		this.type = SymType.PIXEL;
		this.value = value;
	}

	public ArrayList<TypePair> argumentList() {
		return argList;
	}

	public void setArgumentList(ArrayList<TypePair> list) {
	  argList = list;
	}

	public SymType getType() { return type; }
	public Object getValue() {
		if (channel != 0) return getChannel();
		return value;
	}

	public Integer getChannel() {
		Pixel p = (Pixel) value;
		switch (channel) {
		case 'r': return new Integer(p.red);
		case 'b': return new Integer(p.blue);
		case 'g': return new Integer(p.green);
		case 'a': return new Integer(p.alpha);
		}
		return null;
	}

	public Boolean isInt()		{ return this.type == SymType.INT; }
	public Boolean isFloat()	{ return this.type == SymType.FLOAT; }
	public Boolean isString()	{ return this.type == SymType.STRING; }
	public Boolean isPixel()	{ return this.type == SymType.PIXEL; }
	public Boolean isImage()	{ return this.type == SymType.IMAGE; }
	public Boolean isArray()    { return this.type == SymType.ARRAY; }
	public Boolean isFunction() { return this.type == SymType.FUNCDEF; }

	public String toString() {
		if (type == SymType.STRING) return (String) value;
		return getValue().toString();
	}
	public Float toFloat() {
		return ((Float) value).floatValue();
	}
	public Integer toInt() {
		return ((Integer) value).intValue();
	}

	ArrayList<TypePair> argList;
}
```

Mathematical and Logical Functions

```
/*
* Ops.java
* Written by Eric H.  Steven C.  Zach S.
```

```
*/

public class Ops {
      public static Boolean truth(int param) { return (param != 0); }

      public static Boolean truth(Symbol param) {
            if (param.isInt() || param.isFloat())
                  return param.toInt() != 0;
            if (param.isString()) return !param.toString().equals("");
            return param != null;
      }

      public static void typeCheck(Symbol one) throws NoSuchMethodException {
            if(one.getType() != Symbol.SymType.INT &&
               one.getType() != Symbol.SymType.FLOAT)
                  invalidArithmetic();
      }

      private static void invalidArithmetic() throws NoSuchMethodException {
            throw new NoSuchMethodException("Illegal arithmetic!");
      }

      private static void nullSymbol() throws NoSuchMethodException {
            throw new NoSuchMethodException(
                        "Illegal operation on null variable!");
      }

      public static Symbol plus(Symbol one, Symbol two)
      throws NoSuchMethodException {
            one = pixelCheck(one);
            two = pixelCheck(two);
            if (one == null || two == null) { nullSymbol(); return null; }

            if (one.isInt() && two.isInt())
                  return new Symbol(two.toInt() + one.toInt());

            if (one.isFloat() && two.isInt())
                  return new Symbol(two.toInt() + one.toFloat());

            if (one.isInt() && two.isFloat())
                  return new Symbol(two.toFloat() + one.toInt());

            if (one.isFloat() && two.isFloat())
                  return new Symbol(two.toFloat() + one.toFloat());

            if (one.isInt() && two.isString())
                  return new Symbol(one.toInt().toString()
                              + (String) two.getValue());

            if (one.isFloat() && two.isString())
                  return new Symbol(one.toFloat().toString() +
                               (String) two.getValue());

            if (one.isString()) {  // concatenation
                  String L = (String) one.getValue();
                  if (two.isInt())
                        return new Symbol((L) +
                              (two.toInt().toString()));
                  if (two.isFloat())
                        return new Symbol((L) +
                              (two.toFloat().toString()));
                  if (two.isString())
                        return new Symbol((L) + (String)two.getValue());
            }

            invalidArithmetic();
            return null;
      }

      public static Symbol minus(Symbol two, Symbol one)
```

```
throws NoSuchMethodException {
      if (one == null || two == null) { nullSymbol(); return null; }
      one = pixelCheck(one);
      two = pixelCheck(two);
      if (one.isInt() && two.isInt())
            return new Symbol(two.toInt() - one.toInt());
      if (one.isFloat() && two.isInt())
            return new Symbol(two.toInt() - one.toFloat());
      if (one.isInt() && two.isFloat())
            return new Symbol(two.toFloat() - one.toInt());
      if (one.isFloat() && two.isFloat())
            return new Symbol(two.toFloat() - one.toFloat());

      invalidArithmetic();
      return null;
}

public static Symbol multiply(Symbol one, Symbol two)
throws NoSuchMethodException {
      if (one == null || two == null) { nullSymbol(); return null; }
      one = pixelCheck(one);
      two = pixelCheck(two);
      if (one.isInt() && two.isInt())
            return new Symbol(one.toInt() * two.toInt());
      if (one.isFloat() && two.isInt())
            return new Symbol(Symbol.SymType.FLOAT,
                          one.toFloat() * two.toInt());
      if (one.isInt() && two.isFloat())
            return new Symbol(Symbol.SymType.FLOAT,
                          one.toInt() * two.toFloat());
      if (one.isFloat() && two.isFloat())
            return new Symbol(one.toFloat() * two.toFloat());


      invalidArithmetic();
      return null;
}

public static Symbol divide(Symbol one, Symbol two)
throws NoSuchMethodException {
      if (one == null || two == null) { nullSymbol(); return null; }
      one = pixelCheck(one);
      two = pixelCheck(two);
      if (one.isInt() && two.isInt())
            return new Symbol(one.toInt() / two.toInt());
      if (one.isFloat() && two.isInt())
            return new Symbol(Symbol.SymType.FLOAT,
                          one.toFloat() / two.toInt());
      if (one.isInt() && two.isFloat())
            return new Symbol(Symbol.SymType.FLOAT,
                          one.toInt() / two.toFloat());

      if (one.isFloat() && two.isFloat())
            return new Symbol(one.toFloat() / two.toFloat());

      invalidArithmetic();
      return null;
}

public static Symbol modulus(Symbol one, Symbol two)
throws NoSuchMethodException {
      if (one == null || two == null) { nullSymbol(); return null; }
      one = pixelCheck(one);
      two = pixelCheck(two);
      if (one.isInt() && two.isInt())
            return new Symbol(one.toInt() % two.toInt());

      invalidArithmetic();
      return null;

}
```

```
public static Symbol negation(Symbol one) throws NoSuchMethodException {
      if (one == null) { nullSymbol(); return null; }
      one = pixelCheck(one);
      if (one.isInt()) return new Symbol(-1 * one.toInt());
      if (one.isFloat()) return new Symbol(-1 * one.toFloat());

      invalidArithmetic();
      return null;
}

public static Symbol pos(Symbol one) throws NoSuchMethodException {
      if (one == null) { nullSymbol(); return null; }
      one = pixelCheck(one);
      if (one.isInt()) return new Symbol(one.toInt());
      if (one.isFloat()) return new Symbol(one.toFloat());

      invalidArithmetic();
      return null;
}

public static boolean iseq(Symbol one, Symbol two)
throws NoSuchMethodException {
      if (one == null && two == null) { return true; }
      if (one == null || two == null) { return false; }
      one = pixelCheck(one);
      two = pixelCheck(two);

      if (one.isInt() && two.isInt())
            return (one.toInt().intValue() == two.toInt().intValue());
      if (one.isInt() && two.isFloat())
            return (one.toInt().floatValue() == two.toFloat().floatValue());
      if (one.isFloat() && two.isInt())
            return (one.toFloat().floatValue() == two.toInt().floatValue());
      if (one.isFloat() && two.isFloat())
            return (one.toFloat().floatValue() == two.toFloat().floatValue());
      if (one.isPixel() && two.isPixel())
            return ((Pixel) one.getValue()).toRGBA() ==
                   ((Pixel) two.getValue()).toRGBA();

      invalidArithmetic();
      return false;
}

public static boolean isne(Symbol one, Symbol two)
throws NoSuchMethodException {
      return !iseq(one, two);
}

public static boolean islt(Symbol one, Symbol two)
throws NoSuchMethodException {
      if (one == null && two == null) { nullSymbol(); return true; }
      if (one == null || two == null) { nullSymbol(); return false; }
      one = pixelCheck(one);
      two = pixelCheck(two);
      if (one.isInt() && two.isInt())
            return one.toInt() < two.toInt();
      if (one.isInt() && two.isFloat())
            return one.toInt() < two.toFloat();
      if (one.isFloat() && two.isInt())
            return one.toFloat() < two.toInt();
      if (one.isFloat() && two.isFloat())
            return one.toFloat() < two.toFloat();

      invalidArithmetic();
      return false;
}

public static boolean isgt(Symbol one, Symbol two)
throws NoSuchMethodException {
      return !islt(one, two) && !iseq(one, two);
```

```
}

public static boolean isge(Symbol one, Symbol two)
throws NoSuchMethodException {
      return !islt(one, two);
}

public static boolean isle(Symbol one, Symbol two)
throws NoSuchMethodException {
      return islt(one, two) || iseq(one, two);
}

public static Symbol equality(Symbol one, Symbol two)
throws NoSuchMethodException {
      return new Symbol(iseq(one, two));
}

public static Symbol lessthan(Symbol one, Symbol two)
throws NoSuchMethodException {
      return new Symbol(islt(one,two));
}

public static Symbol greaterthan(Symbol one, Symbol two)
throws NoSuchMethodException {
      return new Symbol(isgt(one,two));
}

public static Symbol greaterequal(Symbol one, Symbol two)
throws NoSuchMethodException {
      return new Symbol(isge(one,two));
}

public static Symbol lessequal(Symbol one, Symbol two)
throws NoSuchMethodException {
      return new Symbol(isle(one,two));
}

public static Symbol not(Symbol one) throws NoSuchMethodException {
      boolean result = false;
      one = pixelCheck(one);
      result = result || (one.isInt() && one.toInt() == 0);
      result = result || (one.isFloat() && one.toFloat() == 0);
      result = result || (one.isString() &&
                      one.toString().equals(""));
      return new Symbol(result);
}

/* if s has a channel op, replace it with the appropriate int. */
public static Symbol pixelCheck(Symbol s) {
      if (s.channel != 0) {
            Integer v = (Integer) s.getValue();
            return new Symbol(v);
      }
      return s;
}

public static Symbol or(Symbol one, Symbol two)
throws NoSuchMethodException {
      return new Symbol(truth(one) || truth(two));
}

public static Symbol and(Symbol one, Symbol two)
throws NoSuchMethodException {
      return new Symbol(truth(one) && truth(two));
}

public static void assign(Symbol lhs, Symbol rhs)
throws NoSuchMethodException {
      if (lhs.getType() == rhs.getType()) {
            lhs.setValue(rhs.getValue());
            return;
            }
            if (lhs.isInt() && rhs.isFloat()) {
                  lhs.setValue(new Integer(rhs.toFloat().intValue()));
                  return;
            }

            if (lhs.isFloat() && rhs.isInt()) {
                  lhs.setValue(new Float(rhs.toInt()));
                  return;
            }

            if (lhs.isString()) {
                  lhs.setValue(new String(rhs.toString()));
                  return;
            }

            if (lhs.isPixel() && rhs.isInt() && lhs.channel != 0) {
                  lhs.setValue(rhs.getValue());
                  return;
            }

            if (lhs.isInt() && rhs.isPixel() && rhs.channel != 0) {
                  lhs.setValue(rhs.getValue());
                  return;
            }

            if (lhs.isPixel() && rhs.isFloat() && lhs.channel != 0) {
                  lhs.setValue(new Integer(rhs.toFloat().intValue()));
                  return;
            }

            if (lhs.isFloat() && rhs.isPixel() && rhs.channel != 0) {
                  lhs.setValue((Float) rhs.getValue());
                  return;
            }

            if (lhs.isImage() && rhs.isArray()) {
                  if (rhs.findDims() == 2 && rhs.findType() == Symbol.SymType.PIXEL) {
                        lhs.setValue(rhs.getValue());
                        return;
                  }
            }

            if (lhs.isArray() && rhs.isImage()) {
                  lhs.setValue(rhs.getValue());
                  return;
            }

            throw new NoSuchMethodException("Illegal cast from " +
                  rhs.getType() + " to " + lhs.getType() + "!");
      }
}
```

Array Functions

```
/*
* ArrayOps.java
* Written by Cindy L.
*/

import java.util.ArrayList;

public class ArrayOps {
        public static Symbol length(Symbol ar)
        throws NoSuchMethodException {
                 if (ar.isArray())
                         return new Symbol(new Integer(((ArrayList) ar.getValue()).size()));
                 if (ar.isString())
                        return new Symbol(new Integer(((String) ar.getValue()).length()));
                 throw new NoSuchMethodException("Attempt to index non-array obj!");
        }
        public static Symbol cols(Symbol ar)
        throws NoSuchMethodException {
               requireImage(ar);
               return new Symbol(new Integer(((ArrayList) ar.getValue()).size()));
        }
        public static Symbol rows(Symbol ar)
        throws NoSuchMethodException {
            requireImage(ar);
            Symbol col = (Symbol) ((ArrayList) ar.getValue()).get(0);
            return new Symbol(new Integer(((ArrayList) col.getValue()).size()));
        }

        public static Symbol at(Symbol ar, int pos)
        throws NoSuchMethodException {
               if (ar.isArray() || ar.isImage())
                      return (Symbol) (((ArrayList) ar.getValue()).get(pos));
               if (ar.isString())
                      return new Symbol("" + ((String) ar.getValue()).charAt(pos));
               throw new NoSuchMethodException("Attempt to index non-array obj!");
        }

        private static void requireImage(Symbol a) throws NoSuchMethodException {
               if (!a.isImage())
                      throw new NoSuchMethodException("Attempt to use image ops on non-image");
        }
}
```

Image Functions

```
/*
* IMLImageOps.java
* Written by Cindy L.
*/

/**
 * Static Image Class
 */
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import java.util.ArrayList;
import javax.imageio.*;
import javax.swing.ImageIcon;

/**
 * @author Cindy Liao
 *
 */
public class IMLImageOps { //extends Component {


      public IMLImageOps() {

      }

      public static void open(Symbol lhs, Symbol rhs)
      throws IOException, NullPointerException, NoSuchMethodException {
            if (lhs == null || rhs == null) {
                  throw new NullPointerException("Cannot call open with null Symbol(s).");
            }

            if (!lhs.isString()) {
                  System.err.println("Open failed: " + lhs.toString() + " is not a filename.");
                  return;
            }

            if (!rhs.isImage()) {
                  System.err.println("Open failed: " + rhs.toString() + " is not an Image.");
                  return;
            }

            IMLImage img = loadImage(lhs.toString());
            rhs.imageType = img.imageType;
            rhs.setRefArray(img.getPixels());
      }

      private static IMLImage loadImage(String filename) throws IOException {
            IMLImage img = new IMLImage();
            BufferedImage src_img = ImageIO.read(new File(filename));
            img.imageType = src_img.getType();
            return getPixels(src_img, img);
      }

      private static IMLImage getPixels(BufferedImage src_img, IMLImage img) {

            //Convert the rawImage to numeric pixel representation
            int width = src_img.getWidth();
            int height = src_img.getHeight();

            img.width = width;
            img.height = height;

        int[] pic = new int[width * height];
        int[][]pixels = new int[width][height];

            try{
                  PixelGrabber pgObj = new PixelGrabber(src_img,0,0,width,height,pic,0,width);
              if (pgObj.grabPixels()) {
                  //convert 1D to 2D array of ints
                  for (int x=0; x<width; x++) {
                        for (int y=0; y<height; y++) {
                              pixels[x][y] = pic[y* width +x];
                        }
                  }

                  img.pixels = pixels;
                  return img;

              } else {
                  System.out.println("grabPixels not successful. " + pgObj.getStatus());
                  if ((pgObj.getStatus() & ImageObserver.ERROR) != 0)
                        System.out.println("Error");
                  if ((pgObj.getStatus() & ImageObserver.ABORT) != 0)
                        System.out.println("Abort");
              }
            } catch(InterruptedException e) {
                  System.err.println(e.getMessage());
            }

            return null;
      }



      public static void save(Symbol lhs, Symbol rhs)
      throws IOException, NullPointerException, Exception {
            if (lhs == null || rhs == null) {
                  throw new NullPointerException("Cannot call save with null Symbol(s).");
            }

            if (!rhs.isString()) {
                  System.err.println("Save failed: " + rhs.toString() + " is not a filename.");
                  return;
            }
```

```
            if (!lhs.isImage()) {
                  System.err.println("Save failed: " + lhs.toString() + " is not an Image.");
                  return;
            }

            IMLImage img = new IMLImage(lhs);
            File f = new File(rhs.toString());
            saveImage(f, img);
      }

      private static void saveImage(File f, IMLImage img) throws IOException {
            BufferedImage dest_img = makeImage(img);
            String ext = getExtension(f.getName());
            ImageIO.write(dest_img, ext, f);
      }


      private static BufferedImage makeImage(IMLImage img) {
            int width = img.width;
            int height = img.height;
            int[] pixels1D = new int[width*height];
            int imagetype = img.imageType;
            if (imagetype == 0) imagetype = 5;

            for (int x=0; x<width; x++) {
                  for (int y=0; y<height; y++) {
                        pixels1D[y*width+x] = img.pixels[x][y];
                  }
            }

            Canvas c = new Canvas();
            Image tmp_img = c.createImage(
                  new MemoryImageSource(width, height, pixels1D, 0, width));
            BufferedImage bi = new BufferedImage(width, height, imagetype);
            Graphics2D g = bi.createGraphics();
            g.drawImage(tmp_img, 0, 0, null);
            g.dispose();

            return bi;
      }

      private static String getExtension(String filename) {
            if (filename.contains(".jpg")) return "jpg";
            if (filename.contains(".bmp")) return "bmp";
            if (filename.contains(".png")) return "png";

            // warns user that image is saved a jpg by default
            System.err.println("Format not supported. Saving as jpg instead.");

            return "jpg";
      }
}
```

Pixel Functions

```
/*
* PixelOps.java
* Written by Cindy L.
*/

/* Simple encapsulation functions for picking symbols out of pixels. */

public class PixelOps {
      public static Symbol red(Symbol px)
      throws NoSuchMethodException {
            Symbol s = new Symbol(Symbol.SymType.PIXEL);
            s.copy(px);
            s.channel = 'r';
```

```
        return s;
    }

    public static Symbol blue(Symbol px)
    throws NoSuchMethodException {
        Symbol s = new Symbol(Symbol.SymType.PIXEL);
        s.copy(px);
        s.channel = 'b';
        return s;
    }

    public static Symbol green(Symbol px)
    throws NoSuchMethodException {
        Symbol s = new Symbol(Symbol.SymType.PIXEL);
        s.copy(px);
        s.channel = 'g';
        return s;
    }

    public static Symbol alpha(Symbol px)
    throws NoSuchMethodException {
        Symbol s = new Symbol(Symbol.SymType.PIXEL);
        s.copy(px);
        s.channel = 'a';
        return s;
    }

    private static Pixel getPixel(Symbol px)
    throws NoSuchMethodException {
        if (!px.isPixel())
            throw new NoSuchMethodException("Not a pixel!");
        return (Pixel) px.getValue();
    }
}
```

Image Object

```
/*
* IMLImage.java
* Written by Cindy L.
*/

import javax.imageio.*;
import java.util.ArrayList;
import java.util.LinkedList;


public class IMLImage {

    public int[][] pixels;
    public int width;
    public int height;
    public int imageType;

    public IMLImage() {
        pixels = null;
        width=-1;
        height=-1;
        imageType=0;
    }

    public IMLImage(Symbol s) throws Exception {
        if (!s.isImage()) {
            throw new Exception(s.toString() + " is not an image!");
        }

        ArrayList<Symbol> imglist = (ArrayList<Symbol>)s.getValue();
        imageType = s.imageType;
```

```
		width = imglist.size();

		for (int i=0; i<imglist.size(); i++) {
			if (!imglist.get(i).isArray()) {
				throw new Exception("Row " + i + "of " + s.toString() +
					" is not an array.");
			}
			ArrayList<Symbol> row = (ArrayList<Symbol>) imglist.get(i).getValue();
			if (i == 0) {
				height = row.size();
				pixels = new int[width][height];
			}
			for (int j=0; j<row.size(); j++) {
				Symbol sym = row.get(j);
				if (sym.getType() != Symbol.SymType.PIXEL) {
					throw new Exception(s.toString() + "[" + i + "]" +
						"[" + j + "] is not a pixel.");
				}
				Pixel p = (Pixel)sym.getValue();
				pixels[i][j] = p.toRGBA();
			}
		}
	}

	public ArrayList<Symbol> getPixels() {
		if (pixels == null) return null;
		ArrayList<Symbol> imglist = new ArrayList<Symbol>();
		for (int i=0; i<width; i++) {
			ArrayList<Symbol> row = new ArrayList<Symbol>();
			for (int j=0; j<height; j++) {
				row.add(j, new Symbol(createPixel(pixels[i][j])));
			}
			Symbol rowsym = new Symbol(Symbol.SymType.ARRAY);
			rowsym.setRefArray(row);
			imglist.add(i, rowsym);
		}
		return imglist;
	}

	private static Pixel createPixel(int rgba) {
		Pixel p = new Pixel();
		p.alpha = (rgba >> 24) & 0xff;
		p.red   = (rgba >> 16) & 0xff;
		p.green = (rgba >>  8) & 0xff;
		p.blue  = (rgba      ) & 0xff;
		return p;
	}

}
```

Main Class

```
/*
* IML.java
* Written by Steven C.  Zach S.
*/


import antlr.collections.AST;
import java.io.*;
import antlr.debug.misc.ASTFrame;
import antlr.CommonAST;

public class IML {

	//public IML() { };

	public static void run(String srcfile) throws Exception {
	    FileReader fr = new FileReader(new File(srcfile));
```

```
            String destfile = srcfile + ".preprocessor";
              FileWriter fw = new FileWriter(new File(destfile));

              // While next char is '#' {
              //   Grab a space
              //   Grab all chars until newline
              //   That's the filename
              //   Read that entire file, writing it to fw
              // }
              // Read fr entirely, writing it to fw

              char d;
              while ((d = (char) fr.read()) == '#') {        // while file begins with #
                    fr.read();                           // drop the space
                    String incfile = new String("");
                    char c = 0;
                    while (c != '\n') {
                          c = (char) fr.read();
                          if (c != '\n') incfile += c;
                    }
                    FileReader inc = new FileReader(new File(incfile));
                    while ((c = (char) inc.read()) > 0) {
                          if (c >= 256) break;
                          fw.write(c);
                    }
                    inc.close();
                    fw.write('\n');
              }
              fw.write(d);      // put the char back
              int c;
              while ((c = fr.read()) != -1) fw.write(c);
              fr.close();
              fw.close();

          FileReader processed = new FileReader(new File(destfile));

        IMLLexer lexer = new IMLLexer(processed);
        IMLParser parser = new IMLParser(lexer);
        parser.program();

        CommonAST t = (CommonAST) parser.getAST();

        IMLTreeParser treeParser = new IMLTreeParser();
        try {
              treeParser.program(t);
        } catch (Exception e) {
                    System.out.println("exception! ");
                    e.printStackTrace();
        }

        processed.close();
        new File(destfile).delete();
        }

        public static void test(String srcfile) throws Exception {
              run(srcfile, false);         // without AST jframe
        }

        public static void main(String[] args) {
              try {
                    if (args.length > 0)   run(args[0]);
                    else  System.out.println(“usage: java IML inputfile.iml”);
              } catch (Exception ex) {
                    ex.printStackTrace();
              }
        }
  }
```

Regression Suite

Regression Tester

```
import java.io.*;
import java.util.ArrayList;

/*
 * Cindy L.
 */
public class test_suite_1 {

      BufferedReader runResults = null;
      BufferedReader reader = null;
      ArrayList<String> testFiles, resultFiles;
      String[] results;
      String path;

      public test_suite_1() {

      }

      public test_suite_1(String path) {
            this.path = path;
            File basedir = new File(path);
            String[] list = basedir.list();
            testFiles = new ArrayList<String>();
            resultFiles = new ArrayList<String>();
            for (int i=0; i<list.length; i++) {
                  if (list[i].endsWith(".result")) {
                        String testfile = list[i].replace(".result", ".iml");
                        File f = new File(path + testfile);
                        if (!f.exists()) {
                              System.err.print("Test file: " + f.toString() + " not found! ");
                              System.err.println("Omitting from test suite.");
                        }
                        else {
                              testFiles.add(testfile);
                              resultFiles.add(list[i]);
                        }
                  }
            }
            results = new String[testFiles.size()];
      }

      public void setUp() {
            try {
                  for (int i=0; i<resultFiles.size(); i++) {
                        results[i] = "";

                        reader = new BufferedReader(new FileReader(path + resultFiles.get(i)));
                        String l;
                        while ((l = reader.readLine()) != null) {
                              results[i] += l;
                        }
                        results[i] = results[i].trim();
                        reader.close();
                        reader = null;
                  }
            } catch (IOException ioe) {
                  System.err.println(ioe.getMessage());
            }
      }

      public void runtests() {
            setUp();
            boolean result = true;
            try {
                  for (int i=0; i<testFiles.size(); i++)
```

```
                {
                      PrintStream oldOut = System.out;
                      PrintStream oldErr = System.err;
                      PrintStream stream = new PrintStream(
                            new BufferedOutputStream(
                             new FileOutputStream("../../output" + i + ".txt")));
                      System.setOut(stream);
                      System.setErr(stream);
                      IML.test(path+testFiles.get(i));
                      System.out.close();
                      System.setOut(oldOut);
                      System.setErr(oldErr);
                      result = result && compareResults(i);
                }
          } catch (Exception e) {
                e.printStackTrace();
          }
          if (result) System.out.println("Passed regression suite.");
    }


    public boolean compareResults(int i) {
          try {
          runResults = new BufferedReader(new FileReader("../../output" + i + ".txt"));

          String runLine, runTotal="";

          while ( (runLine = runResults.readLine()) != null) {
                runTotal += runLine;
          }

          runResults.close();

          if (!runTotal.equals(results[i].trim())) {
                System.out.println("FAILED: " + testFiles.get(i));
                System.out.println("expected: \n" + results[i]);
                System.out.println("actual: \n" + runTotal);
                return false;
          }
      }
          catch (FileNotFoundException fe) {
                System.out.println(fe.getMessage());
          }
          catch(IOException ioe) {
                System.out.println(ioe.getMessage());
          }
          return true;
    }

    public static void main(String args[]) {
          test_suite_1 testsuite = new test_suite_1("../testing/good/");
          testsuite.runtests();
    }


}
```

Success Cases

Success cases are working programs that IML is tested with to run correctly.

Returning Arrays

```
// TEST
// author: Steven C.

function ar()
{
        Pixel a[5][4];
        red a[2][3] = 15;
        print(red a[2][3]);
        return a;
}

function main()
{
        Pixel a[5][4];
        a = ar();
        print(a[2][3]);
        return 0;
}

// EXPECTED
5
(15, 0, 0, 0)
```

Unreachable Code

```
// TEST
// author: Zach S.

function a() {
        print(3);
        return 0;
}

function b() {
        print(5);
        return 0;
        print(6);      /* this should be unreachable code */
}

function main() {
        a();
        b();
        return 0;

        print("unreachable code");
}

// EXPECTED
3
5
```

Logical Comparisons

```
// TEST
//author: Zach S.

function main() {
        print("sanity check");
        if (2 <= 2) { print("lessequal"); }
        if (2 <= 3) { print("lessequal"); }
        if (2 <= 1) { print("broken"); }
}

// EXPECTED
sanity check
lessequal
lessequal
```

Channel Operator

```
// TEST
// author: Steven C.

function main()
{
       Pixel r;
       red r = 14;
       print(r);
       return 0;
}

// EXPECTED
(14, 0, 0, 0)
```

Copy by Value

```
// TEST
// author: Steven C.

function t13()
{
       print ("Copy-by-value");
       Int i[2][2];
       Int j[2];

       i[0][0] = 5;
       i[1][0] = 6;
       i[0][1] = 7;
       i[1][1] = 8;

       j = i[0];

       j[0] = 1;

       print(i[0][0]);
       print(i[1][0]);
       print(i[0][1]);
       print(i[1][1]);
       print(j[0]);
       print(j[1]);


       Int a[2][2];
       Float b[2];

       a[0][0] = 5;
       a[1][0] = 6;
       a[0][1] = 7;
       a[1][1] = 8;

       b = a[0];

       b[0] = 1.0;

       print(a[0][0]);
       print(a[1][0]);
       print(a[0][1]);
       print(a[1][1]);
       print(b[0]);
       print(b[1]);

}

function main()
{
       t13();
       return 0;
}
```

```
// EXPECTED
Copy-by-value
5
6
7
8
1
7
5
6
7
8
1.0
7.0
```

Return Statement Within Loops

```
// TEST
// author: Steven C.

function t12()
{
 print("returning out of a function");
 print(returnBreaking());
}

function returnBreaking()
{
 Int i;
 for (i = -1.5; i < 15.6; i = i + 1) {
  print(i);
  if (i >= 12.5) {
   print("foo");
   return 0;
  }
 }
 print("should never reach");
 return 1;
}

function main()
{
 t12();
 return 0;
}


// EXPECTED
returning out of a function
-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
foo
0
```

Function Argument Passing

```
// TEST
// author: Steven C.

function t11()
{
	print ("Argument passing");
	f(2, 5.6, "string");
}

function f(Int a, Float b, String c)
{
	print (a + (b + c));
	return 0;
}

function main()
{
	t11();
	return 0;
}


// EXPECTED
Argument passing
25.6string
```

Operator Chaining

```
// TEST
// author: Steven C.

function t10()
{
	print ("Chaining");

	Int a; Int b; Int c; Int d;
	a = 2; b = 3; c = 4; d = 5;

	d = a + b + c + d;
	print (d);

	d = c = b = a;
	print (d);

	d = f (g (h (a)));
	print(d);

	return 0;
}

function f(Int x) { return x+1; }
function g(Int x) { return x+1; }
function h(Int x) { return x+1; }

function main()
{
	t10();
	return 0;
}


// EXPECTED
Chaining
14
2
5
```

Dynamic Scoping

```
// TEST
// author: Eric H.

function t9()
{
        print ("Scoping");

        String g;
        g = "outermost scope";
        if (1)
        {
                String scope1;
                scope1 = "1st scope";
                if (2)
                {
                        String scope2;
                        scope2 = "2nd scope";

                        if (3)
                        {
                                String g;
                                g = "innermost scope";

                                String scope3;
                                scope3 = "3rd scope";

                                print (scope1);
                                print (scope2);
                                print (scope3);
                                print (g);
                        }
                }
        }

        return 0;
}


function main()
{
        t9();
        return 0;
}


// EXPECTED
Scoping
1st scope
2nd scope
3rd scope
innermost scope
```

Implicit Type Conversions

```
// TEST
// author: Steven C.

function t7()
{
print ("Initialization");
Int x;
print (x);
print ("Casting Int to Float");
x = 4;
Float y;
print (y=x);
print ("Casting Int to String");
String z;
print (z=(x+y)*2.5);
}


function main()
{
        t7();
        return 0;
}


// EXPECTED
Initialization
0
Casting Int to Float
4.0
Casting Int to String
20.0
```

Recursion

```
// TEST
// author: Steven C.

function t6()
{
print ("recursion");
recurse(0);
print ("double-recursion");
toss(0.0);
print ("factorial of 5");
print (factorial(5));
return 0;
}

function recurse(Int x)
{
        print (x);
        if (x < 9) { recurse(x+1); }
        else { print (x+1); }
        print (x);
        return 0;
}

function toss(Float x)
{
        print (x);
        if (x < 1)     { toss_back(x+0.1); }
        return x;
}

function toss_back(Float y)
{
        return toss(y+0.1);
}

function factorial(Int n)
{
        if (n <= 1)
        { return 1; }
        else
        { return n * factorial(n-1); }
}

function main()
{
        t6();
        return 0;
}


// EXPECTED
recursion
0
1
2
3
4
5
6
7
8
9
10
9
8
7
6
5
4
3
2
1
0
double-recursion
0.0
0.2
0.4
0.6
0.8000001
1.0000001
factorial of 5
120
```

Variable Operands

```
// TEST
// author: Steven C.

function t5()
{

print ("printing");
print ("Hard-coded string");

String s;
s = "Variable s is a string";
print (s);

String c;      c = "concat";
String d;      d = "enation";
print (c+d);

String f;
f = "Two plus three is ";
Int a;  a = 2+3;
print (f + a);

String strange_chars;
strange_chars = "<>:?{}       !@#$%^&*( )_+|,.~";
print (strange_chars);

print ("Casting and assignment");
Float y;
String z;
print (z = y);

return 0;
}

function main()
{
	t5();
	return 0;
}


// EXPECTED
printing
Hard-coded string
Variable s is a string
concatenation
Two plus three is 5
<>:?{}      !@#$%^&*( )_+|,.~
Casting and assignment
0.0
```

If - Else

```
// TEST
// author: Steven C.

function t4()
{

print ("if-else-statement");
Int i;  i=10;
if ((i%2) == 0)
{
	print ("even");
}
else
{
	print ("odd");
}

print ("nested if-elses");
Int j;  j=11;
if ((j%3) == 0)
{ print ("no remainder"); }
else
{
	if ((j%3) == 1)
	{ print ("remainder of 1"); }
	else
	{ print ("remainder of 2"); }
}

return 0;
}

function main()
{
	t4();
	return 0;
}


// EXPECTED
if-else-statement
even
nested if-elses
remainder of 2
```

Iteration

```
// TEST
// author: Steven C.

function t3()
{
print ("for loop");
Int i;
for (i=0; i<5; i = i+1)
{
       print (i);
}

print ("for loop with j=0 auto-initialization");
Int j;
for (; j>-5; j = j - 1)
{
       print (j);
}

print ("break statement");
Int k;
for (k=0; k<20; k=k+1)
{
       if (k == 10)
       {
              break;
       }
}
print (k);

print ("nested for loops");
Int s;
Int t;
for (s=0; s<3; s=s+1)
{
       for (t=0; t<3; t=t+1)
       {
              print (s + ", "  + t);
       }
}

print ("for loop with functions");

print ("while loop");
Int r;
r = 1;
while (r <= 2048)
{
       print (r = r * 2);
}

return 0;
}


function main()
{
       t3();
       return 0;
}


// EXPECTED
for loop
0
1
2
```

```
3
4
for loop with j=0 auto-initialization
0
-1
-2
-3
-4
break statement
10
nested for loops
0, 0
0, 1
0, 2
1, 0
1, 1
1, 2
2, 0
2, 1
2, 2
for loop with functions
while loop
2
4
8
16
32
64
128
256
512
1024
2048
4096
```

Mathematical and Logical Operations

```
// TEST
// author: Steven C.

function main()
{
       print("Arithmetic");
       print(1+2+3+4+5);
       print(1-2+3-4+5);
       print(4.3 * 12 * 05);
       print(54 / 5.9);
       print(1.0 / 800);
       print(13 % 7);
       print(3 + 4 / 7);
       print((3 + 4) / 7);

       print("Logic");
       print(5 < 5);
       print(5 <= 5);
       print(5 > 3);
       print(3 == 3.0);
       print(3 != 3);
       print(0 && 2 && 4 && 8);
       print(2 || 4 || 0 || 8);
       print(2 && (0 || 1));
       print (!100.0);
}


// EXPECTED
Arithmetic
15
3
```

```
258.0
9.152542
0.00125
6
3
1
Logic
0
1
1
1
0
0
1
1
0
```

Memory Management

```
// TEST
// author: Zach S.

function main() {
	Pixel a[1024][768];
	print("lived to tell the tale");
}


// EXPECTED
lived to tell the tale
```

Include Directives

```
// TEST FILE 1
// author: Zach S.

incs.txt
# /home/zmv2102/eclipse_workspace/testing/good/included.txt
function main() {
	a();
	print("do nothing");
}


// TEST FILE 2
// author: Zach S.

function a() {
	print("includes work!");
}


// EXPECTED
includes work!
do nothing
```

Image – 2D Pixel Array Conversion

```
// TEST
// author: Steven C.

function main()
{
	Image a;
	Pixel e[2][3];
	a = e;
	print(a);
}

// EXPECTED
[[(0, 0, 0, 0), (0, 0, 0, 0), (0, 0, 0, 0)], [(0, 0, 0, 0), (0, 0, 0, 0), (0, 0, 0, 0)]]
```

Empty Block

```
// TEST
// author: Steven C.

function main() {

        Int i;
        for ( ; i!=1; i=1) { }
        while (1==2) { }
        print ("success");
}


// EXPECTED
success
```

String Operations

```
// TEST
// author: Steven C.

function main()
{
        String s;      Int i;
        s = "A man, a plan, a canal, Panama!";
        for (i = length s; i > 0; i = i - 1) {
                print (s[i-1]);
        }
}


// EXPECTED
!amanaP ,lanac a ,nalp a ,nam a;
```

Break Statement

```
// TEST
// author: Zach S.

function main() {
        Int i;
        print("numbers from one to three:");
        i = 0;
        while (i < 10) {
                i = i + 1;
                print (i);
                if (i == 3) { break; }
        }
        print("numbers from one to four:");
        i = 0;
        while (i < 10) {
                i = i + 1;
                print (i);
                if (i > 1) { if (i > 2) { if (i > 3) { break; } } }
        }
        print("numbers from one to six:");
        i = 0;
        Int j; j = 0;
        while (i < 6) {
                i = i + 1;
```

```
               print (i);
               while (j < 1) { j = j + 1; if (i != 0) { break; print("bad"); } }  // should do
nothing
       }
}


// EXPECTED
numbers from one to three:
1
2
3
numbers from one to four:
1
2
3
4
numbers from one to six:
1
2
3
4
5
6
```

Opening an Image

```
// TEST
// author: Cindy L.

function main() {
       Image a;
       "../testing/good/basic_pixels.bmp" open a;
       print(a);
}


// EXPECTED
[[(0, 0, 0, 255), (255, 255, 255, 255), (0, 0, 0, 255)], [(255, 255, 255, 255), (0, 0, 0, 255),
(255, 255, 255, 255)], [(0, 0, 0, 255), (255, 255, 255, 255), (0, 0, 0, 255)]]
```

Function Calls

```
// TEST
// author: Zach S.

function A() {
       Int a;
       Int b;
       a = 4;
       F();
       for (a = 0; a != 2; a = b = 2) { a = 4; }
       a = 4;
       B();
}

function F() {
       Int a;
       while (a <= 30) {
               a = a + 1;
               if (a == 6 && 1 && 1 && (0 || 1) && !0) { break; }
       }
       b = 6;
}

function B() {
       Int b; E();
       return C(a, b);
}

function C(Int b, Int a) {
       a = 2;
       D(a * b);
       return 5;
}

function D(Int c) {
       print(c);
}

function E() {
       b = 3;
}

function main() {
       Int b;
       b = 2;
       A();
}


// EXPECTED
8
```

Failure Cases

Failure cases are buggy programs that IML is tested with to report and handle errors. Note: “failure cases” does **not** mean the IML interpreter failed to work with these tests.

Divide By Zero

```
function t3()
{
      print ("Divide by zero");
      return 9/0;
}

function main()
{
      t3();
      return 0;
}
```

Array Index out of Bounds

```
function t4()
{
      print ("Array index out of bounds");
      Int e[4];
      print (e[1] = 1); // in bounds
      print (e[4] = 4); // out of bounds
      return 0;
}

function main()
{
      t4();
      return 0;
}
```

Using an Undefined Variable

```
function t5()
{
      print ("Using an undefined variable");
      a = 5;
      Int a;
      return 0;
}

function main()
{
      t5();
      return 0;
}
```

Illegal Scoping

```
function t6()
{
      print ("Illegal scoping");

      if (1)
      {
            String scope1;
            scope1 = "1st scope";
            if (2)
            {
                  String scope2;
                  scope2 = "2nd scope";
            }
            if (3)
            {
                  String scope3;
                  scope3 = "3rd scope";

                  print (scope1);
                  print (scope3);
                  print (scope2);   // failure
            }
      }

      return 0;
}

function main()
{
      t6();
      return 0;
}
```

Incorrect Arguments

```
function t7()
{
      print ("Incorrect arguments");
      f(2);
      return 0;
}

function f(Image a)
{
      return 0;
}

function main()
{
	t7();
	return 0;
}
```

Test Programs

Gaussian Blur

```
/*
* gaussian_blur.iml
* Written by Steven C.
*/


function gaussian_blur(Image a) {
	Int x; Int y;
	for (x = 1; x < cols a - 1; x = x + 1) {
		for (y = 1; y < rows a - 1; y = y + 1) {
			red a[x][y] = (1.0 / 16.0) * (4.0 *red a[x][y] +
				2.0 *red a[x][y+1] + 2.0 *red a[x][y-1] + 2.0 *red a[x+1][y] + 2.0 *red 
a[x-1][y] +
				red a[x+1][y+1] + red a[x+1][y-1] + red a[x-1][y+1] + red a[x-1][y-1]);

			green a[x][y] = (1.0 / 16) * (4 *green a[x][y] +
				2 *green a[x][y+1] + 2 *green a[x][y-1] + 2 *green a[x+1][y] + 2 *green 
a[x-1][y] +
				green a[x+1][y+1] +green a[x+1][y-1] +green a[x-1][y+1] +green a[x-1][y-
1]);

			blue a[x][y] = (1.0 / 16) * (4 *blue a[x][y] +
				2 * blue a[x][y+1] + 2 *blue a[x][y-1] + 2 *blue a[x+1][y] + 2 *blue a[x-
1][y] +
				blue a[x+1][y+1] +blue a[x+1][y-1] +blue a[x-1][y+1] +blue a[x-1][y-1]);

			alpha a[x][y] = 255;

			//print(a[x][y]);
		}
	}
	return a;
}

function main() {
	Image a;
	"<FILE_PATH>" open a;
	a = gaussian_blur(a);
	a save "<FILE_PATH>";
}

/*
* end of deinterlace.iml
*/
```

Image Deinterlacer

```
/*
* deinterlace.iml
* Written by Steven C.
*/

function pixel_average(Pixel a, Pixel b) {
        Pixel c;
        red c = (red a + red b) / 2;
        green c = (green a + green b) / 2;
        blue c = (blue a + blue b) / 2;
        alpha c = 255;
        return c;
}

function deinterlace_average(Image g) {
        Int i;  Int j;
        for ( ; i < cols g; i = i + 1) {
                for (j = 0; j < rows g-1; j = j + 1) {
                        g[i][j] = pixel_average(g[i][j], g[i][j+1]);
                }
        }
        return g;
}

function deinterlace_duplicate(Image g) {
        Int i; Int j;
        for (i = 0; i < cols g; i = i + 1) {
                for (j=1; j < rows g; j = j + 2) {
                        g[i][j] = g[i][j - 1];
                }
        }
        return g;
}

function main() {
        Image interlaced;
        ""<FILE_PATH1>" open interlaced;
        deinterlace_average(interlaced) save ""<FILE_PATH2>";
        deinterlace_duplicate(interlaced) save "<FILE_PATH3>";
                }
        }
        print ("end");
}

/*
* end of deinterlace.iml
*/
```

IML Standard Library

```
function maximum_float(Float a, Float b) {
        if (a > b) { return a; }
        return b;
}

function maximum_integer(Int a, Int b) {
        if (a > b) { return a; }
        return b;
}

function power(Float x, Int n)
{
        Float r;        r = 1;
        while (n) {
                if (n % 2) {
                        r = r * x;
                        n = n -1;
                }
                x = x * x;
                n = n / 2;
        }
        return r;
}
```

```
function factorial(Int n) {
        if (n <= 1) { return n; }
        return n * factorial(n - 1);
}

function cos(Float x) {
        return (sin(x + 3.1415926535 / 2));
}

function sin(Float x) { // sin of angle x in degrees
        while (x > 2 * 3.14159265358) { x = x - (2 * 3.14159265358); }
        if (x > 3.14159265358) { x = 3.14159265358 - x; }

        Int precision; precision = 8;
        Float val; val = 1;

        while (precision > 0) {
                val = 1 - val * x * x / (2 * precision) / (2 * precision + 1);
                precision = precision - 1;
        }
        return x * val;
}

function tan(Float x) {
        return sin(x) / cos(x);
}

function largest_dimension(Image g) {
        Int h;  h = rows g;
        Int w;  w = cols g;
        if (h > w) { return h; }
        return w;
}

function crop_absolute(Image g, Int x1, Int y1, Int x2, Int y2) {
        if (bad_bounds(g, x1, y1) || bad_bounds(g, x2, y2) || x2 <= x1 || y2 <= y1) {
                print("error: bad crop bounds");
                return 0;
        }
        Int x; Int y;
        Pixel im[x2 - x1][y2 - y1];

        for (x = x1; x < x2; x = x + 1) {
                for (y = y1; y < y2; y = y + 1) {
                        im[x - x1][y - y1] = g[x][y];
                }
        }
        return im;
}


function crop_relative(Image g, Int top, Int bottom, Int left, Int right) {
        Int h;  h = rows g;
        Int w;  w = cols g;

        if (top < 0 || bottom < 0 || left < 0 || right < 0) {
                print("error: crop bounds must be positive");
                return 0;
        }
        if (h - top - bottom < 0 || w - left - right < 0) {
                print("error: crop bounds must be shorter than image dimensions");
                return 0;
        }

        Int i;  Int j;
        Pixel im[w - left - right][h - top - bottom];

        for ( ; i < w - left - right; i = i + 1) {
                for (j = 0; j < h - top - bottom; j = j + 1) {
                        im[i][j] = g[i+top][j+left];
                }
        }
        return im;
}

function bad_bounds(Image g, Int x, Int y) {
        return x < 0 || y < 0 || x >= cols g || y >= rows g;
}

function rad(Float x) {
        return x * 3.14159255358 / 180.0;
}

function grayscale (Image g)
{
      Int i;      Int j;
      for (i = 0; i < rows g; i = i + 1) {
            for (j = 0; j < cols g; j = j + 1) {
                  Float avg;
                  Pixel pxl;
                  pxl = input[i][j];
                  avg = ( red pxl + green pxl + blue pxl ) / 3.0 ;
                  red input[i][j] = green input[i][j] = blue input[i][j] = avg ;
            }
      }
      return input;
}
```