YOLOP:

Your Octothorpean Language for Optical Processing

Final Report

Friday December 20th, 2013

Sasha McIntosh [sam2270]

Jonathan Liu [jl3516]

Lisa Li [ll2768]

1 Introduction

1.1 Motivation

The YOLOP Language is design to aid developers in creating image manipulation languages. Cropping, changing images to black and white and importing photos can prove very difficult in most languages. YOLOP aims to simplifies these common processing tasks and provide a user-friendly language to transform photos by simplifying data types and providing key methods with our standard library.

Potential uses for programming in YOLOP thus include developing typical image processing programs to apply a standardized color mapping onto each part of the image (i.e. like a popular application's use of filters), sharply saturating colors to identify and separate specific areas of an image, manipulating and comparing multiple charts and graphs together, and more!

2 Language Reference Manual

YOLOP (Your Octothorpean Language for Optical Processing) is an image manipulation language designed to aid developers in programming applications to process their images and simplify complicated tasks. This language reference manual details the specific lexical and naming conventions, syntax notation, declarations and statements, as well as the built-in functions and standard grammar of YOLOP.

2.1 Lexical Conventions

Comments

Comments begin with this symbols @# and terminate with the symbols #@. Only the text inside the symbols are commented out. There is no separate definition for single-line comments.

@# VALID COMMENT #@

@# NOT VALID

Variable Names

Variable Names are made up of a sequence of letters. Only the first 15 characters are significant.

Keywords

Keywords in the language and cannot be used in any way other than their respective purposes. These words are:

for	int	function	read_in	getc
while	string	return	write_out	setc
if	pixel	continue	get	__print
else	image	break	set	

Types

integer	string	image	pixel

2.2 Syntax Notation

Tokens in a program are a series of characters broken up by separators. Spaces are considered whitespace and a series of spaces is treated as a single whitespace character. Tokens must be separated by a separator or an operator in order to be valid.

Token Separators

Token separators include whitespace characters and commas. They are used to distinguish one token from another.

New Line Separators

A semi colon is used to distinguish the end of one line in the program from another.

Grouping Separators

Grouping separators are used to group sets of tokens. Grouping separators include parentheses (), braces {} and brackets []. The correct usage of these separators is discussed in depth later in the manual.

2.3 Naming

Identifiers

Function and variable names are identifiers and can be made up of alphanumeric characters (0-9 and a-z) as well as underscores. The identifier must begin with either an underscore or a letter and is case sensitive. While a name can be infinitely long, only the first 15 characters will be used to reference an identifier.

Example:

function a_1() {}   @# legal #@

function 1_a() {}   @# illegal #@

@# identifier cannot begin with a number #@

Example:

int _12345678901234444   @# legal #@

int _12345678901234555   @# illegal #@

@# variable name is already in use #@

2.4 Expressions

Primary Expressions

constant

An integer type constant is a primary expression.

string

A string is a primary expression.

pixel

A pixel is a primary expression.

image

An image is a primary expression.

[expr]

Closed brackets around an integer number indicate access to an array element in the position determined by the contained integer.

(expr1)

Parentheses are used to declare function parameters, enclose if statements, while statements and for statements.

semicolons (**;**)

Semicolons are used to signal the end of a line and to distinguish one statement from another.

comma (**,**)

Commas are used to separate parameters in function call.

Mathematical Operators

expr + expr

The operator + groups left-to-right. It computes the sum of two expressions.

expr - expr

The operator - groups left-to-right. It computes the difference of two expressions.

expr * expr

The operator * groups left-to-right. It computes the product of two expressions.

expr / expr

The operator / groups left-to-right. It computes the quotient of two expressions.

Relational/Equality Operators

expr < expr

The operator < groups left-to-right. It returns 1 (meaning true) when the value of the left expression is less than that of the expression on the right, and 0 otherwise.

expr > expr

The operator > groups left-to-right. It returns 1 (meaning true) when the value of the left expression is greater than that of the expression on the right, and 0 otherwise.

expr <= expr

The operator <= groups left-to-right. It returns 1 (meaning true) when the value of the left expression is less than or equal to that of the expression on the right, and 0 otherwise.

expr >= expr

The operator >= groups left-to-right. It returns 1 (meaning true) when the value of the left expression is greater than or equal to that of the expression on the right, and 0 otherwise.

expr == expr

The operator == groups left-to-right. It returns 1 (meaning true) when the left expression is exactly equal to the value than the expression on the right, and 0 otherwise.

expr ~= expr

The operator ~= groups left-to-right. It returns 1 (meaning true) when the left expression is not exactly equal to the value than the expression on the right, and returns 0 otherwise.

Logical Operators

expr* && *expr

Evaluates to 1 when both expressions evaluate to non-zero values. There is a left-to-right evaluation. If *expr1* evaluates to 0, then *expr2* will not be evaluated.

expr* || *expr

Evaluates to 1 when either expression evaluates to a non-zero value. There is a left-to-right evaluation. If *expr1* evaluates to 1, then *expr2* will not be evaluated.

~expr

The not operator negates the boolean value of the expression. “True” becomes 0 and “false” becomes 1.

Assignment Operator

lvalue* = *expr

All assignment operators are right associative. They assign the evaluation of *expr* to the object *lvalue*.

2.5 Declarations

Variable Declarations

Variable declarations have the form *specifier var_name.* Variables must be declared before a variable is assigned to them. Type specifiers are *int* for integers, *string* for strings, *pixel* for pixels and *image* for images. A variable cannot be created without a type specifier.

Examples:

```
int x; x = 6;                  @# Integer Declaration and Assignment #@
string y; y = “why”;           @# String Declaration  and Assignment #@
pixel z; z = [255, 255, 255, 100];      @# Pixel Declaration  and Assignment  #@
image g; g =[200, 300];                 @# Image Declaration and Assignment  #@
```

Function Declarations

Function declarations have the form *function type var_name(type param1, type param2, …) {stmt1; stmt; …}.* The contents of the function include all lines within the brackets separated by semi-colons. The function parameters are contained within a set of parentheses, immediately following the variable name.

Examples:

```
function int add( type x, type y ) { return x + y; }
```

2.6 Statements

Statements are executed in sequence, and consist of either new object/variable declarations, modifying or assigning declarations, block statements, or control flow statements.

Conditional Statements

Conditional statements consist of either *if* or *if-else* blocks.

Examples:

```
if ( expression ) {
        if (expression) { statement; }
} else { statement; }
```

Iteration Statements

Iteration statements consist of *for* or *while* blocks.

For Loops

For loops must take in three expressions contained in parentheses and separated by semi-colons, and it evaluates *expression1* first and only once. *expression2* is a boolean expression and upon evaluating true, runs the *statement*. After the *statement* has been executed, *expression3* is executed. This continues until *expression2* evaluates false, at which point the for loop is terminated.

Example:

```
for ( expression1; expression2; expression 3 ) {
        statement; }
```

While Loops

While loops take in one *expression*, which is a boolean expression that is evaluated each time before *statement* is executed. The while loop executes the *statement* repeatedly until the *expression* is evaluated as false.

Example:

```
while ( expression ) { statement; }
```

Jump Statements

Jump statements consist of *return*, *break*, or *continue* statements.

Return

The *return* keyword ends the function call by returning the value of an *expression* to the function’s caller.

Example:

```
    function int add ( expression ) {
        if (expression) { return integer; }
        statement;
    }
```

Break

The *break* keyword terminates the smallest enclosing *while* or *for* loop, and jumps to the code displayed after that loop.

Example:

```
    while ( expression ) {
        if ( expression ) { break; }
        statement;
    }
```

Continue

The *continue* statement allows control to pass to the end of the smallest enclosing *for* or *while* loop, essentially skipping over one iteration.

Example:

```
    while ( expression ) {
        if ( expression ) { continue; }
        statement;
    }
```

2.7 Built-In Functions

print(str);

Prints the string str to the console.

read_in(img, filename, ext);

Reads an image file into image img.

write_out(img, filename, ext);

Writes an image img to a file *filename.ext*.

px = get(img, x, y);

Retrieves the pixel px at coordinate (x,y) from image img.

set(img, x, y, px);

Sets the pixel at coordinate (x,y) in the image img to be the color denoted by pixel px.

val = getc(px, "option");

Retrieves the integer value of option "r", "g", "b" or "o" (red, green, blue, opacity) from pixel px.

setc(px, "option", val);

Sets the value of the option "r", "g", "b" or "o" (red, green, blue, opacity) of pixel px to the given integer value val.

2.8 Scope Rules

Global Scope

Global constants defined outside of any function calls are noted as global constants and contain a global scope. They can be accessed by any statements or expressions within the file.

Example:

```
int z = 25;
function int add( int x, int y ) {
        return x + y + z; }
```

Block Scope

Blocks contain a local scope, where variables defined within a block are local and accessible within that block. Blocks can access previously defined local variables in their parent's block as well as global variables.

Example:

```
    function int add( int x, int y ) {          @# x and y are local variables #@
        int z; z = x + y;          @# z is a local variable #@
        if ( x == 2 ) {
            int a; a = x + 2;          @# a and x are accessible local variables #@
            return a+2; }
        if ( x < 2 ) {
            return a; }                @# error: a does not exist in this scope #@
        return z; }                    @# z is within the local scope #@
```

See Apendix for more examples.

3 Tutorial

3.1 Variables

Variable declarations and initializations must occur separately. For example

```
int x; x=5;          @#     valid          #@
int x = 5;           @#     invalid #@
```

3.1.1 Pixels and Images

Pixels are declared like any other variable, however they are initialized differently. For a pixel p, we can initialize it to have R/G/B/O values of 50, 100, 150 and 100 respectively as follows:

```
p = [50,100,150,100];@# A pixel with R = 50, G = 100, B = 150, and O = 100 #@
```

In general, the pixel is created as

[REDVALUE, GREENVALUE, BLUEVALUE, OPACITYVALUE] where the colour values are of type int and vary from 0-255 and opacity is also of type int varying from 0-100.

Image initialization is very similar. They are created as shown below

```
Image i; i = [400,500];@# Creates a 400x500 pixel blank image #@
```

Note that 400 is the height, 500 the width, and that they are both values of type int.

3.1.2 Accessing and changing pixels

To get a pixel at position (x,y) we call p = get (img, x, y) where p is a pixel and img is the image in question and x and y are the x and y coordinates of the pixel in question.

To change the pixel at position (x,y) we call set(img,x,y,p) where img is the image in question, x and y are the x and y coordinates respectively, and p is the pixel which we wish to place at position (x,y).

To get the red, blue, green, or opacity value of a pixel, we call i = getc(p,s) where i is an int, p is the pixel in question, and s is either “r” for red, “g” for green, “b” for blue, or “o” for opacity. So to get the blue value of pixel p and store it into i, we call i = getc(p,”b”).
Similarly, we can set one of the r/g/b/o values of a pixel with setc(p,s,v) where p and s have the same restrictions as for getc, and v is an int. So to set the red of pixel p to 50, we call setc(p,”r”,50).

3.1.3 Image I/O

To load an image file from the current directory, we call read_in(img, f , e), where img is the image we want to associate to this file, f is the filename (string), and e is the extension (string). For example, to read “cat.jpg” into image i, we call read_in(i, “cat”, “jpg”).

To save an image to the current directory, we call write_out(img, f, e). If f = “dog” and e = “png”, this will write the image img into the file “dog.png” in the current directory.

3.2 Function declarations

Function declarations are as follows

```
function return_type function_id ( param1, param2) {

        function body

}
```
Note that a valid program **must** contain a main() function.

4 Project Plan

4.1 Project Timeline & Team Responsibilities

Week 1

Sasha: scanner.mll, parser.mly

Lisa: interpret.ml

Jon: translate.ml

Week 2

Sasha: ast.ml, Makefile

Lisa: interpret.ml, test suite

Jon: translate.ml, test suite

Week 3

Sasha: ast.ml, Makefile

Lisa: interpret.ml, test suite

Jon: translate.ml, test suite

Week 4

Sasha: test suite

Lisa: yolop.ml, test suite

Jon: yolop.ml, test suite

Week 5

Sasha: LRM Changes, Final Report

Lisa: presentation PPT

Jon: test program

4.2 Software Development Environment

As a team, we agreed to use Git for version control because we were all familiar with it. We developed code on separate machines using mainly emacs and vim. Working in the same room, we were able to easily resolve any merge conflicts that came up.

4.3 Project Milestones

Week of 9/22:	Language Proposal
Week of 10/6:	Scanner
	Parser
Week of 10/20:	Language Reference Manual (Draft)
	Abstract Syntax Tree
	Makefile
Week of 11/3:	Interpreter
	Translator
Week of 12/1:	Test Suite
Week of 12/8:	Compiler
Week of 12/15:	Final Report
	Language Reference Manual (Final)
	Presentation (and Rehearsed)

5 Architectural Design

Because writing the source code for an image editing suite would have taken an enormous amount of time, we decided to use a pre-existing library. This library was Magick++ and it is written in C++. Because it is not native Ocaml code, we chose to approach the compilation as follows: we first tokenize the source code and create an abstract syntax tree that represents the program. From this intermediate representation, we generate the appropriate C++ code that captures the desired behaviour. This C++ back-end utilizes the Magick++ library to perform image manipulation.

5.1 Scanner

Takes the source code and produces a set of tokens.

5.2 Parser

Takes the tokens from the scanner output and via ast.ml generates an abstract syntax tree represnting the program.

5.3 Translate to C++

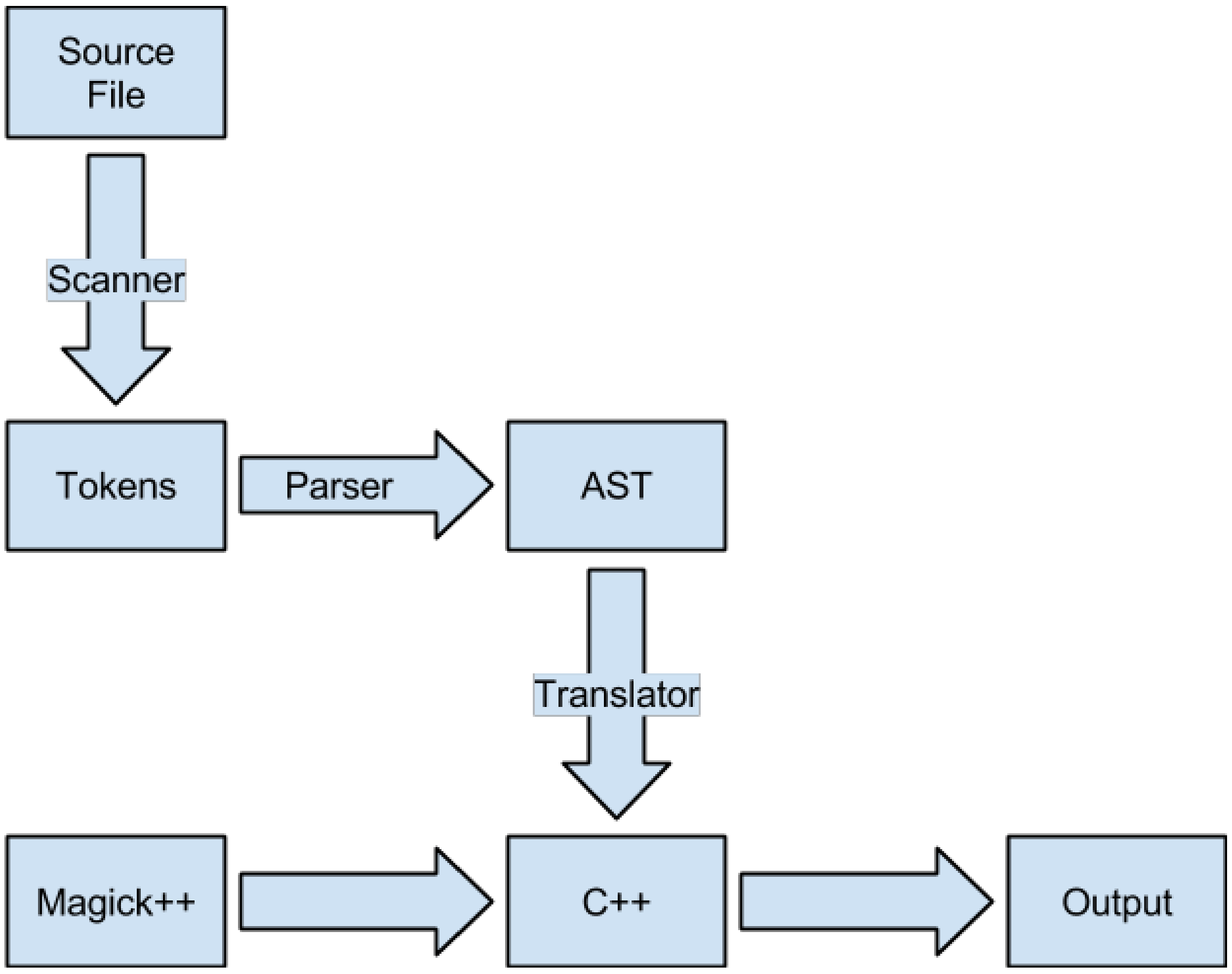
From the AST, we generate C++ code via translate.ml.

5.4 Compiling and Running

After running the compiler, a .cpp file will be produced. For example if you were to compile “test.yolop” you would run “./yolop -c test.yolop” which in turn produces test.cpp which will then need to be compiled with a C++ compiler (e.g. gcc).

5.5 Magick++

This is the C++ library that performs the heavy lifting for our image processing. In combination with the generated C++ code from interpret.ml, we produce the desired output.



6 Test Suite

6.1 Basics

We created a basic suite of test programs to verify that the syntax syntax used in our program did in fact function as we aimed it to. See Appendix 8.1 for test suite code.

6.3 Programs

See Appendix for test programs.

7 Lessons Learned

7.1 Set Firm Deadlines

In the same vein as “Start Early” make and set hard deadlines. We aimed to meet once a week and had deadlines but as the semester went on, homework and exams and other things took up more time. We ended putting things off and missing goal dates which made finals week crunch time.

7.2 Thoroughly Plan Your Language

This one is somewhat difficult to accomplish. Though we thought we had thought through our language well, when it came to development we ended up spending a lot of time going back and modifying/revising previously written code. We wasted a lot of time doing this.

8 Appendix

8.1 Test Suite

8.1.1 test-decl-func.yolop

```
function main () {
	print("OK");

}
```

8.1.2 test-decl-vars.yolop

```
function main () {
	int x;
	if ( 1 )  { print("OK"); }
	string s;
	if ( 1 )  { print("OK"); }
	pixel p;
	if ( 1 )  { print("OK"); }
	image g;
	if ( 1 )  { print("OK"); }
}
```

8.1.3 test-func-getcsetc.yolop

```
function main () {
   @# getc #@
   pixel px; px = [255, 255, 255, 100];
   int val;
   val = getc( px, "r");
   if ( val == 255 )   { print( "OK" );      }
   else                 { print( "FAIL" );    }
   @# setc #@
   set( px, "b", 0);
   print( "OK" );
}
```

8.1.4 test-func-getset.yolop

```
function main() {
   @# set function #@
   image img; img = [3,3];
   set( img, 2, 2, 200 );
   write_out ( img, "test-func-getset-out", "jpg");
   @# get function #@
   pixel px;
   px = get(img, 2, 2);
   print( "OK" );
}
```

8.1.5 test-func-print.yolop

```
function main () {
   print( "OK" );
}
```

8.1.6 test-func-readwrite.yolop

```
function main () {
   image img;
   read_in( img, "cat1", "jpg" );
   print( "OK" );
   write_out( img, "out", "jpg");
   print( "OK" );
}
```

8.1.7 test-ops-asn.yolop

```
function main () {
      int x; x = 10;
      if ( x == 10 )  { print("OK");    }
      else            { print("FAIL");  }
      string s; s = "OK";
      if ( 1 )        { print(s);       }
      pixel p; p = [255, 255, 255, 100];
      if ( 1 )        { print("OK");    }
      image g; g = [100, 200];
      if ( 1 )        { print("OK");    }
}
```

8.1.8 test-ops-logic.yolop

```
function main () {
        int w; w = 0;
        int x; x = 1;
        int y; y = 1;
        int z; z = 0;
        if ( x && y )   { print("OK");   }
        else             { print("FAIL"); }
        if ~( x && z ) { print("OK");   }
        else             { print("FAIL"); }
        if ( w || x )   { print("OK");   }
        else           { print("FAIL"); }
        if ( w || z )   { print("OK");   }
        else           { print("FAIL"); }
        if ( x == 50 )  { print("OK");   }
        else           { print("FAIL"); }
        if ( x ~= y  )  { print("OK");   }
        else           { print("FAIL"); }
}
```

8.1.9 test-ops-math.yolop

```
function main () {
        int x; x = 10;
        int y; y = 5;
        if ( x + y == 15 )  { print("OK");   }
        else               { print("FAIL"); }
        if ( x - y == 5  )  { print("OK");   }
        else               { print("FAIL"); }
        if ( x * y == 50 )  { print("OK");   }
        else               { print("FAIL"); }
        if ( x / y == 2  )  { print("OK");   }
        else               { print("FAIL"); }
}
```

8.1.10 test-ops-relations.yolop

```
function main () {
        int x; x = 10;
        int y; y = 5;
        if ( x > y )    { print("OK");   }
        else            { print("FAIL"); }
        if ( y < x )    { print("OK");   }
        else            { print("FAIL"); }
        if ( x >= y )   { print("OK");   }
        else            { print("FAIL"); }
        if ( y <= x )   { print("OK");   }
        else            { print("FAIL"); }
        if ( x == 50 )  { print("OK");   }
        else            { print("FAIL"); }
        if ( x ~= y  )  { print("OK");   }
        else            { print("FAIL"); }
}
```

8.1.11 test-ops-unary.yolop

```
function main () {
        int x; x = 10;
        if ( x++ == 11 ) { print("OK");   }
        else                    { print("FAIL"); }
        if ( x-- == 10 ) { print("OK");   }
        else                    { print("FAIL"); }
        if ( -x == -10 ) { print("OK");   }
        else             { print("FAIL"); }
}
```

8.1.12 test-stmts.yolop

```
function main() {
	@# if/else #@
	if ( 1 )  { print("OK");    }
	if ( 0 )  { print("FAIL");  }
	else      { print("OK");    }
	@# for loops #@
	int x;
	for ( x = 0; x < 2; x=x+1 ) {
		print("OK");
	}
	@# while loops #@
	x = 0;
	while ( x < 2 ) {
		print("OK");
		x = x+1;
	}
	@# return #@
	x = 1;
	function int incr ( int y ) {
		return y+1;
	}
	x = incr(x);
	if ( x == 2 )   { print( "OK" );      }
	else            { print( "FAIL" );    }
	@# break #@
	x = 1;
	while ( x ) {
		break;
	}
	print( "OK" ); }
	@# continue #@
	x = 1;
	while ( x < 5) {
		if ( x == 3 ) { continue; }
	}
	if ( x == 3)    { print( "OK" );      }
	else            { print( "FAIL");     }
}
```

8.2 Test Programs

8.2.1 test-prog1.yolop

```
function main () {

   img cat;
   read_in( cat, "cat1", "jpg" );

   int w; w = dimensions( cat, "width" );
   int h; h = dimensions( cat, "height" );

   pixel p; p = [255,255,255,100];
   int i; int j;

   for (i=0; i<w; i=i+1) {
      for (j=0; j<h; j=j+1) {
         set( cat, i, j, p );
      }
   }

   write_out( cat, "whiteout", "jpg");
}
```

8.2.2 test-prog2.yolop

```
function main () {

    img cat;
    read_in( cat, "cat2", "jpg" );

    int w; w = dimensions( cat, "width" );
    int h; h = dimensions( cat, "height" );

    pixel px;
    int val;
    int i; int j;

    for (i=0; i<w; i++) {
        for (j=0; j<h; j++) {
            px = get(cat, i, j);

                    val = getc( p, "b" );
                    val = val + 50;
                    if ( val > 255 ) { val = 255; }

                    set( px, "b", val );
        }
    }

    write_out(cat, "cool_cat", "jpg");
}
```

8.3 Source files

8.3.1 ast.ml

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | And | Or

type prim = Int | String | Pixel | Image | Void

type var_decl = {vname: string; vtype: prim}

type expr =
    IntLiteral of int
  | StringLiteral of string
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Not of expr
  | ImgPixel of expr * expr * expr * expr
  | ImgImg of expr * expr
  | Readin of string * expr * expr
  | Writeout of expr * expr * expr
  | Get of expr * expr * expr
  | Set of expr * expr * expr * expr
  | Getc of expr * expr
  | Setc of expr * expr * expr
  | Dimension of expr * string
  | Noexpr

type var_init =
    Vinit of var_decl * expr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Break
  | Continue

type var_def =
   VarDecl of var_decl
  | Varinit of var_init

type func_decl = {
    fname : string;
    formals : var_decl list;
    locals : var_def list;
    body : stmt list;
    freturn: prim
  }

type program = var_def list * func_decl list

let string_of_vartype = function
   Int -> "int"
 | Void -> "void"
 | String -> "string"
 | Pixel -> "pixel"
 | Image -> "img"

let rec string_of_expr = function
    IntLiteral(l) -> string_of_int l
  | StringLiteral(l) -> l
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
           Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
      | And -> "&&" | Or -> "||") ^ " " ^
      string_of_expr e2
  | Not(e) -> "!" ^ string_of_expr e
  | ImgPixel(e1, e2, e3, e4) -> "Color(" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^ ", " ^
string_of_expr e3 ^ ", " ^ string_of_expr e4 ^ ");\n"
  | ImgImg(e1, e2) -> "Image(" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^ "Color(\"white\");\n"
  | Readin(e1, e2, e3) -> "read_in(" ^ e1 ^ ", " ^ string_of_expr e2 ^ ", " ^ string_of_expr e3 ^ ");\n"
  | Writeout(e1, e2, e3) -> "write_out(" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^ ", " ^
string_of_expr e3 ^ ");\n"
  | Get(e1, e2, e3) -> "get(" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^ ", " ^ string_of_expr e3 ^
");\n"
  | Set(e1, e2, e3, e4) -> "set(" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^ ", " ^ string_of_expr
e3 ^ ", " ^ string_of_expr e4 ^ ");\n"
```

```
  | Getc(e1, e2) -> "getc(" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^ ");\n"
  | Setc(e1, e2, e3) -> "setc(" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^ ", " ^ string_of_expr e3
^ ");\n"
  | Dimension(e1, e2) -> "dimension(" ^ string_of_expr e1 ^ ", " ^ e2 ^ ");\n"
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

let rec string_of_stmt = function
    Expr(expr) -> string_of_expr expr ^ ";\n"
  | Block(stmts) -> "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n {" ^ string_of_stmt s ^ "}"
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n {" ^
      string_of_stmt s1 ^ "} else\n {" ^ string_of_stmt s2 ^ "}"
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3 ^ ")\n{" ^ string_of_stmt s ^ "}"
  | While(e, s) -> "while (" ^ string_of_expr e ^ ")\n {" ^ string_of_stmt s ^ "}"
  | Break -> "break;\n"
  | Continue -> "continue;\n"

let string_of_vartype = function
   Int -> "int"
 | Void -> "void"
 | String -> "string"
 | Pixel -> "Color"
 | Image -> "Image"

(* Have to edit this so that it allows strings as well *)
let string_of_vdecl id = (string_of_vartype id.vtype) ^ " " ^ id.vname

let string_of_vinit = function
   Vinit(v, e) -> string_of_vdecl v ^ " = " ^ string_of_expr e

let string_of_vdef = function
   VarDecl(v) -> string_of_vdecl v ^ ";\n"
 | Varinit(vi) -> string_of_vinit vi ^ ";\n"

let string_of_fdecl fdecl =
  fdecl.fname ^ " (" ^ String.concat ", " (List.map string_of_vdecl fdecl.formals) ^ ")\n{\n" ^
  (String.concat "" (List.map string_of_vdef fdecl.locals)) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

(* Might need to edit this as well? Dunno. *)
let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdef vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

8.3.2 Makefile

```
TARFILES = Makefile scanner.mll parser.mly ast.mli translate.ml yolop.ml

OBJS = scanner.cmo parser.cmo ast.cmo translate.cmo yolop.cmo

#TESTS =

yolop : $(OBJS)
	ocamlc -o yolop $(OBJS)

#.PHONY : test
#test : yolop testall.sh
#	./testall.sh

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -c $<

%.cmi : %.mli
	ocamlc -c $<


yolop_submit.tar.gz : $(TARFILES)
	cd .. && tar czf src.tar.gz $(TARFILES:%=src/%)

.PHONY : clean
clean :
	rm -f yolop parser.ml parser.mli scanner.ml *.cmo *.cmi *.cmx *.cpp

.PHONY : yolo
```

```
yolo :
	g++ -I../src/lib/Magick++/lib -o *.cpp

# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
translate.cmo: ast.cmo
translate.cmx: ast.cmx
yolop.cmo: scanner.cmo parser.cmi translate.cmo ast.cmo
yolop.cmx: scanner.cmx parser.cmx translate.cmx ast.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
```

8.3.3 parser.mly

```
%{ open Ast %}

%token READ_IN WRITE_OUT SET GET DIMENSION GETC SETC
%token SEMI COMMA QUOTE LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ AND OR NOT
%token PIXEL IMAGE
%token RETURN IF ELSE FOR WHILE BREAK CONTINUE INT STRING
%token FUNCTION
%token <int> ILITERAL
%token <string> SLITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%left AND OR
%right NOT
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS INCR DECR
%left TIMES DIVIDE
```

```
%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [], [] }
 | program vdef { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   FUNCTION ftype_opt ID LPAREN formals_opt RPAREN LBRACE vdef_list stmt_list RBRACE
     { { fname = $3;
          formals = $5;
          locals = List.rev $8;
          body = List.rev $9;
   freturn = $2} }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

ftype_opt:
/*nothing*/ {Void}
| basic_type {$1}

vdef_list:
  /* nothing */ { []}
| vdef_list vdef {$2 :: $1}

vdef:
  vdecl {VarDecl($1)}
| vinit { Varinit($1) }

vdecl:
  basic_type ID SEMI {{vname=$2;vtype=$1}}

vinit:
   basic_type ID ASSIGN expr SEMI { Vinit ({vname = $2; vtype = $1}, $4)}

basic_type:
  INT { Int }
| STRING {String}
| PIXEL {Pixel}
| IMAGE {Image}

formal_list:
   fdef              { [$1] }
  | formal_list COMMA fdef  { $3 :: $1 }

fdef:
   basic_type ID {{ vname=$2; vtype = $1}}

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

stmt:
    expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt   { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt  { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | BREAK SEMI { Break }
  | CONTINUE SEMI { Continue }


expr_opt:
    /* nothing */ { Noexpr }
  | expr         { $1 }

expr:
    SLITERAL        { StringLiteral($1)    }
  | ILITERAL        { IntLiteral($1)       }
  | ID              { Id($1)               }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | NOT expr         { Not($2)            }
  | LBRACK expr COMMA expr COMMA expr COMMA expr RBRACK SEMI { ImgPixel($2, $4, $6,
$8) }
  | LBRACK expr COMMA expr RBRACK SEMI { ImgImg($2, $4)}
  | READ_IN LPAREN ID COMMA expr COMMA expr RPAREN SEMI { Readin($3, $5, $7)}
  | WRITE_OUT LPAREN expr COMMA expr COMMA expr RPAREN SEMI { Writeout($3, $5,
$7)}
  | GET LPAREN expr COMMA expr COMMA expr RPAREN SEMI { Get($3, $5, $7)}
  | SET LPAREN expr COMMA expr COMMA expr COMMA expr RPAREN SEMI { Set($3, $5, $7,
$9)}
  | GETC LPAREN expr COMMA expr RPAREN SEMI { Getc($3, $5) }
  | SETC LPAREN expr COMMA expr COMMA expr RPAREN SEMI { Setc($3, $5, $7) }
  | DIMENSION LPAREN expr COMMA SLITERAL RPAREN SEMI { Dimension($3, $5)}
  | ID ASSIGN expr  { Assign($1, $3)      }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

8.3.4 test_suite.sh

```
#!/bin/sh

YOLOP="./yolop -c"
TEST="../src/tests/test-"

#time limit
ulimit -t 30

make clean
make

for f in "$TEST"*
do
    echo "$f"
```

```
   $YOLOP $f
done
```

8.3.5 translate.ml

```
open Ast

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)
type env = {
   functionDec : string StringMap.t;   (* Signature for each function *)
   globalVars   :  prim StringMap.t;             (* global variables and their types *)
   localVars    :  prim StringMap.t;             (* locals + function params and their types *)
  }

(* Include the necessary headers for the C++ back end *)
let headers = "#include <Magick++.h>\n" ^ "using namespace Magick;\n\n"


(* Get the function signature *)
let fsigToString fdecl =
   fdecl.fname ^ "_" ^ String.concat "_" (List.map Ast.string_of_vdecl fdecl.formals) (* This needs
to map the var type to the name *)


(* Some helper functions *)

let vdecl_of_vinit = function
  Vinit(v, _) -> (v.vtype, v.vname)

(* Get the type and name from a variable definition *)
let getTypeAndName_vdef = function
  VarDecl (v) -> (v.vtype, v.vname)
 |Varinit(vi) -> vdecl_of_vinit vi

(* let fparamString fdecl =
   String.concat "_" (List.map Ast.string_of_vartype (fdecl.formals).vtype) *)

(* take the list of vars/names into a list of pairs (type, name) *)
(* this is for a variable declared *)
let rec enum_vardec = function
     [] -> []
   | hd::tl -> (hd.vtype, hd.vname) :: enum_vardec tl
```

```
(* this is for a variable declared above *)
let rec enum_vardef = function
      [] -> []
    | hd::tl -> ( getTypeAndName_vdef hd) :: enum_vardef tl

let rec enum_func = function
      [] -> []
    | hd::tl -> (fsigToString hd, hd.fname) ::  enum_func tl

(* functions *)
(* val enum : int -> 'a list -> (int * 'a) list *)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

(* variable declarations *)

(* This maps pairs in a graph *)
(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)
let string_map_pairs map pairs =
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

(* Translate the AST into a valid C++ program *)
let yolop_to_cpp (globals, functions) out =

  (* Map the global variables *)
  let global_Vars = string_map_pairs StringMap.empty (enum_vardef globals) in
  let functionDecs = string_map_pairs StringMap.empty (enum_func functions) in

  (* Translate the AST representation of the function*)
  let translate env fdecl =
    let localVars = enum_vardef fdecl.locals
    and formalVars = enum_vardec fdecl.formals in
    let env = { env with localVars = string_map_pairs StringMap.empty (localVars @ formalVars)}
in
    (* get vars *)
    (* Begin translation... *)

    (* Expressions *)
    let rec expr e =
    (match e with
      IntLiteral(i)       ->  string_of_int i
      | StringLiteral(i)    ->  i
      | Id(x)              ->
      if ((StringMap.mem x env.localVars) || (StringMap.mem x env.globalVars))
           then x   (* Check locals *)
      else raise (Failure ("Undeclared identifier:  " ^ x ))
      | Binop(e1,o,e2)      ->
      expr e1 ^ " " ^
         (match o with
            Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
          | Equal   -> "==" | Neq     -> "!=" | Less    -> "<" | Leq     -> "<=" | Greater -> ">" | Geq ->
">="
          | And -> "&&" | Or -> "||") ^ " " ^ expr e2
      | Assign(var, e)      ->
      if ((StringMap.mem var env.localVars) || (StringMap.mem var env.globalVars))
         then var ^ "=" ^ expr e
       else raise (Failure ("Undeclared identifier: " ^ var))
      (* Built-in Functions *)
      (* look for e in local vars and then in globals *)
      | Call("print",[e]) ->  "printf (%s," ^ expr e ^ ")"
      | ImgImg(x,y)        ->  "Image(" ^ expr x ^ "," ^ expr y ^ ", Color(\"white\");"
      | ImgPixel(r,g,b,o)   ->  "Color(" ^ expr r ^ "," ^ expr g ^ "," ^ expr b ^ "," ^ expr o ^ ");"
      | Readin(i,s,e)       ->   i ^ ".read(" ^ expr s ^ "." ^ expr e ^ ");"
      | Writeout(i,s,e)     ->  expr i ^ ".write(" ^ expr s ^ "." ^ expr e ^ ");"
      | Dimension(i,e)      ->  expr i ^ "." ^ (match e with
                                  "height" -> "height();"
                                | "width"  -> "width();"
                                | _         -> raise (Failure("Must be \"width\" or \"height\".")))
      | Get(i,x,y)         ->  expr i ^ ".pixelColor(" ^ expr x ^ "," ^ expr y ^ ");"
      | Set(i,x,y,p)        ->  expr i ^ ".pixelColor(" ^ expr x ^ "," ^ expr y ^ "," ^ expr p ^ ");"
      | Getc(p,c)           ->  expr p ^ "." ^ (match expr c with
                                        "r"    ->  "redQuantum();"
                                      | "g"    ->  "greenQuantum();"
                                      | "b"    ->  "blueQuantum();"
                                      | "o"    ->  "alphaQuantum();"
                                      | _     ->  raise(Failure("Must be 'r' 'g' 'b' or 'o'")))
      | Setc(p,c,v)         -> expr p ^ "." ^ (match expr c with
                                        "r"    -> "redQuantum(" ^ expr v ^ ");"
                                      | "g"    -> "greenQuantum(" ^ expr v ^ ");"
                                      | "b"    -> "blueQuantum(" ^ expr v ^ ");"
                                      | "o"    -> "alphaQuantum(" ^ expr v ^ ");"
                                      | _     ->  raise(Failure("Must be 'r' 'g' 'b' or 'o'")))
      | Call(s, e)         ->  if (StringMap.mem s env.functionDec) (* Look for the function *)
                           then ( s ^ "(" ^ (String.concat "," (List.map
                        expr (List.rev e))) ^ ");")  (* found the function, add the explicit parameters *)
                        else raise (Failure ("Undefined function: " ^ s)) (* didn't find the function *)
      | Not(e)              -> "!" ^ "(" ^ expr e ^ ")"
      | Noexpr               -> "")
    in let rec stmt = function
        Block(stmts)        -> String.concat "" (List.map stmt stmts) ^ "\n"  (* Clean up the statments
*)
      | Expr(e)             -> expr e ^ ";\n"
      | Return(e)            -> "return " ^ expr e ^ ";\n"
      | If(e,s,Block([]))   -> "if(" ^ expr e ^ "){\n" ^ stmt s ^ "}\n"
      | If(e,s1,s2)         -> "if(" ^ expr e ^ "){\n" ^ stmt s1 ^ "}\nelse\n{\n" ^ stmt s2 ^ "}\n"
      | For(e1,e2,e3,s)     -> "for(" ^ expr e1 ^ "," ^ expr e2 ^ "," ^ expr e3^ "){\n" ^ stmt s ^ "}\n"
      | While(e,s)           -> "while(" ^ expr e ^ "){\n" ^ stmt s ^ "}\n"
      | Break               -> "break;\n"
      | Continue             -> "continue;\n"
    in let prim = function
        Int     ->  "int"
      | Void -> "void"
      | String  ->  "string"
      | Pixel   ->  "Color"
      | Image   ->  "Image"

    in let func_params_type = function
        Void -> "void"
      | String  ->  "string"
      | Int -> "int"
      | Pixel -> "Color"
      | Image -> "Image"
    in   (prim fdecl.freturn ^ " " ^ fdecl.fname
          ^ if ((List.length fdecl.formals) != 0)
            then ("("
          ^ func_params_type (List.hd fdecl.formals).vtype ^ " " ^ (List.hd fdecl.formals).vname ^ " "
          ^ String.concat "" (List.map (fun formal -> ", " ^ func_params_type formal.vtype ^ " " ^
formal.vname) (List.tl fdecl.formals)) ^ ")\n{\n")
            else "()\n{\n"
            )
          ^ String.concat "" (List.map Ast.string_of_vdef (List.rev fdecl.locals)) ^ "\n"
          ^ stmt (Block fdecl.body) ^ "\n" ^
          if ((String.compare fdecl.fname "main") == 0)
        then "    return 0;\n}\n"
          else "\n}\n"

    in let env = {
        functionDec = functionDecs;
        globalVars = global_Vars;
        localVars = StringMap.empty } in

    (* Look for main *)

    let _ = try
      (StringMap.find "main" functionDecs)
    with Not_found -> raise (Failure ("Could not find \"main\" function"))

   (* Found main, now finish translation! *)
    in headers ^
      String.concat "" (List.map Ast.string_of_vdef (List.rev globals)) ^ "\n" ^
    (String.concat "\n" (List.map (translate env) (List.rev functions))) ^ "\n"
```

8.3.6 yolop.ml

```
open Printf

type action = Ast | Compile | Error

let out_name = ref "a"

(* Helper function that write "content" into a file named "name" *)
let write fName line =
    let out = open_out fName in
      fprintf out "%s\n" line;
      close_out out

(* note that this means we are passing the full filepath to the compiler *)
let getFileName path =
    let j = (String.rindex path '/') in
    String.sub path (j + 1) ((String.length path) - (j + (String.length ".yolop"
    +1)))

let usage =
    "Usage:\n\n" ^
    "    Compiling to AST Tree        : yolop -a <full filepath>\n" ^
    "    Compiling to C++ to stdin    : yolop -c <full filepath>\n"

let main () =
  let action =
    if Array.length Sys.argv > 2 then
    try
```

```
        List.assoc Sys.argv.(1) [ ("-a", Ast);
                                   ("-c", Compile)]
      with
      _ -> Error
    else Error in
        let lexbuf = ignore(out_name := getFileName Sys.argv.(2));
                     Lexing.from_channel (open_in Sys.argv.(2)) in
        let program = Parser.program Scanner.token lexbuf in
        match action with
          Ast -> let listing = Ast.string_of_program program in
                 write (!out_name ^ ".ast") listing
        | Compile -> let listing = Translate.yolop_to_cpp program !out_name in
                 write (!out_name ^ ".cpp") listing;
        | Error -> print_string usage

let _ = main ()
```

8.3.7 scanner.mll

```
{ open Parser }

let digit     = ['0'-'9']
let int_t     = digit+
let string_r  = '\\'+ | [^ '"']
let string_t  = '"' string_r* '"'

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "@#"    { comment lexbuf }        (* Comments *)
                                    (* Grouping *)
| '('     { LPAREN }
| ')'     { RPAREN }
| '{'     { LBRACE }
| '}'     { RBRACE }
| '['     { LBRACK }
| ']'     { RBRACK }
| ';'     { SEMI   }
| ','     { COMMA  }
                                    (* Binary ops *)
| '+'     { PLUS   }
| '-'     { MINUS  }
| '*'     { TIMES  }
| '/'     { DIVIDE }
| '='     { ASSIGN }
| "=="     { EQ     }
| "!="     { NEQ    }
| '<'      { LT     }
| "<="     { LEQ    }
| ">"      { GT     }
| ">="     { GEQ    }
| "&&"     { AND    }
| "||"     { OR     }
| '~'      { NOT    }
                                    (* Control Flow *)
| "if"     { IF     }
| "else"   { ELSE   }
| "for"    { FOR    }
| "while"  { WHILE  }
| "return" { RETURN }
                                    (* Primitives *)
| "int"    { INT    }
| "string" { STRING }
| "pixel"  { PIXEL  }
| "image"    { IMAGE  }
                                    (* Reserved *)
| "function"    { FUNCTION  }
| "read_in"     { READ_IN   }
| "write_out"   { WRITE_OUT }
| "get"         { GET       }
| "set"         { SET       }
| "getc"        { GETC      }
| "setc"        { SETC      }
                                    (* Literals *)
| int_t    as lxm { ILITERAL(int_of_string lxm) }
| string_t as lxm { SLITERAL(lxm) }

| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "#@" { token lexbuf }
| _    { comment lexbuf }
```

9 Acknowledgements

Special thanks to Emad Barsoum for his source files listed on the course website. They were immensely helpful as his implementation was similar to ours. Also special thanks to Magick++ as the backbone of our image processing.