Fall - 2013

Simplified Image Processing (SIP)

Final Report

By :-
Shubhanshu Yadav(sy2511)
Vaibhav Jagannathan(vj2192)
Bhargav Sethuram(bs2814)

Contents

1. INTRODUCTION

1.1 BACKGROUND

We aim to create a simple, easy to use, language that will provide people with the means to process images. People may sometimes find it hard to abstract the various components of the image and can often get tangled with the gory details related to the images, instead of focusing on the work that they are trying to accomplish with those images. Hence, the language is designed to be intuitive in nature, making it easy to be used by people having basic knowledge of image processing.

Image Processing involves many stages, right from image acquisition, sampling, storing to processing and rendering. We are motivated to work on that part of Image Processing which involves the application of various transforms and computations in order to have the desired effect on a digital image.

1.2 PROJECT OVERVIEW

Our Language is actually a programming language for image processing. With our Language, a user can perform computations and transforms on images and also define the relationship between several components of the image in a convenient way. Also, the user can verify his/her speculations on the images and witness how the other attributes will change as certain components change.

We keep most of the data types in C++ like int, float, bool, string and define basic image types Pixel and Image, so that all the images and image operations can be composed from these two types. These additional data types were chosen because of two reasons. One, all the point processing algorithms (for example contrast stretching, histogram normalization etc) can be performed efficiently if one has knowledge of the layout of the pixels in the images. This is because all the point processing algorithms modify the image by performing their operations directly on the pixels contained in the images. Apart from the point processing algorithms, there is another class of image processing algorithms. These image processing algorithms require the spatial information of the pixels contained in the image. These image processing algorithms can also be defined very easily in our language. Therefore, it will be very easy for a user with limited programming skills to get the answer he/she wants to an image processing problem. Also, one can develop some complicated algorithms with the control structure provided in the language.

Our compiler does not actually compile the source code into binary code, it translates the SIP source code into a correct C++ code, and we will get the executable file through the C++ compiler. The process to translate SIP code to C++ code requires the collaboration of scanner, parser, ast, semantics checking, and code generation. Furthermore, to make the process more reliable, we also built numerous test cases. These test cases were aimed at exhaustively testing our language and ensuring that it produces the correct c++ code. We will discuss the details of these in the following parts.

1.3 Language Features

- SIP is not restrictive in terms of algorithms that can be used with images. The general nature of the language allows for the implementation of simple as well as increasingly complex algorithms that can be applied to images.
- There is no main in our language. The execution starts from the very first executable statement that it can find and then proceeds in order, until it reaches the end of program.
- There are many different ways through which user can access the images and sub-images. One of the ways of accessing the sub-images is to specify the range.
- The language will judge whether a value for a certain image type is valid or not. It will also check for valid pixel variable declarations.
- The same operator could be used by many data types.
- The same function could be used by various input arguments. For example, if the inbuilt display function can be used to send output to console as well as display the images.

2. Language Tutorial

2.1 Getting Started with the Compiler

Before installing our compiler, configure your environment first as in Section 4-5-Environment (including download and install ‘opencv’);

Also, ‘SIP Compiler’ will compiler the SIP source code into executable codes, with the help of c++ compiler ‘g++’. You should also install ‘g++’ in your system, ‘g++4.7.0’ or above is required.

2.2 Installing and Compiling the Compiler

After the procedure above, you need to put our project folder ‘SIP’ into your file system. These are source files. You need to compile the ‘SIP Compiler’ first, and then use it to compile SIP programs.

To compile the ‘SIP Compiler’, go into the project folder, and then use the following commands to build:

make clean

make

After doing so, it will produce an executable file named ‘svipc’. Then use ‘svipc’ to compile your SIP program with the following commands:

./svipc [-option] inputfilename

Where,

Inputfilename is the path to your SIP source code file.

Options :-

-c -> This option will compile the SIP source code file and will produce an executable of the inputfilename.

–t -> This option will check the SIP source code for the syntax and semantic errors. This is mainly used for testing the source code.

No Option specified -> If no option is specified, it will default to –c option.

For example, if we have a source code file name "*test.svip*" , then we can get a executable like this :-

./svipc –c test.svip

Or making use of the default behavior :-

./svipc test.svip

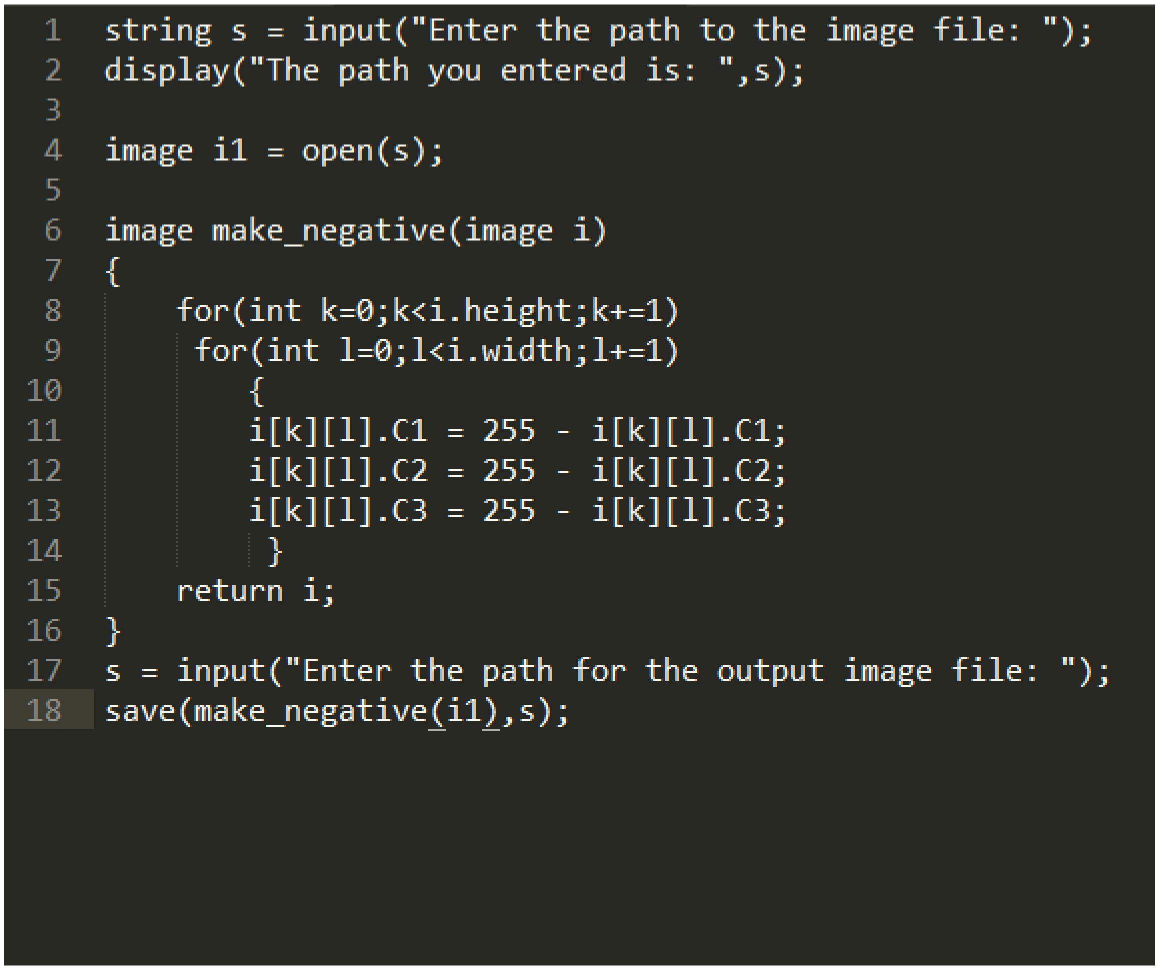
After the program compiles successfully, the program can be run in the following manner :-

./test

Note: *inputfilename* is the file name of SIP program. An output file name of same name as the input file is generated. Although you only need to use the executable file, the compiler will generate intermediate c++ code in the file '*inputfilename*.cpp'.

2.3 A First Example of SIP program

2.3.1 SIP Program(sample_prog1.svip)



```
1   string s = input("Enter the path to the image file: ");
2   display("The path you entered is: ",s);
3
4   image i1 = open(s);
5
6   image make_negative(image i)
7   {
8       for(int k=0;k<i.height;k+=1)
9        for(int l=0;l<i.width;l+=1)
10          {
11          i[k][l].C1 = 255 - i[k][l].C1;
12          i[k][l].C2 = 255 - i[k][l].C2;
13          i[k][l].C3 = 255 - i[k][l].C3;
14           }
15      return i;
16  }
17  s = input("Enter the path for the output image file: ");
18  save(make_negative(i1),s);
```

This is an SIP program which takes an image as an input and outputs a file which is the negative of the input image.

Line-1 : Declares a string variable s which takes as input the path to the image file.

Line-2 : Outputs to the console, the path that the user enters.

Line-4 : Declares an image variable i1, which will load the image from the path specified in the s variable.

Line 6-16: Declares a function make_negative, which takes as input an image and outputs an image. Inside the body of the function, we have two for loops which basically access each pixel of the image. The negative operation is performed and the negative image is returned.

Line 17-18 : These lines basically take the output file path from the user and saves the negative image to that path.

2.3.2 Compile the program

To compile the program, use the following command:

./svipc sample_prog1.svip

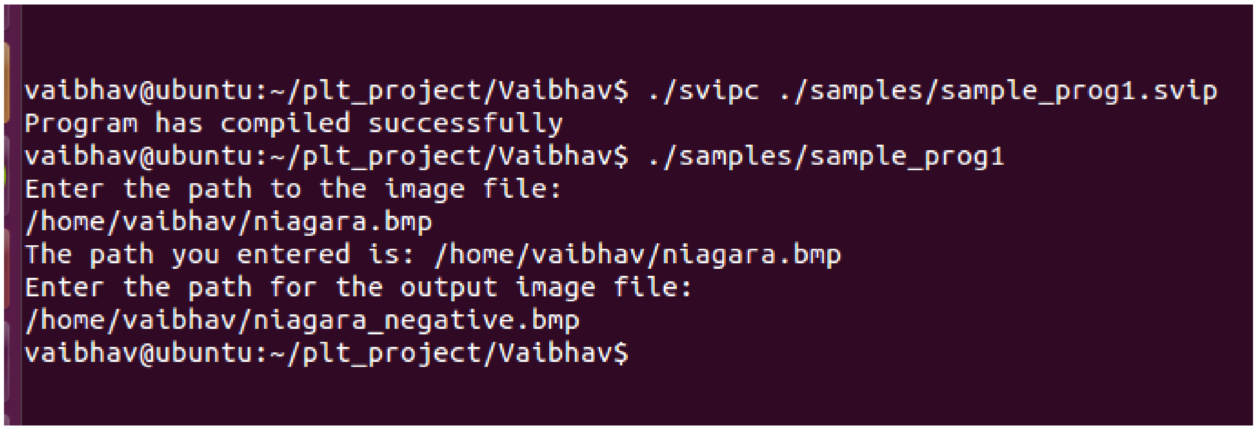
2.3.3 Running the program

Then use the following command to execute sample_prog1:

./sample_prog1

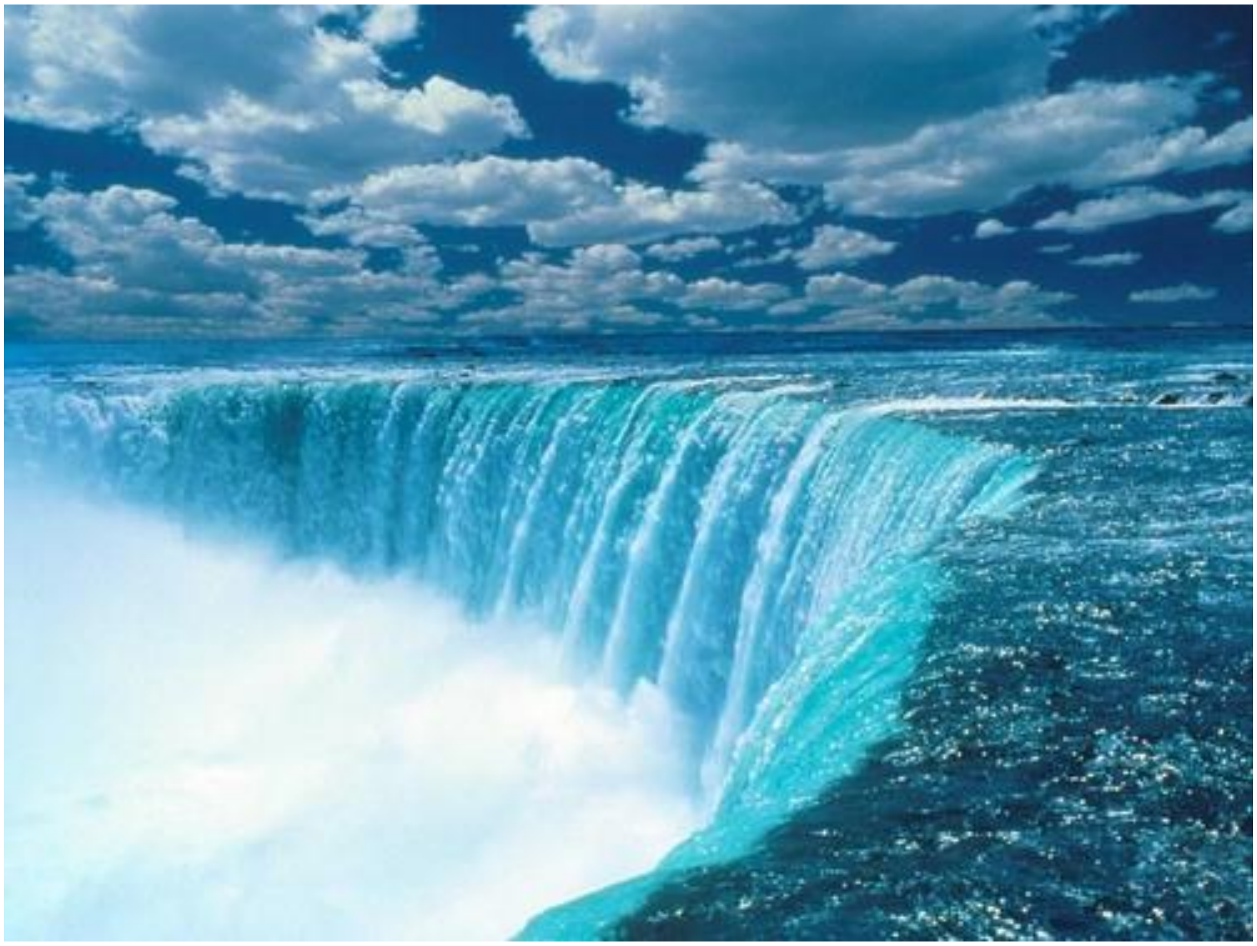
2.3.4 Result

The output of the above program is as below:



```
vaibhav@ubuntu:~/plt_project/Vaibhav$ ./svipc ./samples/sample_prog1.svip
Program has compiled successfully
vaibhav@ubuntu:~/plt_project/Vaibhav$ ./samples/sample_prog1
Enter the path to the image file:
/home/vaibhav/niagara.bmp
The path you entered is: /home/vaibhav/niagara.bmp
Enter the path for the output image file:
/home/vaibhav/niagara_negative.bmp
vaibhav@ubuntu:~/plt_project/Vaibhav$
```

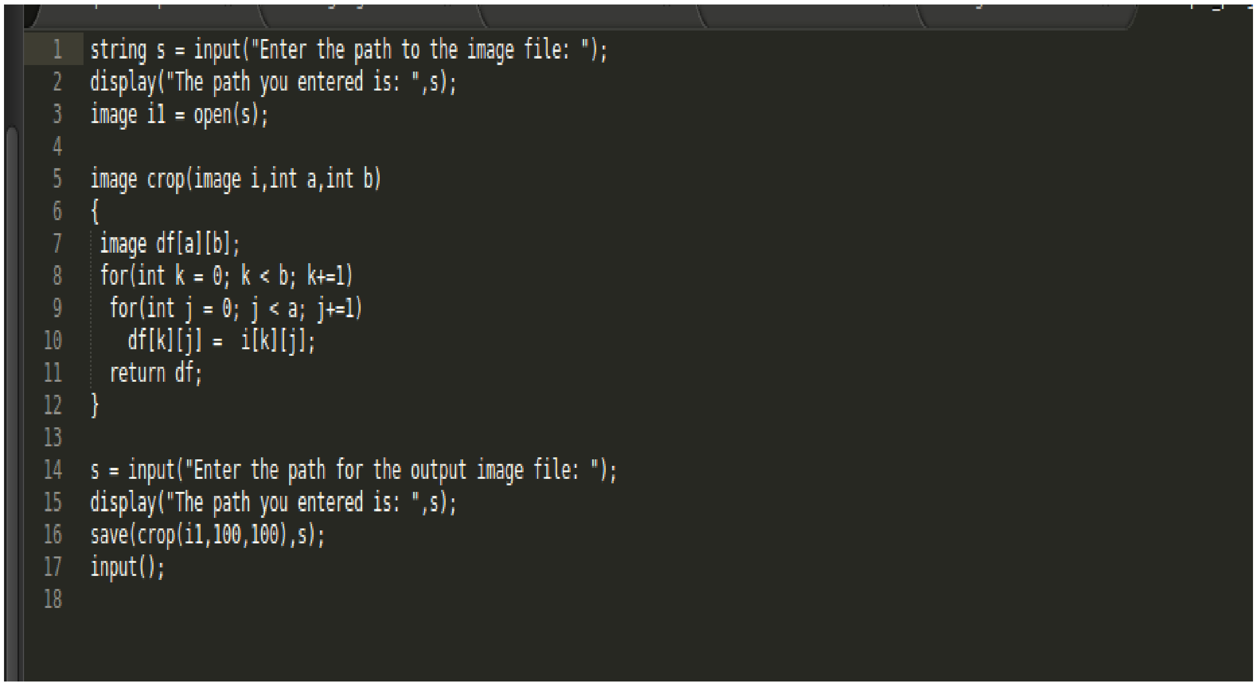
The input sample image file given was the following :-



The output image obtained was the following :-

2.4 Additional Examples

2.4.1.1 SIP Program



```
1  string s = input("Enter the path to the image file: ");
2  display("The path you entered is: ",s);
3  image i1 = open(s);
4
5  image crop(image i,int a,int b)
6  {
7   image df[a][b];
8   for(int k = 0; k < b; k+=1)
9    for(int j = 0; j < a; j+=1)
10     df[k][j] =  i[k][j];
11   return df;
12 }
13
14 s = input("Enter the path for the output image file: ");
15 display("The path you entered is: ",s);
16 save(crop(i1,100,100),s);
17 input();
18
```

Line-1 : Declares a string variable s which takes as input the path to the image file.

Line-2 : Outputs to the console, the path that the user enters.

Line-3 : Declares an image variable i1, which will load the image from the path specified in the s variable.

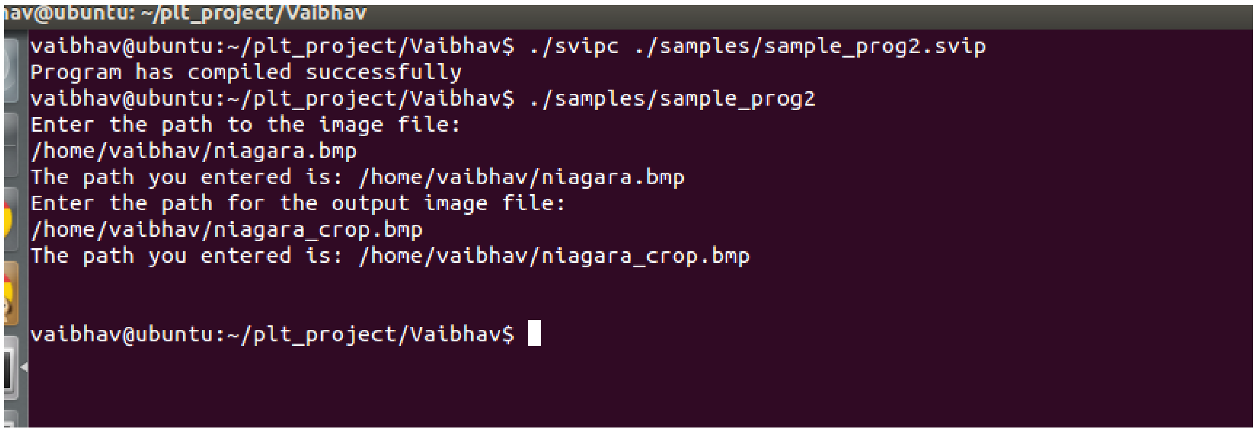
Line 5-12: Declares a function crop, which takes as input an image and outputs an image. Inside the body of the function, we have two for loops which basically access each pixel of the image. The crop function is performed and the cropped image is returned.

Line 14-16 : These lines basically take the output file path from the user and saves the croppped image to that path.

Line 17: This line is basically telling the execution to wait for the input and then, after the input is obtained proceed forward. This statement is used here so that we can see the output obtained by the execution.

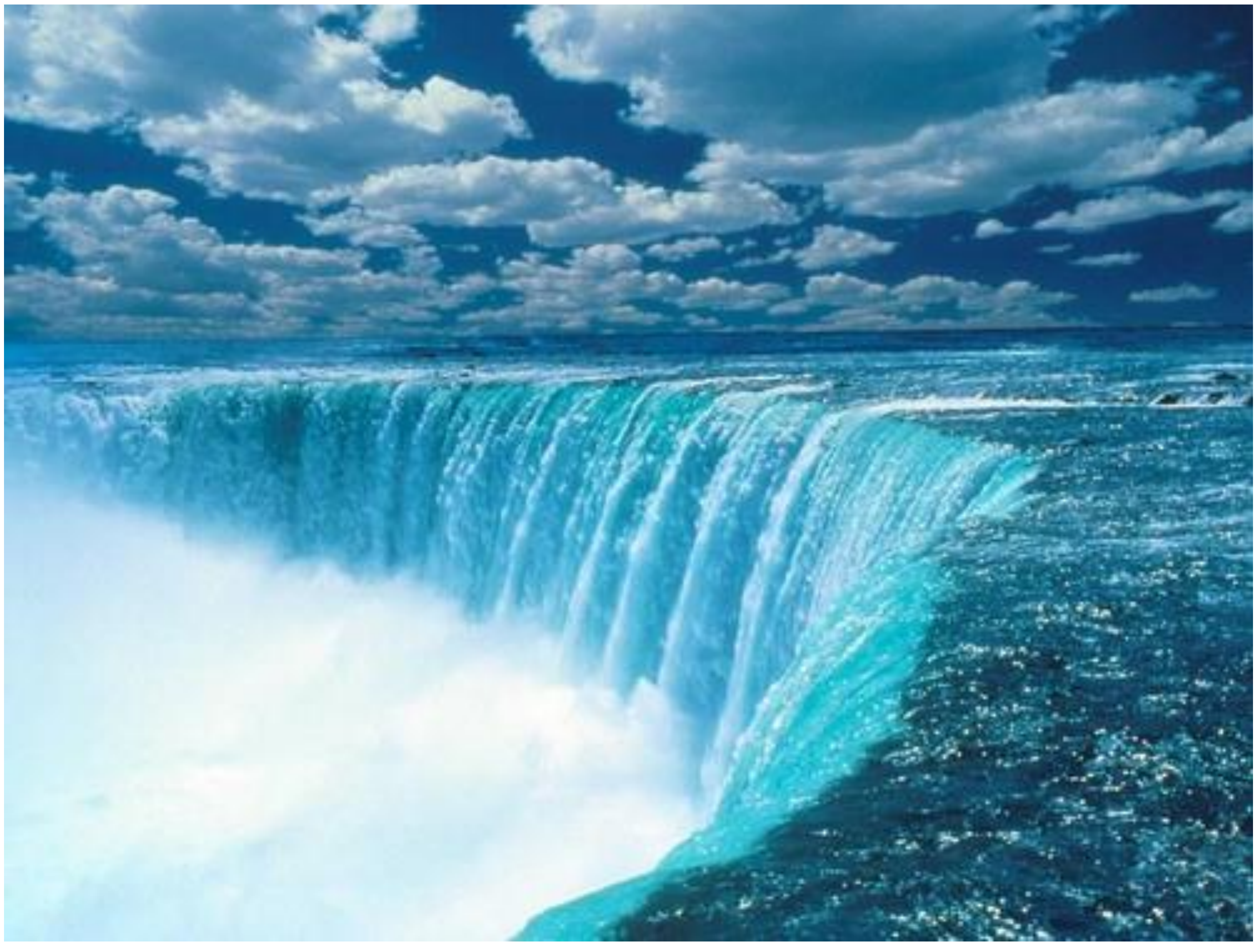
2.4.1.2 Result

The result of the program is as follows :-

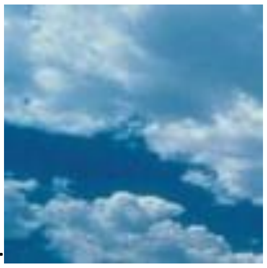


```
vaibhav@ubuntu:~/plt_project/Vaibhav$ ./svipc ./samples/sample_prog2.svip
Program has compiled successfully
vaibhav@ubuntu:~/plt_project/Vaibhav$ ./samples/sample_prog2
Enter the path to the image file:
/home/vaibhav/niagara.bmp
The path you entered is: /home/vaibhav/niagara.bmp
Enter the path for the output image file:
/home/vaibhav/niagara_crop.bmp
The path you entered is: /home/vaibhav/niagara_crop.bmp


vaibhav@ubuntu:~/plt_project/Vaibhav$
```

The input sample image given was the following :-

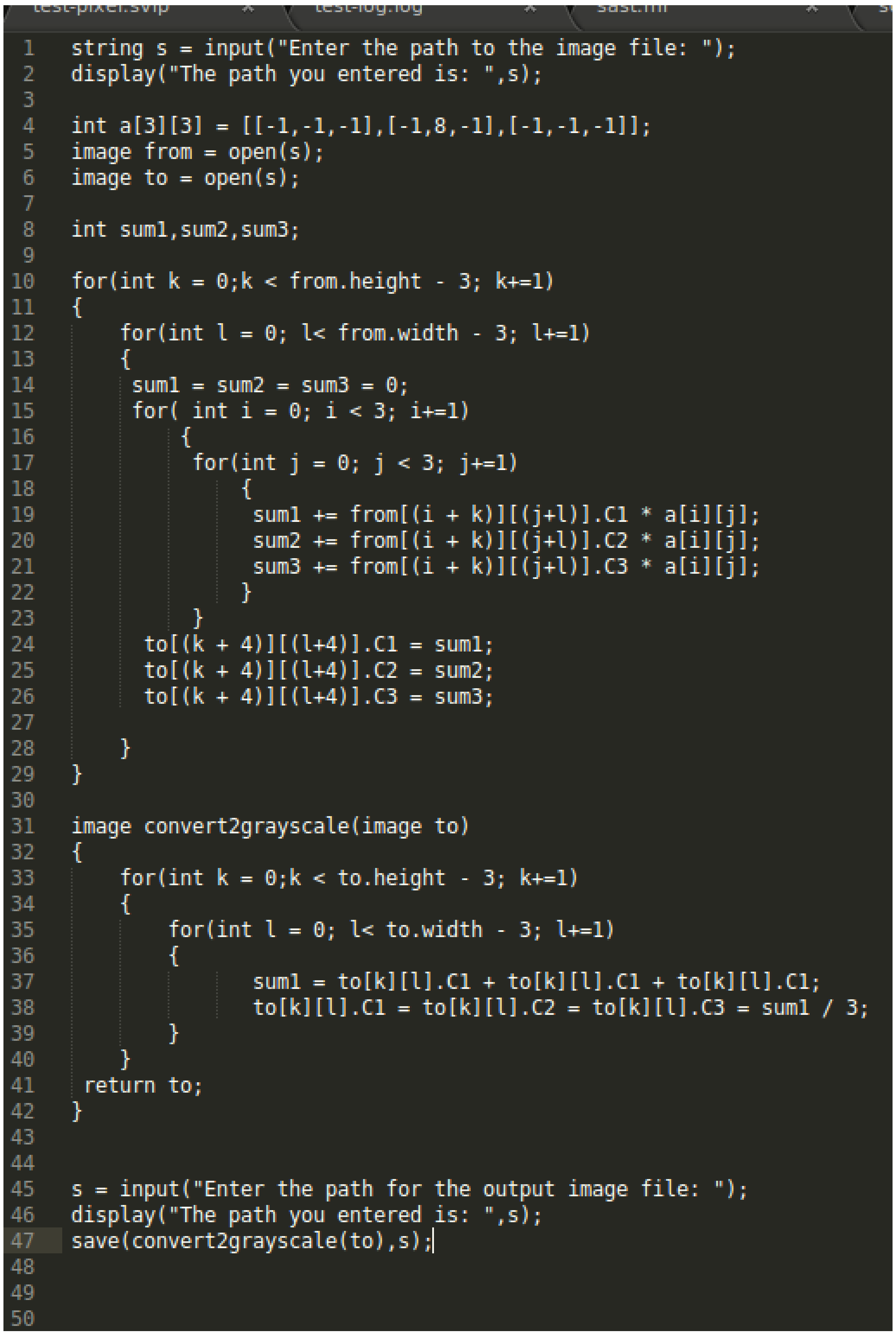


The output image obtained was the following :-

-

2.4.2.1 Writing Complex Image Manipulation Algorithms

2.4.2.2 SIP Program



```
string s = input("Enter the path to the image file: ");
display("The path you entered is: ",s);

int a[3][3] = [[-1,-1,-1],[-1,8,-1],[-1,-1,-1]];
image from = open(s);
image to = open(s);

int sum1,sum2,sum3;

for(int k = 0;k < from.height - 3; k+=1)
{
    for(int l = 0; l< from.width - 3; l+=1)
    {
     sum1 = sum2 = sum3 = 0;
     for( int i = 0; i < 3; i+=1)
         {
          for(int j = 0; j < 3; j+=1)
             {
              sum1 += from[(i + k)][(j+l)].C1 * a[i][j];
              sum2 += from[(i + k)][(j+l)].C2 * a[i][j];
              sum3 += from[(i + k)][(j+l)].C3 * a[i][j];
             }
          }
      to[(k + 4)][(l+4)].C1 = sum1;
      to[(k + 4)][(l+4)].C2 = sum2;
      to[(k + 4)][(l+4)].C3 = sum3;

    }
}

image convert2grayscale(image to)
{
    for(int k = 0;k < to.height - 3; k+=1)
    {
        for(int l = 0; l< to.width - 3; l+=1)
        {
               sum1 = to[k][l].C1 + to[k][l].C1 + to[k][l].C1;
               to[k][l].C1 = to[k][l].C2 = to[k][l].C3 = sum1 / 3;
        }
    }
 return to;
}


s = input("Enter the path for the output image file: ");
display("The path you entered is: ",s);
save(convert2grayscale(to),s);
```

Line 4 : This defines a 3x3 mask (represented as 2D arrays) which is used to perform the edge detection operation on the image.

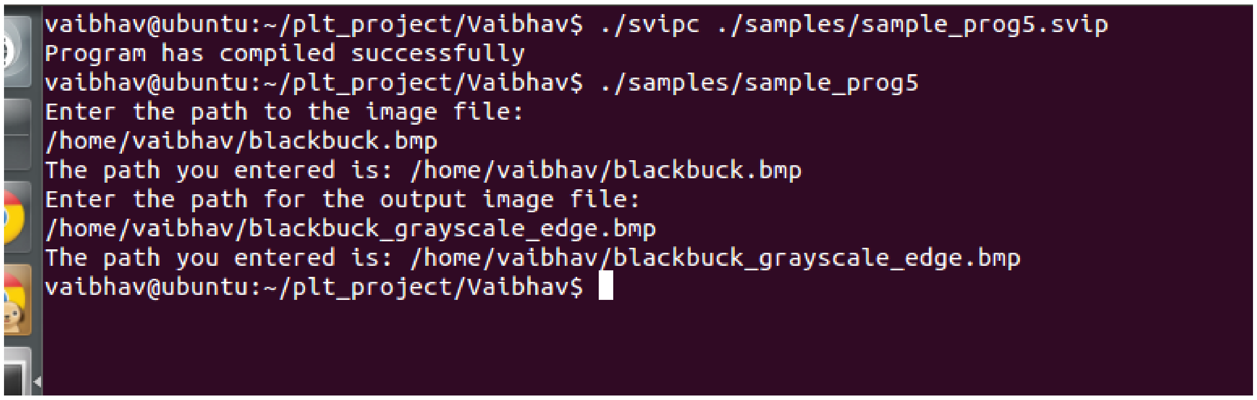
Lines 5-9 : Loads the image given by input path into the variables. Also declares the other variables to be used.

Lines 10-29: Performs the image detection operation on the image. In these lines, basically the mask is applied to each 3x3 subarray of the image and the result is written back to the image variable to.

Lines 31-42: Defines a function called convert2grayscale which takes as input an image and converts it from an rgb image to a grayscale image. A grayscale image is one which only consists of pixel intensity values. The function returns this grayscale image.

Lines 45-47: Takes as input a path, where the grayscale applied edge detected image will be stored.

2.4.2.3 Console Output

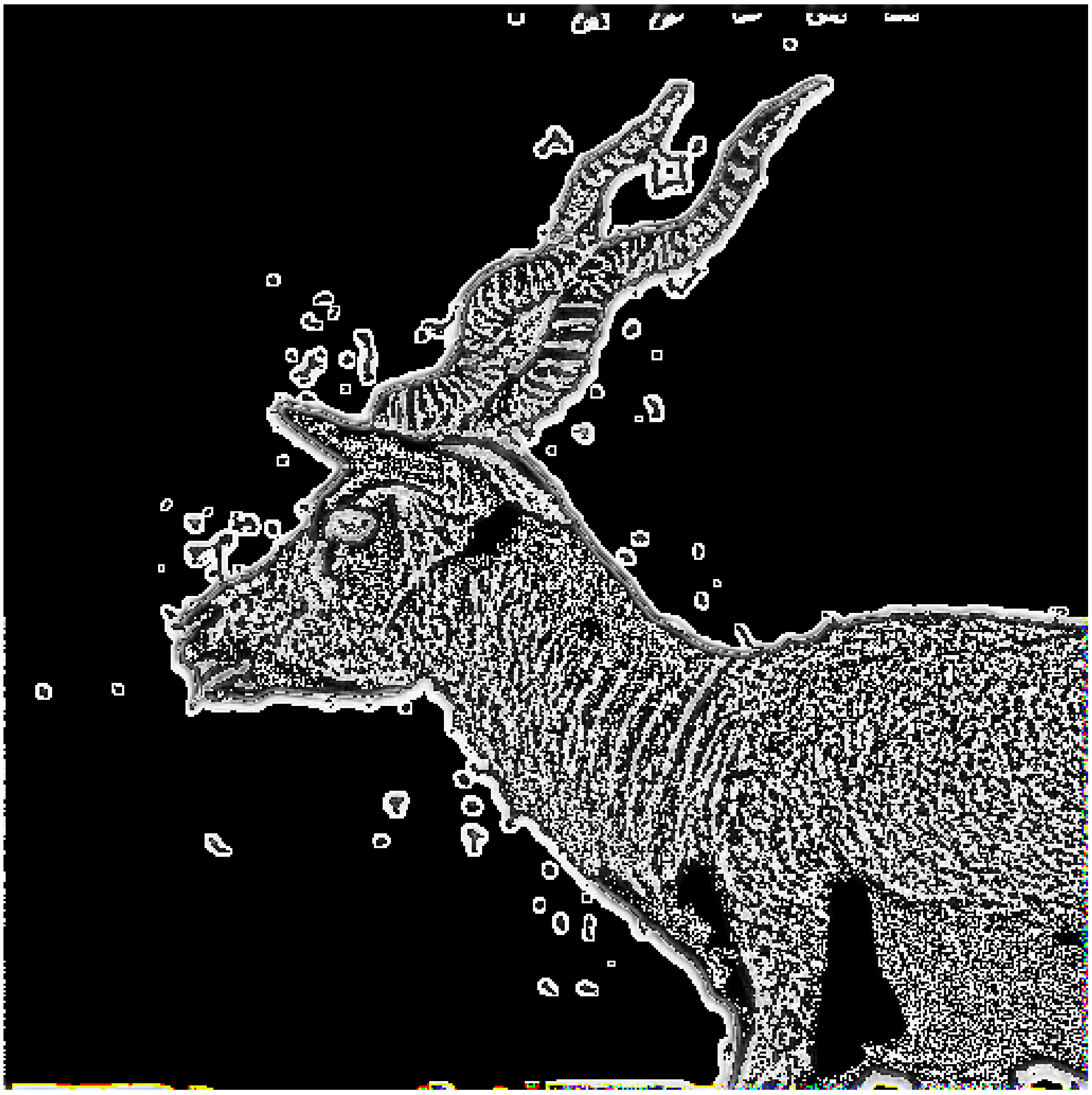


```
vaibhav@ubuntu:~/plt_project/Vaibhav$ ./svipc ./samples/sample_prog5.svip
Program has compiled successfully
vaibhav@ubuntu:~/plt_project/Vaibhav$ ./samples/sample_prog5
Enter the path to the image file:
/home/vaibhav/blackbuck.bmp
The path you entered is: /home/vaibhav/blackbuck.bmp
Enter the path for the output image file:
/home/vaibhav/blackbuck_grayscale_edge.bmp
The path you entered is: /home/vaibhav/blackbuck_grayscale_edge.bmp
vaibhav@ubuntu:~/plt_project/Vaibhav$
```

2.4.2.4 Image Results

The input image was taken as the following :-

The output obtained was the following :-



3. Language Reference Manual

3.1 Introduction

Simplified Image Processing (SIP) is a programming language for image processing. With SIP, user can perform operations on images.

SIP programs are first compiled into “C++” modules and will be further compiled into machine binary by C Language compiler.

3.2 Lexical conventions

The following lexical elements are part of our compiler :

- Tokens
- Comments
- Identifiers
- Keywords
- Punctuators
- Operators
- Literals

Tokens are the smallest elements of our language that the compiler recognizes. They are separated using one or a combination of blank spaces, horizontal tabs, new lines or comments.

Tokens are separated by one or more of the following: spaces, tabs or newlines. A token is also the longest sub-string of characters that is accepted as legal by our compiler. For example, if there following two characters in the input buffer are “+” and “=”, we will get the token from “+=” rather than “+” and “=” separately.

The different type of tokens in our language are mentioned below in a little more detail.

3.2.1 Comments

A comment can start with “//”. For example, this would be considered a comment in our language:

// example comment

A comment can also span on multiline. For example, if one of the line has “/*” in it, it will comment out everything until it encounters the matching “*/”.

*/*example*

/ MultilineComment */*

Svip code

**/*

We also have the **nested comments** feature. Basically, this implies that every “/*” should have a corresponding “*/” and everything between “/*” and “*/” will be commented out.

3.2.2 Identifiers

An identifier is a combination of one or more letters and/or digits and the underscore character (_). It is used to represent an object of a particular data type or is the name of a function. No other special character can be a part of the identifier. They must start with a letter and are case sensitive.

The following characters are legal as the first character of an identifier, or any subsequent character:

_ a b c d e f g h i j k l m

n o p q r s t u v w x y z

A B C D E F G H I J K L M

N O P Q R S T U V W X Y Z

The following characters are legal as any character in an identifier except the first:

0 1 2 3 4 5 6 7 8 9

3.2.3 Keywords

Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program. The following keywords are reserved for **SIP** :-

int float string bool void true false pixel image break continue if else for do while in with return

Here, int, float, string, bool, pixel and image represent data types and the rest are keywords that function similar to their counterparts in C++.

3.2.4 Constants

SIP allows constants of any of the types discussed above.

3.2.4.1 Integer(int)

Integer constants consist of decimal integers as a signed sequence of one or more digits.

For example:

int x = 15;

3.2.4.2 Float(float)

A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing. Every floating constant is taken to be double precision.

For instance, float number can be like:

0.125

1.25

123.33542000

While these numbers are **illegal**:

1..2 (More than one decimal)

.123 (integer part missing)

3.2.4.3 pixel

We have decided to extend the concept of constant to all of our data types. A constant pix is associated with a particular color and a format which is in the **RGBA** format. Hence a constant pixel has a particular color and format defined.

3.2.4.4 image

A constant image follows the same concept. An image is associated with a file and is interpreted as a 2D array of pixels.

3.2.4.5 String constants

A string is a sequence of characters(‘a’-’z’, ‘A’-’Z’, ‘0’-’9’) surrounded by double quotes ‘‘ " ’’. "This is a string" defines a string in ALG. Two " must use together, none of them could be missing.

3.3 Data Types

As mentioned earlier, the identifiers are either function names or related to one of the data types defined in SIP. The interpretation of an identifier by the compiler is thus dependent on the data type. The following types are supported:-

3.3.1 int

Any integer within the range – (2^31) to ((2^31)--1) or 2147483648 to 2147483647 can be stored in an int type. The size associated with an int attribute is 2^32. If not initialized, the default value considered is 0.

3.3.2 float

64-bit signed decimal point numbers are denoted by type float. Default value is 0.0.

3.3.3 pixel

This data type is the basic unit of an image – a pixel. It consists of four components - C1, C2, C3 and alpha. C1, C2, C3 basically provide us with the value of red channel, green channel and the blue channel of the pixel respectively. For the first three components, the values between 0 and 255 would be sufficient to discern the color. The final component – alpha can also have values from 0-255. By default, this value is set to 255.

The alpha channel is normally used as an opacity channel. If a pixel has a value of 0 in its alpha channel, it is fully transparent (and, thus, invisible), whereas a value of 255 in the alpha channel gives a fully opaque pixel (traditional digital images). Values between 0 and 255 make it possible for pixels to show through a background like a glass (translucency), an effect not possible with simple binary (transparent or opaque) transparency. It allows easy for image compositing.

PNG is an image format that uses RGBA.

The user can access these components using the dot notation. For example, for a pixel p:-

p.C1 will give us the red channel value of the pixel.

p.C2 will give us the green channel value of the pixel.

p.C3 will give us the blue channel value of the pixel.

3.3.4 image

This is essentially a 2-dimensional matrix of pixels. This data type makes it easy to perform functions such as adding a group of pixels to a pre-existing image.

It has two components : width and height. These can be accessed using the dot notation. For example, for an image i :-

i.width -> will give us the width of the image

i.height -> will give is us the height of the image.

We can access the individual pixels by using the following: ***image_name[row_value][column_value].***

Continuing the above example, we can access the pixel in row 2 and column 3 by using the following :-

i[2][3]

This value can be assigned either to a pixel data-type or can be treated as a regular pixel on its own. This means that the properties of the pixel applies to these as well.

For example,

i[2][3].C1 -> will give us the red channel value of the pixel at row 2, column 3 in image i.

i[2][3].C2 -> will give us the green channel value of the pixel at row 2, column 3 in image i.

i[2][3].C3 -> will give us the blue channel value of the pixel at row 2, column 3 in image i.

The user also has an option to **retrieve the subset of any given image variable.** This is, in turn, equivalent to obtaining a cropped version of that image. The syntax for that is as follows :-

i[2:3][10:22]; //this will return an image that consists of rows 2 to 3 and columns 10 to 22 of image i

The user can also **obtain a single row or single column** with the above syntax. For example,

i[2][10:22]; // This will return columns 10 to 22 of row 2 of image i.

i[3:10][2]; // This will return rows 3 to 10 of column 2 of image i.

3.3.5 User defined

Besides the basic types the user can also define the following: arrays, functions.

3.3.5.1 Arrays

Users can define a collection of elements of the same type by declaring arrays. Only arrays of type int are allowed. Our language allows for creation of 1D, 2D and 3D arrays. Arrays ensure contiguous memory allocation for the elements within the array. The general syntax for declaring arrays is given below:

type variablename[size]

This statement allocates memory of size equal to n times the size of data_type name[n]

This statement allocates memory of size equal to n times the size of 'data_type' and uses 'variablename' to refer to that memory location.

Examples,

1D – Arrays :-

int arr[5];

2D – Arrays :-

int arr[5][6];

3D – Arrays :-

int arr[5][3][7];

3.3.5.2 Functions

Functions in sVIP have the following syntax for definition:

returntype functionname (args-list) {statements;}

Here, returntype is a valid sVIP data type, functionname is a valid identifier according to the sVIP identifier rules, and args-list should be a valid comma separated list of identifiers. The args-list may be left empty. The function body may be empty or can contain sVIP statements and expressions.

3.4 Conversions

Lower numerical types can be expanded into higher ones. Example: A type int can be assigned to a type float but not vice-versa. When the int is assigned to float, the int is converted to float data format.

3.5. Expressions

3.5.1 Primary expressions

A primary expression can be an identifier, any of the constants defined above, an expression contained in parentheses.

3.5.2 Unary operators

SIP supports one unary operator, that of negation. It is denoted by a "-" sign, which, when placed before an expression negates the expression. Only int and float and bool can be negated.

Negation of numeric values means multiplying the value by -1. Negation of bool is basically performing the not operation.

For example,

int x = -32; // defines the number -32

bool b = -true; // will evaluate to true

In order for the user's convenience, we have also provided the "!" as an alternative. For example,

bool b = -true;

bool c = !truel; //will be the same as bool b

3.5.3 Multiplicative operators

The multiplicative operators *, /, and % group left to right.

3.5.3.1 Multiplication (expression*expression)

The result is the product of two expressions. If both expressions are integers, the result is an integer. If both expressions are float, the result is float. If one expression is integer and the other one is float, then the integer is converted into float and the result is float.

No other combination is allowed.

3.5.3.2 Division (expression/expression)

The binary “/” operator indicates division. The same type considerations as for multiplication apply.

3.5.3.3 Modulus (expression%expression)

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

No other combinations are allowed.

3.5.4 Additive operators

The additive operators + and - group left to right.

3.5.4.1 Addition/Subtraction

An int can be added to an int or float. A float can be added to a float and int. The result of addition/subtraction involving float and int is a float. Additionally, we can also perform addition operations on the image along with the pixels. So we can do pixel+ pixel. Pixel subtraction is a process whereby the digital numeric value of one pixel is subtracted from another pixel. This is primarily done for one of two reasons – levelling uneven sections of an image such as half an image having a shadow on it, or detecting changes between two images

3.5.5 Logical Operators

“*” corresponds to logical AND. “+” corresponds to logical OR. “!‟ corresponds to logical NOT. “*” and “+” are binary operators whereas “!” is a unary operator. None of the logical operators are short circuiting.

3.5.6 Relational Operators

==, !=, < , <= , >, >= are all binary operators.

3.6. Type-specifier

Type-specifiers in SIP have similar types with C++ as well as its own types. They are:

- int
- Boolean
- string
- float
- void
- image
- pixel

3.7 Function Definitions

The type-specifier of a function specifies the type of return value from the function. The return type of the function can be each of the above types in 3.6. However, every function can only have one certain return value. The decl part consists of a function name, arguments of the function and body of the function. A specific format of a function declaration is:

type-specifier fname (farg-list) {fbody}

where,

fname : is an identifier for the function which identifies the name of the function;

farg-list: contains zero or more arguments, and each argument has its own type-specifier and its name, different arguments are separated by ",";

fbody: implements the function, which consists of statements and variable declarations. Particularly, in fbody, all the variable declarations appear before the statements. And no variable declarations are allowed among statements.

A common example of function declaration is:

image i1 = open(s);

image make_negative(image i)

```
{
        for(int k=0;k<i.height;k+=1)
         for(int l=0;l<i.width;l+=1)
                {
                i[k][l].C1 = 255 - i[k][l].C1;
                i[k][l].C2 = 255 - i[k][l].C2;
                i[k][l].C3 = 255 - i[k][l].C3;
                 }
        return i;
}
```

3.8 Variable Declaration

Each variable has its own type as its type-specifier indicates. The decl part the variable declaration contain one identifier.

Here is the example of the format:

type-specifier identifier1;

Common examples of variable declaration are:

```
int a ;
image i;
string b,c,d,e;
```

3.9. Statements

Expressions followed by semi colons are statements in SIP. They are executed in sequence.

3.9.1 Selection Statements

Selection statements evaluate conditions and direct control flow appropriately.

if (expression) statement-block

if (expression) statement-block else statement-block

3.9.2 For Loops

A valid for statement form is:

for (expression-statement; expression-statement;expression- statement) statement-block

The first statement is evaluated before the loop begins, the second expression is evaluated at the beginning of each iteration and, if false, ends loop execution. The third statement is evaluated at the end of each iteration. Each expression can be multiple expressions separated by commas.

3.9.3. Break

The break statement allows the termination of the current for loop and takes execution to the statement immediately after the for loop.

3.9.4 Continue Statement

The continue statement can be used only within a for loop. When encountered, the remaining part of the for loop is ignored and the iteration execution goes to the condition evaluation of the for loop, possibly for the next iteration.

3.9.5 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.

3.10. Scope

3.10.1 Static Scoping

SIP uses static scoping. That is, the scope of a variable is a function of the program text and is unrelated to the runtime call stack. In SIP, the scope of a variable is the most immediately enclosing block, excluding any enclosed blocks where the variable has been re-declared.

3.10.2 Global vs. Local

Global variable: The variables declared outside of the function are global variables, which will be applied in the whole program except the function where there is a local variable with the same name as that of the global variable. Global variables will exist until the program terminates.

Local variable: The variables declared inside of the function are local variables, which will exist and be applied only inside that function.

Scope conflicts: Since we are using static scoping, whenever there are variables with same name in local scope as well as global scope, preference is given to local scope.

3.10.3 Forward Declarations

SIP requires forward declarations for variables and functions. That is, a variable needs to be declared before it can be referenced, and any function needs to be defined before it can be invoked.

For example, SIP generally prohibits the following and will throw an error:

float a; float b; float mean;

mean = func(a, b);

...

In this case, the function func() needs to be defined before it is called.

3.10.4 Arithmetic Operator Overloading

Arithmetic operators (+, -, *, /) are overloaded in SIP. They can be used in expressions where integers and floats are mixed, and where an image/filter is mixed with a scalar value.

3.10.5 Function Name Overloading

SIP does not allow function name overloading. That is, each function should have a unique function name, or SIP compiler will complain.

3.11 In-built Functions

3.11.1 display()

This function is one of the few instances in SIP where the function overloading was implemented.

display(string,int,float,….) will print a concatenated version of the arguments passed to it on the console.

display(image) will show the image in a new window.

The return type of display() is void. This forbids the display() from being used in assignments.

Display function explicitly forbids the user from mixing the image with any other data type. In order to display the image, we used an external library known as **cImg.** We had to do this because because we cannot output the image to a console window. And furthermore, to make the image display we would have had to write our own graphics library.

3.11.2 input()

This is the only other over-loaded function in the SIP. The return type of this function depends on the usage. This function takes an input from stdin and assigns it to a variable. For example,

String s = input(); //assigns the input eneterd by user in the console window to s

int x = input(); // *assigns the input eneterd by user in the console window to x*

float y = input(); //assigns the input eneterd by user in the console window to y

Also, the input function can take an optional argument of type string. This string will be printed on the console window. This could be used to print a message telling the user with anything that the programmer wants.

A special use case of the input function is when it is not assigned to any variable but instead is used as a stand-alone statement. In such a case, the input will function much like a getch() present in the c language.

3.11.3 open()

The return type of the open is image. It will take as an input parameter a string, which will basically be the path where the image is located.

3.11.4 save()

The return type of this function is void. This function will take 2 parameters – an image and a string. The string specifies the path where the image will be stored. Just like the display() function, save() function cannot be used in assignments.

4 Project Plan

4.1 Process

Our team was formed in late September. During the early days of the course, the three of us would meet once a week for approximately two hours to discuss and debate ideas for implementing a new language. This continued till the proposal was drafted.

Till the LRM phase, the entire team was on a learning curve of OCaml. All of us had ambitious ideas for the language, but they had to be tempered down in keeping with our timeline. For example, we initially were contemplating on implementing the video processing as well for our language. But then we came to realize that the language would get quite complex and hence we scraped it. In this phase, the three of us met every week on Wednesday after the PLT class for few hours.

By the end of mid-term we had a fairly clear idea of how our language compiler was going to get built. We divided ourselves into two teams of two members and one member. The two person team took over the part of designing the grammar: AST, Scanner and Parser, AST walk component whereas the other took over the part of designing the C++ converter thereof. As the functionality was getting added to the code base, each developer wrote small test cases for each of the functionality.

We used Github to help us control the code version. Every member of the group contributed to the project repository once their small code assignment was accomplished. Each team member did frequent updates to their code to fix bugs and make optimizations. Since this is the first time we all were using a Sub Version Control System, we initially encountered lots of difficulties in getting the hang of it. But in the end, using github had become second nature to us and found it very helpful as it brought much convenience to our project.

4.2 Coding Convention

Our coding convention was largely guided by the MicroC compiler, which was authored by Professor Stephen. We kept the format neat with proper tabs and spaces so that it would be relatively easy for the team members to read and understand each other's code. For the c++ code, we followed the standard coding guidelines of c++.

4.3 Project Timeline

2013/9/20 - Team formed

2013/9/25 - Language proposal submitted

2013/10/4 – Received feedback on the proposal from the TA

2013/10/10 - Proposal modified according to TA's feedback

2013/10/28 - Language Reference Manual submitted

2013/11/20 - Scanner, Parser completed

2013/11/25 - AST completed

2013/11/30 – Scanner, Parser tested

2013/12/15 - Semantic check completed and tested

2013/12/16 - C++ code for built-in functions completed

2013/12/20 - Code Generation and testing suites completed

4.4 Roles and Responsibilities

Team Member	Worked On
Shubhanshu Yadav	scanner.mll, parser.mly, sast.ml, C++ back-end, Final Project Report
Vaibhav Jagannathan	codegen.ml, parser.mly, sast.ml, C++ back-end, Final Project Report
Bhargav Sethuram	C++ back-end, Testing-Suite, testall.sh, Project Report

4.5 Environment

4.5.1 Language

We used O'Caml, C++, Shell Script and our own language SIP in our project to compose the files. Specifically, we used these languages to perform the following tasks:-

- **O'Caml -** To write the Scanner, Parser, Ast, Semantics Checking, and Code Generation parts.
- **C++** - To write the implementation of the built-in image manipulation data structures and functions.
- **SIP** – To write the programs and the testing cases.
- **Shell Script** - To make the executable programs and to perform automated testing.

4.5.2 Environment

1. Operating System: Ubuntu Release 12.04, Windows 8

2. OCaml compiler: Version 4

3. Sublime Text 2 for ocaml

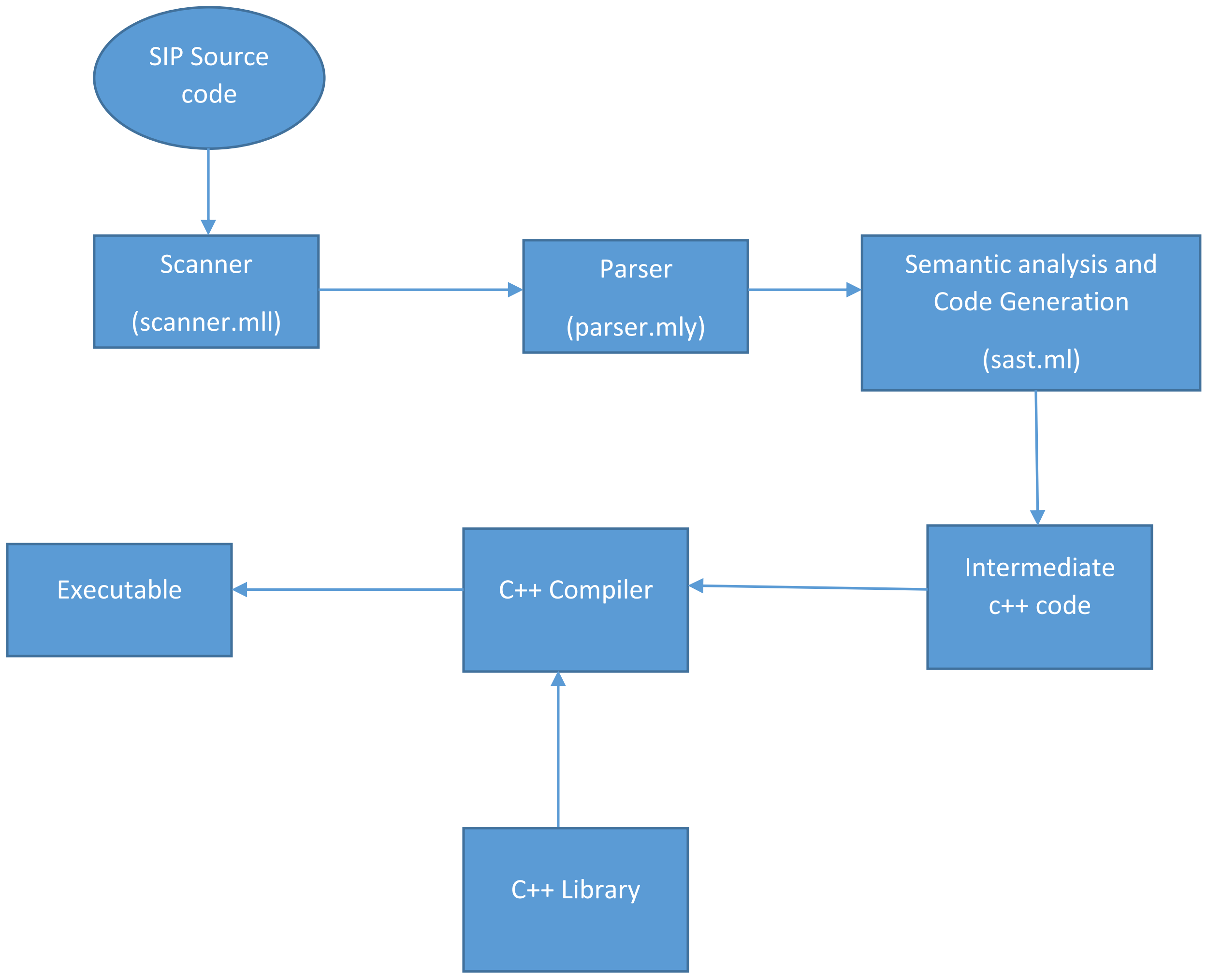
4. Visual Studio, Sublime Text 2 for c++ backend

5. g++

5 Project Architecture

5.1 Architectural Design Diagram

Here is a diagram to illustrate the main components of SIP project.



5.2 Structure and Interface

1. **Scanner:** This part of the compiler is the lexical analysis part. It is realized by the file *scanner.mll*.

The scanner relies on the token definitions of the Parser, to read the input file and convert the valid character groupings into tokens.

2. **Parser:** The parser is realized by the file *parser.mly*. The role of the parser is to consume the tokens generated by the Scanner and try and fit them in the grammar defined in the parser. The parser also uses the interface defined by the *ast.ml* to generate the ast nodes. The parser builds the Abstract Syntax Tree of the input program. If the parser is not able to fit the current token in the specified grammar, it starts discarding tokens to reach an error state. From here, it will try to drop input token to recover from the error based on the rules provided by the grammar. It also raises the Parsing.parse exception. The advantage we obtained from using this approach was that we were able to catch most of the parse error belonging to a source program in one pass. In our view, this was a much better approach than raising exception after every encountered parse error.

3. **Semantic Checking and Code Generation**: The semantic analyzer includes the important features like: check the duplication of variables in global/local variables, duplication of function names, validity of expression, validity of statements, validity of function declaration, etc. If there is any error in the SIP source code that has passed the parser section, it will be caught by the semantic analyzer and exception will be thrown to inform the user which kind of error has been detected.

We maintain an external error count variable across the entire compiler. If the error is present, then no intermediate c++ code will be generated and an exception will be raised. The exception raised will also provide us with the information of the number of errors generated during both syntax and semantics checks.

6. Test Plan

The testing of SIP was implemented in two manners – one for testing the output of images, and one for testing everything else. Our discussion begins with the latter. In testing the non-image-specific functionality of the language, we adopted the testing practices implemented in MicroC. Namely, we created a file, named *testall.sh*, which would run a battery of tests, and save the output to temporary files. For each one of our tests, we also created files that contained the correct output. Our testall.sh would run each of the test-files and compare, via *diff*, the output files from the execution of our test files against the expected output in the files we had created. It reports failure if a difference exists, otherwise it returns success. For the image-based tester files, the output are printed and then tested against the correct result.

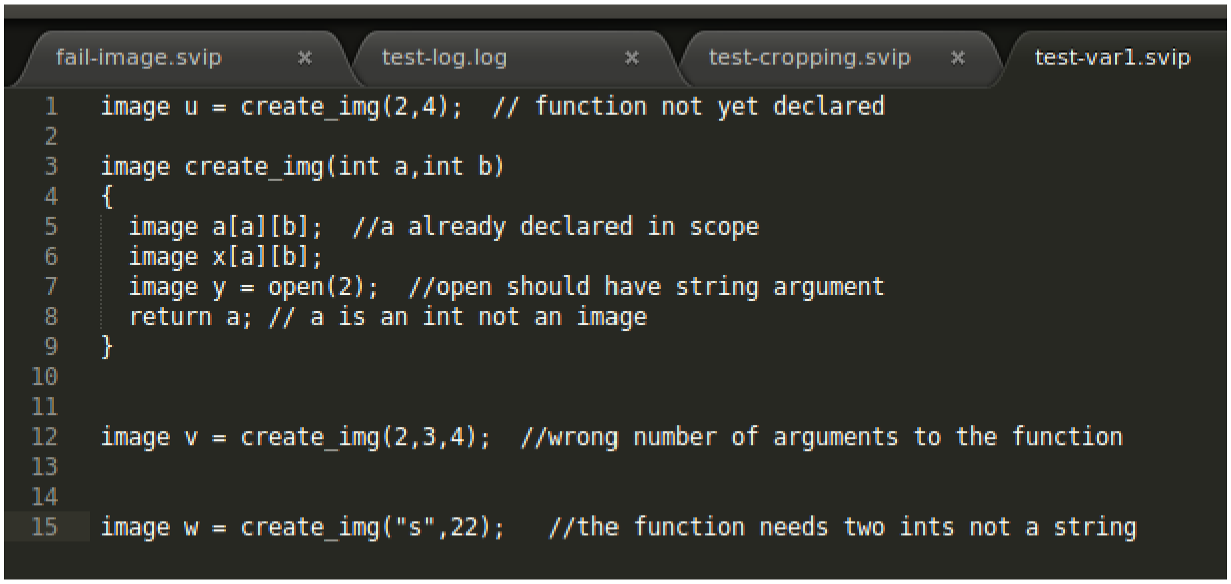
Utilizing this test suite has been very helpful to us throughout the creation of SIP. Since we are all new to OCaml specifically, and functional programming in general, we were consistently introducing bugs and this enabled us to see what was breaking – key information for repairing those bugs.

6.1 Semantic Test

In semantic checking, we basically check to see whether the semantic analyzer is able to catch the errors before proceeding. Typical errors are as follows :-

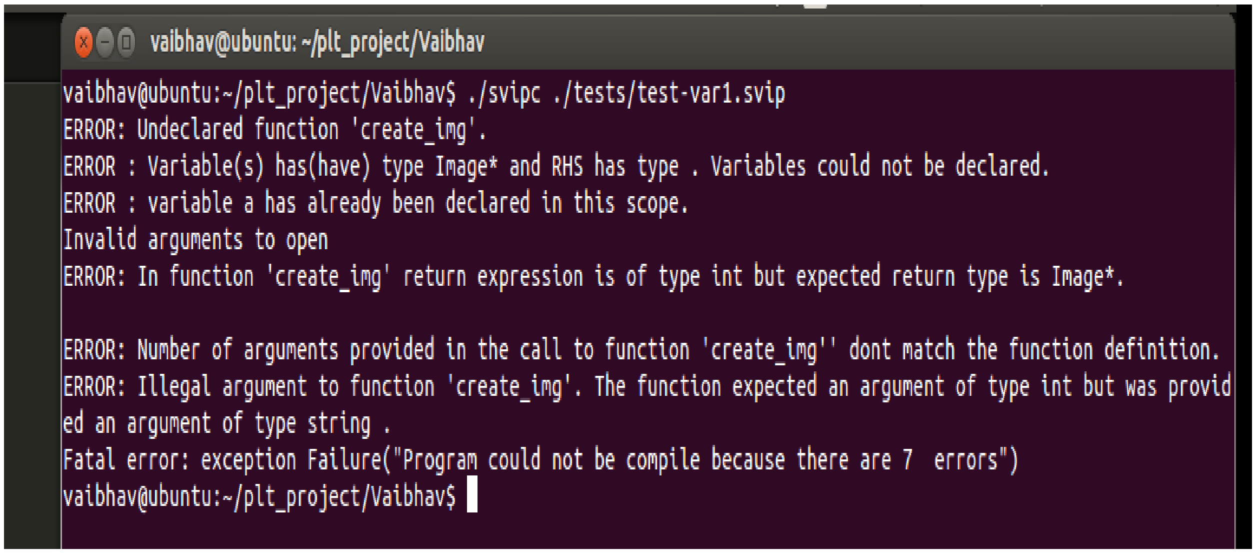
- Duplicated function name/ variable name/ parameter name.
- Wrong number of parameters when using a function.
- Non-defined function.
- Wrong expression type in assign or function call.
- Wrong statement form.
- Lack of return statement in a function, no main function, etc.

For each of these possible cases, a SIP code script is written and they are all stored. A sample testing code is:



```
image u = create_img(2,4);  // function not yet declared

image create_img(int a,int b)
{
  image a[a][b];  //a already declared in scope
  image x[a][b];
  image y = open(2);  //open should have string argument
  return a; // a is an int not an image
}


image v = create_img(2,3,4);  //wrong number of arguments to the function


image w = create_img("s",22);   //the function needs two ints not a string
```

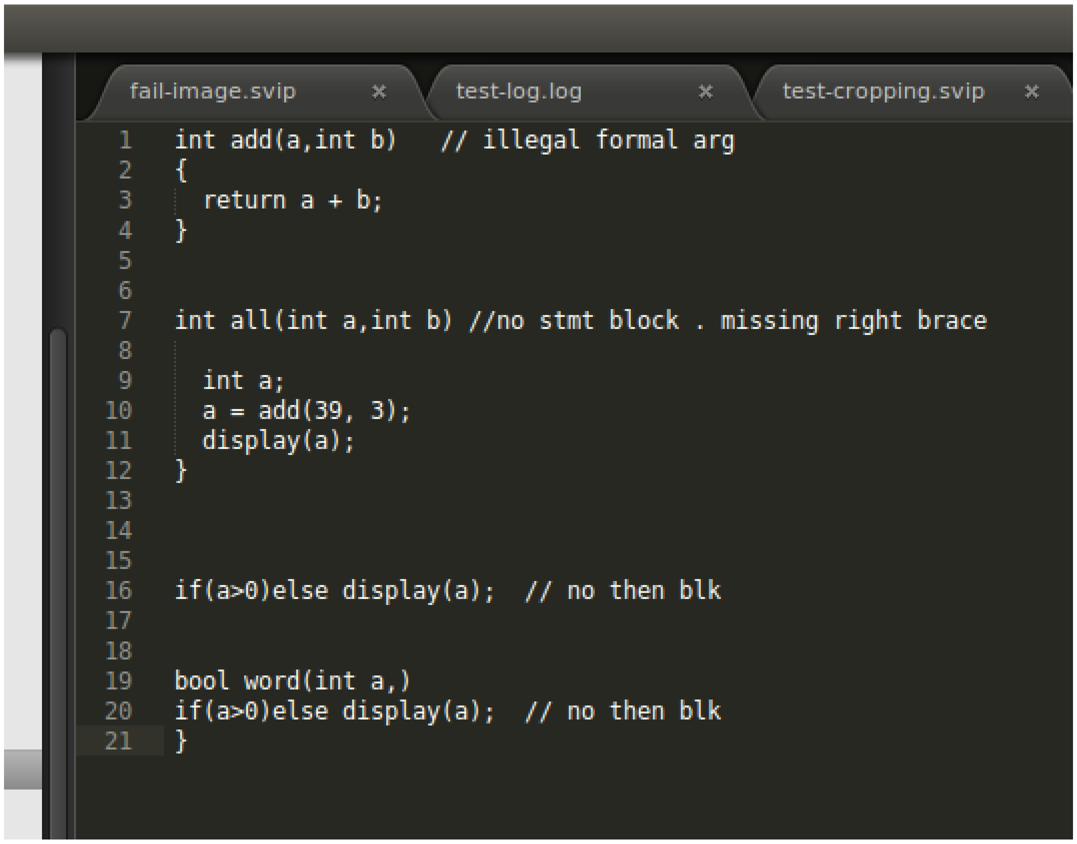
The error trail generate by this faulty program is as follows :-



```
vaibhav@ubuntu: ~/plt_project/Vaibhav
vaibhav@ubuntu:~/plt_project/Vaibhav$ ./svipc ./tests/test-var1.svip
ERROR: Undeclared function 'create_img'.
ERROR : Variable(s) has(have) type Image* and RHS has type . Variables could not be declared.
ERROR : variable a has already been declared in this scope.
Invalid arguments to open
ERROR: In function 'create_img' return expression is of type int but expected return type is Image*.

ERROR: Number of arguments provided in the call to function 'create_img'' dont match the function definition.
ERROR: Illegal argument to function 'create_img'. The function expected an argument of type int but was provid
ed an argument of type string .
Fatal error: exception Failure("Program could not be compile because there are 7  errors")
vaibhav@ubuntu:~/plt_project/Vaibhav$ 
```

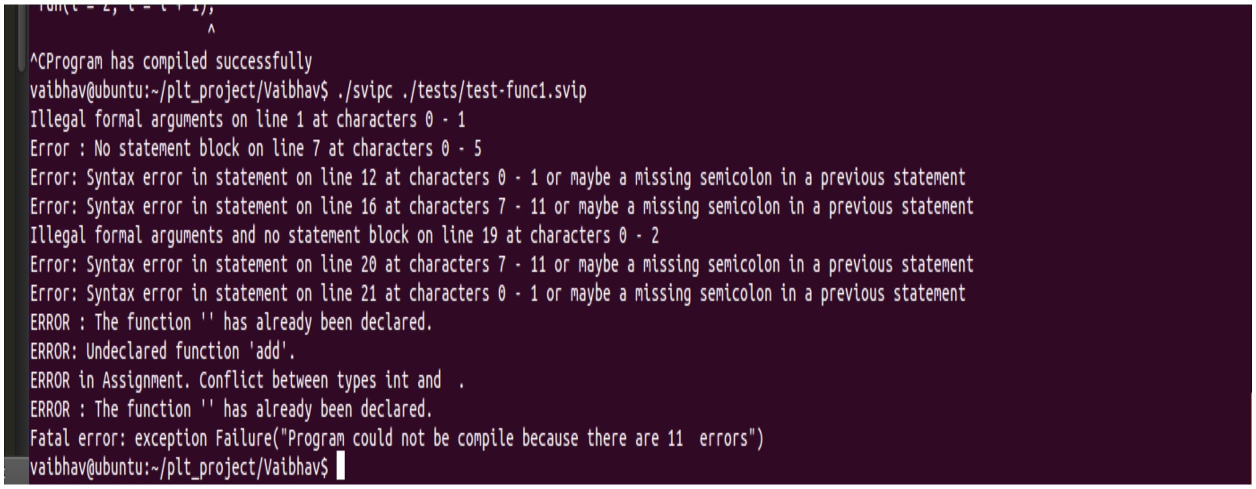
6.2 Syntax Checking

The input SIP source code program is as following :-



```
int add(a,int b)   // illegal formal arg
{
  return a + b;
}


int all(int a,int b) //no stmt block . missing right brace

  int a;
  a = add(39, 3);
  display(a);
}



if(a>0)else display(a);  // no then blk


bool word(int a,)
if(a>0)else display(a);  // no then blk
}
```

The error trail generated by this program is as follows:-



```
fun(i = 2, i = i + 1);
                 ^
^CProgram has compiled successfully
vaibhav@ubuntu:~/plt_project/Vaibhav$ ./svipc ./tests/test-func1.svip
Illegal formal arguments on line 1 at characters 0 - 1
Error : No statement block on line 7 at characters 0 - 5
Error: Syntax error in statement on line 12 at characters 0 - 1 or maybe a missing semicolon in a previous statement
Error: Syntax error in statement on line 16 at characters 7 - 11 or maybe a missing semicolon in a previous statement
Illegal formal arguments and no statement block on line 19 at characters 0 - 2
Error: Syntax error in statement on line 20 at characters 7 - 11 or maybe a missing semicolon in a previous statement
Error: Syntax error in statement on line 21 at characters 0 - 1 or maybe a missing semicolon in a previous statement
ERROR : The function '' has already been declared.
ERROR: Undeclared function 'add'.
ERROR in Assignment. Conflict between types int and  .
ERROR : The function '' has already been declared.
Fatal error: exception Failure("Program could not be compile because there are 11  errors")
vaibhav@ubuntu:~/plt_project/Vaibhav$ 
```

7. Lessons Lerned

7.1 Shubhanshu -> As a semester-long Computer Science programming project, working on SIP has taught me lot of things. Since I did my under-grad in the “Electronics and Communication” engineering, it was initially very difficult for me to get used to this course. But, as time went on, my understanding on the concepts discussed In the class increased. Furthermore, as I had just a basic level of programming experience prior to taking this course, I found that o’caml had a very steep learning curve. It requires a completely different way of thinking as to how to approach problem solving as compared to c++ or java. But in the end, it’s totally worth it.

Last but not the least, getting involved in making a compiler helped me immensely in gaining a deeper knowledge of the workings of a compiler. Also, initially, we didn’t have a clear cut description of the language we were designing which posed a great deal of problem afterwards. I, myself had to make several changes to the parser and semantic analysis because of that.

All in all, it was a great project. There is a great sense of achievement that comes with seeing your compiler in action, working as it's supposed to be.

7.2 Vaibhav Jagannathan -> Working on SIP has been very educational. I learnt the importance of finishing work earlier and the problems encountered while working with teams.

I found using git to manage the project files extremely cumbersome and it took me a lot of time to get used to the interface but I finally see the benefits of using a version control system from the several times where minor changes i made to the code caused numerous bugs. It was at times like these that being able to roll back all files to the last stable state was great help. Also, since we would only meet once a week to discuss the code once a week we were more often than not, informed of the progress made through git. One last lesson i learned is just because two discrete parts of a project work perfectly individually they can wreak havoc when put together. The conflict between code compiled in Visual Studio on Windows and g++ on Linux was a disaster i could never have foreseen. We ended up rewriting most of the C++ code-base of our project in the last days due to this unexpected problem.

All in all it was a refreshing experience. It seemed like every time I would be satisfied with the progress of the project there would be a problem lurking around the corner constantly keeping on my toes which i thoroughly enjoyed.

7.3 Bhargav Sethuram -> Through this project in programming languages and translators, first and foremost, I value the experience I have gained in designing and implementing a fully functional compiler. Breaking down the compiler into its components - the parser, scanner, the Abstract syntax tree, et al. - made the implementation of a seemingly impossible project possible. Hence, breaking down a complex problem into it's simple components and working together as a team to meet the deadlines - which was quite important considering this was a group of three - are two valuable life lessons I will not forget.

I believe, in many ways, working on this project has given me a taste of my future career.

8. APPENDIX

****

Scanner.mll – Authored by Shubhanshu Yadav

****

```
{ open Parser }

let digit = ['0' - '9']

let integer = (digit)+'.'(digit)*('e'('+'|'-')?((digit)+))?
let fraction = '.'(digit)+('e'(('+'|'-')?)((digit)+))?

let exponent = (digit)+'e'('+'|'-')?(digit)+

let identifier = ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']*



rule token = parse

  [' ' '\t'] { token lexbuf } (* Whitespace *)

| ['\r' '\n'] {Lexing.new_line lexbuf; token lexbuf }

| "/*"     { comment 0 lexbuf.Lexing.lex_curr_p.Lexing.pos_lnum lexbuf }

| "//" [^ '\n' '\r']* eof                        { EOF }        (* Comments *)

| "//"[^ '\n' '\r']*['\n' '\r']                  {Lexing.new_line lexbuf; token lexbuf }

| '('                                            { LPAREN }

| ')'                                            { RPAREN }

| '{'                                            { LBRACE }

| '}'                                            { RBRACE }

| '['                                            { LBRACKET }

| ']'                                            { RBRACKET }

| ';'                                            { SEMI }

| ':'                                            { TO }

| ','                                            { COMMA }

| '.'                                            { DOT }

| '+'                                            { PLUS }

| '-'                                            { MINUS }

| '*'                                            { TIMES }

| '/'                                            { DIVIDE }

| '='                                            { ASSIGN }

| '^'                                            { POWER }
| "=="                                  { EQ }
| "!="                                  { NEQ }
| '<'                                   { LT }
| '>'                                   { GT }
| "<="                                  { LEQ }
| ">="                                  { GEQ }
| '%'                                   { MOD }
| "-="                                  { MINUSEQ }
| "*="                                  { TIMESEQ }
| "/="                                  { DIVIDEEQ }
| "%="                                  { MODEQ }
| "+="                                  { PLUSEQ }
| '!'                                   { NOT }
| "in"                                  { IN }
| "with"                                { WITH }
| "if"                                  { IF }
| "else"                                { ELSE }
| "for"                                 { FOR }
| "do"                                  { DO }
| "while"                               { WHILE }
| "return"                              { RETURN }
| "int"                                 { INT }
| "image"                               { IMAGE }
| "pixel"                               { PIXEL }
| "float"                               { FLOAT }
| "string"                              { STRING }
| "bool"                                { BOOL }
| "video"                                    { VIDEO }

| "true"                                     { TRUE(true) }

| "false"                                    { FALSE(false) }

| "void"                                     { VOID }

| "break"                                    { BREAK }

| "continue"                                 { CONTINUE}

| '"'([^'"']* as str)'"'                     { STRINGS(str) }

| digit+ as lxm                              { INTEGERS(int_of_string lxm) }

| (integer|fraction|exponent) as flt         { FLOATS(float_of_string flt) }

| identifier as lxm                          { ID(lxm) }

| eof                                        { EOF }

| _ as chr        { print_endline ("Illegal Character '"^ Char.escaped chr ^ "' at line " ^
string_of_int lexbuf.Lexing.lex_curr_p.Lexing.pos_lnum); incr Ast.error_count; token lexbuf }


and comment level start_line = parse

  "*/" { if (level=0) then (token lexbuf) else (comment (level-1) start_line lexbuf)}

| "/*" { comment (level+1) start_line lexbuf}

| ['\r' '\n'] {Lexing.new_line lexbuf ; comment level start_line lexbuf }

| eof  { print_endline ("A comment started on line "^string_of_int start_line^" was not closed.
Or maybe a comment nested within the first one was not closed.");EOF}

| _   { comment level start_line lexbuf }
```

****

parser.mly – Authored by Shubhanshu and Vaibhav

****

```
%{ open Ast ;;

  (*Parsing.set_trace true;;*)
  let error_position () =
  incr Ast.error_count;
  let spos = Parsing.symbol_start_pos() and epos = Parsing.symbol_end_pos() in
      "on line " ^ string_of_int spos.Lexing.pos_lnum ^ " at characters " ^ string_of_int
(spos.Lexing.pos_cnum - spos.Lexing.pos_bol) ^ " - " ^ string_of_int (epos.Lexing.pos_cnum -
epos.Lexing.pos_bol)


 (* let parse_error s =  incr Ast.error_count; print_string ("Error "^string_of_int !Ast.error_count
^ " : ")*)

%}
```

```
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA TO LBRACKET RBRACKET DOT

%token PLUS MINUS TIMES DIVIDE ASSIGN MOD PLUSEQ MINUSEQ TIMESEQ DIVIDEEQ
MODEQ POWER

%token EQ NEQ LT LEQ GT GEQ NOT

%token RETURN IF ELSE FOR WHILE BREAK CONTINUE IN WITH DO

%token INT IMAGE PIXEL FLOAT STRING BOOL VOID VIDEO

%token <bool> TRUE FALSE

%token <int> INTEGERS

%token <float> FLOATS

%token <string> ID STRINGS

%token EOF


%nonassoc NOELSE

%nonassoc ELSE

%right ASSIGN PLUSEQ MINUSEQ TIMESEQ DIVIDEEQ MODEQ

%nonassoc  UMINUS NOT INCR DECR

%left EQ NEQ

%left LT GT LEQ GEQ

%left POWER

%left PLUS MINUS

%left TIMES DIVIDE

%left MOD


%start program

%type <Ast.program> program
```

```
%%
program:
  rev_program { List.rev $1 }

rev_program:
  /* nothing */ { [] }
 | rev_program stmt { Stmt($2)::$1 }
 | rev_program fdefn { Fdefn($2)::$1 }



fdefn:
    v_type ID LPAREN formal_opt RPAREN stmt_block
     { {
      rtype = $1;
      fname = $2;
      formals = $4;
      body = $6
      } }
   | v_type ID LPAREN error RPAREN stmt_block
     { print_endline ("Illegal formal arguments "  ^ error_position ());{
      rtype = "";
      fname = "";
      formals = [];
      body = []
      } }
   | v_type ID LPAREN error RPAREN error
```

```
    { print_endline ("Illegal formal arguments and no statement block "  ^ error_position ());{
     rtype = "";
     fname = "";
     formals = [];
     body = []
     } }
  | v_type ID LPAREN formal_opt RPAREN error
    { print_endline ("Error : No statement block "  ^ error_position ());{
     rtype = "";
     fname = "";
     formals = [];
     body = []
     } }

vdecl:
   v_type id_list                        { Vdefn($1,List.rev $2) }
  | v_type id_array ASSIGN expr          { Vassign($1,$2,$4) }



id_array:
  ID  { Single($1) }
  |ID LBRACKET expr RBRACKET { Array($1,$3,Integers(0),Integers(0)) }
  |ID LBRACKET expr RBRACKET LBRACKET expr RBRACKET { Array($1,$3,$6,Integers(0)) }
  |ID LBRACKET expr RBRACKET LBRACKET expr RBRACKET LBRACKET expr RBRACKET {
Array($1,$3,$6,$9) }
```

```
id_list:
   id_array                  { [$1] }
 | id_list COMMA id_array    { ($3) :: $1  }



v_type:
 | VOID                      { "void" }
 | INT                       { "int" }
 | FLOAT                     { "float" }
 | PIXEL                     { "Pixel" }
 | IMAGE                     { "Image*" }
 | STRING                    { "string" }
 | BOOL                      { "bool" }
 | VIDEO                     { "Video" }

stmt_block:
   LBRACE stmt_list RBRACE           { List.rev $2 }

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

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

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

formal_list:
   v_type id_array                        { [($1,$2)] }
  | formal_list COMMA v_type id_array   { ($3,$4) :: $1 }

actual_opt:
   /*nothing*/                          { [] }
  | expr                                { [$1] }
  |actual_opt COMMA expr                { $3::$1 }

v_expr_opt:
  |  vdecl                              { Vdecl($1) }
  |  expr_opt                           { Expr($1) }

stmt:
 v_expr_opt SEMI                        { Vexpr($1) }
| stmt_block                            { Block($1) }
| IF LPAREN expr RPAREN stmt %prec NOELSE  { If($3,$5,Block([])) }
| IF LPAREN expr RPAREN stmt ELSE stmt     { If($3,$5,$7) }
| FOR LPAREN v_expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt { For($3,$5,$7,$9) }
/* | FOR LPAREN v_expr_opt IN ID RPAREN stmt { For_list($3,$5,$7) }
| FOR LPAREN v_expr_opt IN range RPAREN stmt { For($3,,,$9) }
| FOR LPAREN v_expr_opt IN range WITH expr RPAREN stmt { For($3,,$9,$11) }*/
| WHILE LPAREN expr RPAREN stmt            { While($3,$5) }
| DO stmt WHILE LPAREN expr RPAREN        { Do_while($5,$2) }

| RETURN expr_opt SEMI                     { Return($2) }

| BREAK SEMI                               { Break }

| CONTINUE SEMI                            { Continue }

| error   { print_endline ("Error: Syntax error in statement " ^ error_position() ^ " or maybe a
missing semicolon in a previous statement"); Vexpr(Expr(Noexpr)) }



range:

  |expr                               { (false,$1,Noexpr) }

  |expr TO expr                       { (true,$1,$3) }


id_array_access:

  | ID LBRACKET range RBRACKET

                      { Var_access(1,Id($1),$3,(false,Noexpr,Noexpr),(false,Noexpr,Noexpr)) }

  | ID LBRACKET range RBRACKET LBRACKET range RBRACKET

                                      { Var_access(2,Id($1),$3,$6,(false,Noexpr,Noexpr)) }

  | ID LBRACKET range RBRACKET LBRACKET range RBRACKET LBRACKET range RBRACKET

                                      { Var_access(3,Id($1),$3,$6,$9) }

  | LPAREN expr RPAREN LBRACKET range RBRACKET

                 { Var_access(1,Paren($2),$5,(false,Noexpr,Noexpr),(false,Noexpr,Noexpr)) }

  | LPAREN expr RPAREN LBRACKET range RBRACKET LBRACKET range RBRACKET

                                      { Var_access(2,Paren($2),$5,$8,(false,Noexpr,Noexpr)) }



integer_list:

| expr                                     { [$1] }
```

```
  | integer_list COMMA expr                  { $3 :: $1 }

expr:
   | ID ASSIGN expr                          { Assign(Id($1),$3) }
   | id_array_access ASSIGN expr             { Assign($1,$3) }
   | objid ASSIGN expr                       { Assign($1,$3) }
   | binop_assign                            { $1 }
   | objid                                   { $1 }
   | LBRACKET integer_list RBRACKET          { Int_list(List.rev $2) }
   | ID                                      { Id($1) }
   | id_array_access                         { $1 }
   | INTEGERS                                { Integers($1) }
   | STRINGS                                 { Strings($1) }
   | FLOATS                                  { Floats($1) }
   | TRUE                                    { Boolean($1) }
   | FALSE                                   { Boolean($1) }
   | uop                                     { $1 }
   | binop                                   { $1 }
   | LPAREN expr RPAREN                      { Paren($2) }
   | ID LPAREN actual_opt RPAREN             { Call($1,List.rev $3) }
   | objid LPAREN actual_opt RPAREN          { Objcall($1,List.rev $3) }

objid:
   | LPAREN expr RPAREN DOT ID               { Objid($2,$5) }
   | id_array_access DOT ID                  { Objid($1,$3) }
   | ID DOT ID                               {Objid(Id($1),$3)}
```

```
uop:
  | MINUS expr  %prec UMINUS          { Uop(Sub,$2) }
  | NOT expr  %prec NOT               { Uop(Not,$2) }

binop:
  | 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 MOD expr                     { Binop($1, Mod , $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 POWER expr                   { Binop($1,Pow,$3) }

 binop_assign:
  | ID PLUSEQ expr                    { Assign(Id($1),Binop(Id($1),Add,$3)) }
  | id_array_access PLUSEQ expr       { Assign($1,Binop($1,Add,$3)) }
  | ID MINUSEQ expr                   { Assign(Id($1),Binop(Id($1),Sub,$3)) }
  | id_array_access MINUSEQ expr      { Assign($1,Binop($1,Sub,$3)) }
  | ID TIMESEQ expr                   { Assign(Id($1),Binop(Id($1),Mult,$3)) }
  | id_array_access TIMESEQ expr      { Assign($1,Binop($1,Mult,$3)) }
  | ID DIVIDEEQ expr                  { Assign(Id($1),Binop(Id($1),Div,$3)) }
  | id_array_access DIVIDEEQ expr              { Assign($1,Binop($1,Div,$3)) }
  | ID MODEQ expr                              { Assign(Id($1),Binop(Id($1),Mod,$3)) }
  | id_array_access MODEQ expr                 { Assign($1,Binop($1,Mod,$3)) }
```

****

sast.ml – Authored by Shubhanshu and Vaibhav

****

```
open Ast

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

type globals =
{
        funMap : func_decl NameMap.t ;      (*all functions declared*)
        varMap : string NameMap.t ; (*all global variables*)
}

type locals =
{
        outMap : string NameMap.t ;(*variables outside of the current scope*)
        currMap : string NameMap.t ;        (*variables in the current scope*)
}



(*
type env = globals * locals or globals * locals * (stmt list)
*)

let string_of_id = function
| Single(id) -> id
| Array(id,e1,e2,e3) ->
(
if e3 <> Integers(0) then
id ^ "[" ^string_of_expr e1 ^"]" ^ "[" ^ string_of_expr e2 ^ "]" ^ "[" ^ string_of_expr e2 ^ "]"
else if e2 <> Integers(0) then
id ^ "[" ^string_of_expr e1 ^"]" ^ "[" ^ string_of_expr e2 ^ "]"
else
id ^ "[" ^string_of_expr e1 ^"]"        )
(*takes globals, locals and an expr to return a string which is the return datatype of the expr,
string to print to file*)

let rec check_expr (globals,locals) = function
   Integers(l) -> ("int",string_of_int l)
  | Floats(f) -> ("float",string_of_float f)
  | Strings(s) -> ("string",("\""^s^"\""))
  | Boolean(b) -> ("bool",(if(b) then "true" else "false"))
  | Id(s) ->
```

```
        (try (
                NameMap.find s locals.currMap,s
                )
        with
                | Not_found -> (
                        try (
                        NameMap.find s locals.outMap,s
                        )
                            with
                        | Not_found -> (
                            print_endline ("ERROR: Undelared Variable '"^s^"'.");
                            incr Ast.error_count;
                            ("",s)
                            )
                    )
        )

    | Var_access(i,e,(b1,e11,e12),(b2,e21,e22),(b3,e31,e32)) ->
                let     r,str   = check_expr (globals,locals) e   and
                            r11,str11 = check_expr (globals,locals) e11 and
                            r12,str12 = check_expr (globals,locals) e12 and
                            r21,str21 = check_expr (globals,locals) e21 and
                            r22,str22 = check_expr (globals,locals) e22 and
                            r31,str31 = check_expr (globals,locals) e31 and
                            r32,str32 = check_expr (globals,locals) e32 in
                            (match (i,r,(b1,r11,r12),(b2,r21,r22),(b3,r31,r32)) with
                            | (1,r,(true,"int","int"),_,_) when r = "Video" -> "Video", str
```

```
                                              | (1,r,(false,"int",_),_,_) when r = "int" -> ("int",
str^"["^str11^"]" )

                                              | (1,r,(false,"int",_),_,_)          when r = "Pixel" ->
("int","("^str^")."^(match str11 with | "0" -> "r" | "1" -> "g" | "2" -> "b" | _ ->(print_endline
"Erroneous array access.";incr Ast.error_count ;"")))

                                              | (1,r,(false,"int",_),_,_)          when r = "Video" ->
"Image*",str

                                              | (2,r,(true,"int","int"),(true,"int","int"),_) when r =
"Image*" -> "Image*","getCropped("^str^","^str11^","^str12^","^str21^","^str22^")"

                                              | (2,r,(true,"int","int"),(false,"int",_),_) when r = "Image*"
-> "Image*","getCropped("^str^","^str11^","^str12^","^str21^","^str21^")"

                                              | (2,r,(false,"int",_),(true,"int","int"),_) when r = "Image*"
-> "Image*","getCropped("^str^","^str11^","^str11^","^str21^","^str22^")"

                                              | (2,r,(false,"int",_),(false,"int",_),_) when r = "Image*" ->
"Pixel","(*("^str^")).pixels["^str11^" * (*("^str^")).width + "^str21^"]"

                                              | (2,r,(false,"int",_),(false,"int",_),_) when r = "int" ->
"int",str^"["^str11^"]["^str21^"]"

                                              | (3,r,(true,"int","int"),(true,"int","int"),(true,"int","int"))
when r = "Video" -> "",str

                                              | (3,r,(true,"int","int"),(true,"int","int"),(false,"int",_))
when r = "Video" -> "",str

                                              | (3,r,(true,"int","int"),(false,"int",_),(true,"int","int"))
when r = "Video" -> "",str

                                              | (3,r,(true,"int","int"),(false,"int",_),(false,"int",_)) when r
= "Video" -> "",str

                                              | (3,r,(false,"int",_),(true,"int","int"),(true,"int","int"))
when r = "Video" -> "",str

                                              | (3,r,(false,"int",_),(true,"int","int"),(false,"int",_)) when r
= "Video" -> "",str

                                              | (3,r,(false,"int",_),(false,"int",_),(true,"int","int")) when r
= "Video" -> "",str

                                              | (3,r,(false,"int",_),(false,"int",_),(false,"int",_)) when r =
"int" -> "int",str^"["^str11^"]["^str21^"]["^str31^"]"
```

```
                                        | _ -> print_endline "Erroneous array access.";incr
Ast.error_count ;"",str
                                        )

(*
add op overloading ??

*)

| Binop(e1,o,e2) ->
        let t1,str1 = check_expr (globals,locals) e1 and t2,str2 = check_expr (globals,locals) e2 in
                (match o,t1,t2 with
                        | (Add,"int","int") -> "int", (str1^" + "^ str2)
                        | (Add,"int","float") -> "float", (str1^" + " ^ str2)
                        | (Add,"float","int") -> "float", (str1^" + " ^ str2)
                        | (Add,"float","float") -> "float", (str1^" + " ^ str2)
                        | (Add,"string","string") -> "string",str1^" + " ^ str2
                        | (Add,"bool","bool") -> "bool",str1^" || " ^ str2
                        | (Add,"Pixel","Pixel") -> "Pixel",str1^" + " ^ str2
                        | (Add,"Image*","Image*") -> "Video",str1^" + " ^ str2
                        | (Add,"Image*","Video") -> "Video",str1^" + " ^ str2
                        | (Add,"Video","Image*") -> "Video",str1^" + " ^ str2
                        | (Add,"Video","Video") -> "Video",str1^" + " ^ str2
                        | (Sub,"int","int") -> "int",str1^" - " ^ str2
                        | (Sub,"int","float") -> "float",str1^" - " ^ str2
                        | (Sub,"float","int") -> "float",str1^" - " ^ str2
                        | (Sub,"float","float") -> "float",str1^" - " ^ str2
                        | (Mult,"int","int") -> "int",str1^" * " ^ str2
                        | (Mult,"int","float") -> "float",str1^" * " ^ str2
                        | (Mult,"int","string") -> "string","String_rep("^str1^","^str2^")"
                        | (Mult,"int","Pixel") -> "Image*",str1^" * " ^ str2
                        | (Mult,"int","Image*") -> "Video",str1^" * " ^ str2
                        | (Mult,"int","Video") -> "Video",str1^" * " ^ str2
                        | (Mult,"float","int") -> "float",str1^" * " ^ str2
                        | (Mult,"float","float") -> "float",str1^" * " ^ str2
                        | (Mult,"string","int") -> "string","String_rep("^str2^","^str1^")"
                        | (Mult,"bool","bool") -> "bool",str1^" && " ^ str2
                        | (Mult,"Pixel","int") -> "Image*",str1^" * " ^ str2
                        | (Mult,"Image*","int") -> "Video",str1^" * " ^ str2
                        | (Mult,"Video","int") -> "Video",str1^" * " ^ str2
                        | (Div,"int","int") -> "int",str1^" / " ^ str2
                        | (Div,"int","float") -> "float",str1^" / " ^ str2
                        | (Div,"float","int") -> "float",str1^" / " ^ str2
                        | (Div,"float","float") -> "float",str1^" / " ^ str2
                        | (Mod,"int","int") -> "int",str1^" % " ^ str2
                        | (Equal,t1,t2) when t1=t2 && t1 <> "void" -> "bool",str1^" == " ^ str2
                        | (Neq,t1,t2) when t1=t2 && t1 <> "void" -> "bool",str1^" != " ^ str2
                        | (Greater,"int","int") -> "bool",str1^" > " ^ str2
                        | (Greater,"float","float") -> "bool",str1^" > " ^ str2
                        | (Geq,"int","int") -> "bool",str1^" >= " ^ str2
                        | (Geq,"float","float") -> "bool",str1^" >= " ^ str2
                        | (Less,"int","int") -> "bool",str1^" < " ^ str2
                        | (Less,"float","float") -> "bool",str1^" < " ^ str2
                        | (Leq,"int","int") -> "bool",str1^" <= " ^ str2
```

```
                        | (Leq,"float","float") -> "bool",str1^" <= " ^ str2
                        | (_,t1,t2) -> (print_endline ("ERROR in op. Conflict between types "^t1^"
and "^t2^" .");
                             incr Ast.error_count;
                             "","")
                   )

| Uop(o,e) -> let t,str = check_expr (globals,locals) e in
     (match o with
                             | Sub ->
                                  (match t with
                                   | "int" -> "int","-("^str^")"
                                   | "float" -> "float","-("^str^")"
                                   | "bool" -> "bool","!("^str^")"
                                   | s -> (print_endline ("ERROR in op. Conflict in type "^t^"
.");
                                   incr Ast.error_count;
                                   "","!("^str^")"))
                             | Not ->
                                  (match t with
                                       | "bool" -> "bool","!("^str^")"
                                       | s -> (print_endline ("ERROR in op. Conflict in type
"^t^" .");
                                       incr Ast.error_count;
                                       "","!("^str^")"))
                                  | _ -> "","!("^str^")")
| Paren(e) -> let a = check_expr (globals,locals) e in fst(a),"("^snd(a)^")"


| Assign(e1, e2) -> let t1,str1 = check_expr (globals,locals) e1 and t2,str2 =  check_expr
(globals,locals) e2 in
                                                                                                    if t1 = t2 then
t1,(str1^" = "^str2)
                                                                                                    else
(print_endline ("ERROR in Assignment. Conflict between types "^t1^" and "^t2^" .");

        incr Ast.error_count;

        t1,str1^" = "^str2)



(*Handle all specific inbuilt functions before the generic call*)


(*
| Call("open",actuals) ->
(*Handle inbuilt open function here*)
ignore
(let len = List.length actuals in
        (if len = 1 then
                (let param = List.hd actuals in
                        let ty = check_expr (globals,locals) param in
                                (if ty <> "string" then
                                (print_endline("ERROR: Illegal argument to function 'open'. The
function expected an argument of type string but was provided an argument of type "^ty^" .");
                                incr Ast.error_count)))
        else if len = 2 then
```

```
                    (let param1::param2::[] = actuals in
                              let ty1 = check_expr (globals,locals) param1 and ty2 = check_expr
(globals,locals) param2 in
                                                  if ty1 <> "string" || ty2 <> "bool" then
                                                            (print_endline("ERROR: Illegal argument to function
'open'. The function expected arguments of type string and bool but was provided arguments of
type "^ty1^", "^ty2^" .");
                                                            incr Ast.error_count)
                    )
          else
                    (print_endline("ERROR: Wrong number of arguments in call to the 'open'
function.");
                    incr Ast.error_count)
          )
);
"Image*",""



| Call("save",actuals) ->
(*Handle inbuilt save function here*)
ignore
(if List.length actuals <> 1 then
          (print_endline("ERROR: Wrong number of arguments in call to the 'save' function.");
          incr Ast.error_count)
 else
          (let param = List.hd actuals in
                    let ty =  check_expr (globals,locals) param in
                              if ty <> "string" then
```

```
                                        (print_endline("ERROR: Illegal argument to function 'save'. The
function expected an argument of type string but was provided an argument of type "^ty^" .");
                                                  incr Ast.error_count)
          )
);
"void",""



*)


| Call(id,actuals) ->
(*Handle all other generic calls here*)
if NameMap.mem id globals.funMap then (
          let fdecl = (NameMap.find id globals.funMap) in
          ignore (
                    try (List.fold_left2 (fun () actual formal ->
                              let tactual,str = check_expr (globals,locals) actual and (tformal,formalid)
= formal in
                                        (if tactual <> tformal then
                                                  (print_endline ("ERROR: Illegal argument to function
'"^fdecl.fname^"'. The function expected an argument of type "^tformal^" but was provided an
argument of type "^tactual^" .");
                                                  incr Ast.error_count)
                                        )
                              ) () actuals fdecl.formals
                    )
   with Invalid_argument(s) ->
          (print_endline ("ERROR: Number of arguments provided in the call to function
'"^fdecl.fname^"' dont match the function definition.");
```

```
                    incr Ast.error_count
                    )
                    );
                              fdecl.rtype,fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_expr
actuals) ^ ")"
)
else (print_endline ("ERROR: Undeclared function '"^id^"'.");
                              incr Ast.error_count;
                              "",""
)
(*
| Call("open",act)-> "open(" ^ String.concat ", " (List.map string_of_expr act) ^ ")"
| Call("save",act)-> "save(" ^ String.concat ", " (List.map string_of_expr act) ^ ")"
*)
| Int_list(lst) -> "int","{"^ String.concat "," (List.map (fun expr -> snd(check_expr (globals,locals)
expr)) lst) ^"}"
| Objid(obj,var) ->
let ty,objid = check_expr (globals,locals) obj in
          (match var with
                    | "height" when ty = "Image*" -> "int","(*("^objid^")).height "
                    | "width" when ty = "Image*" -> "int","(*("^objid^")).width "
                    | "mode" when ty = "Image*" -> "int","((*"^objid^")).GetMode() "
                    | "C1" when ty = "Pixel" ->  "int",objid^".r "
                    | "C2" when ty = "Pixel" ->  "int",objid^".g "
                    | "C3" when ty = "Pixel" ->  "int",objid^".b "
                    | _ -> ty,objid^"."^var
          )
```

```
| Objcall(Objid(obj,met),act) ->
        let ty,objid = check_expr (globals,locals) obj in
                let actuals = (List.map (fun expr -> (let typ,str = check_expr (globals,locals) expr
in
                        str) ) act)
                in

                (match met with
                                | "getPixel" when ty = "Image*" ->
"Pixel",objid^".GetPixel("^String.concat "," actuals^") "
                                | "getWidth" when ty = "Image*" ->  "int",objid^".GetWidth() "
                                | "getHeight" when ty = "Image*" -> "int",objid^".GetHeight() "
                                | "getMode" when ty = "Image*" -> "bool",objid^".GetMode() "
                                | "setPixel" when ty = "Image*" ->
"void",objid^".SetPixel("^String.concat "," actuals^") "
                                | "getRows" when ty = "Image*" ->
"Image*",objid^".GetRows("^String.concat "," actuals^") "
                                | "getCols" when ty = "Image*" ->
"Image*",objid^".GetCols("^String.concat "," actuals^") "
                                | "BMP2RGB" when ty = "Image*" ->
"Image*",objid^".BMP2RGB("^String.concat "," actuals^") "
                                | "RGB2BMP" when ty = "Image*" ->
"Image*",objid^".RGB2BMP() "
                                | "changeMode" when ty = "Image*" ->
"Image*",objid^".ChangeMode() "
                                | "RGB2YUV" when ty = "Image*" || ty = "Pixel" ->
ty,objid^".RGB2YUV() "
                                | "YUV2RGB" when ty = "Image*" || ty = "Pixel" ->
ty,objid^".YUV2RGB() "
                                | "getC1" when ty = "Pixel" -> "int",objid^".getC1() "
```

```
                                        | "getC2" when ty = "Pixel" -> "int",objid^".getC2() "

                                        | "getC3" when ty = "Pixel" -> "int",objid^".getC3() "

                                        | "setY" when ty = "Image*" ->
"int",objid^".alterBufY("^String.concat "," actuals^") "

                                        | "setR" when ty = "Image*" ->
"int",objid^".alterBufY("^String.concat "," actuals^") "

                                        | "setU" when ty = "Image*" ->
"int",objid^".alterBufU("^String.concat "," actuals^") "

                                        | "setG" when ty = "Image*" ->
"int",objid^".alterBufU("^String.concat "," actuals^") "

                                        | "setV" when ty = "Image*" ->
"int",objid^".alterBufV("^String.concat "," actuals^") "

                                        | "setB" when ty = "Image*" ->
"int",objid^".alterBufV("^String.concat "," actuals^") "

                                        | "addFrame" when ty = "Video" ->
"int",objid^".AddFrame("^String.concat "," actuals^") "

                                        | "getFrame" when ty = "Video" ->
"Image*","extract(readfile,writefile,start frame,no of frames)"

                                        | _ -> ty,objid^"."^met^"("^   String.concat "," actuals      ^") "

       )


| Noexpr -> "void",""

| _ -> "",""








let addMap map1 map2 =

NameMap.merge (fun k xo yo -> match xo,yo with
    | Some x, Some y -> Some y
    | None, yo -> yo
    | xo, None -> xo
  ) map1 map2


let check_vdecl (globals,locals) = function


|Vassign(ty,Single(id),Call("open",act))->
        let locals = {locals with currMap =
                if NameMap.mem id locals.currMap then (
                        print_endline ("ERROR : variable "^id^" has already been declared in this
scope.");
                        incr Ast.error_count;
                        locals.currMap
                        )
        else NameMap.add id ty locals.currMap}
   in
        if ty <> "Image*" && ty <> "Video" then (print_endline "open is used for images and
videos only."; incr Ast.error_count);
        if List.length act <> 1 then (print_endline "Invalid arguments to open" ; incr
Ast.error_count);
        let typ,strng = check_expr (globals,locals) (List.hd act) in
                if typ <> "string" then (print_endline "Invalid arguments to open" ; incr
Ast.error_count);
                locals,("\n{\nstring S = "^strng^";\nFILE *F = fopen(S.c_str(),\"r\");\n "^id^"=
BMPReadImage(F);\nfclose(F);\n}\n")
```

```
|Vassign(ty,Array(_),Call("open",_))->
        print_endline "Wrong use of variable declaration and open"; incr Ast.error_count;
        locals,";\n"

| Vassign(ty,ida,Call("input",act))->
        let id = string_of_id ida in
        let locals = {locals with currMap =
                if NameMap.mem id locals.currMap then (
                        print_endline ("ERROR : variable "^id^" has already been declared in this
scope.");
                        incr Ast.error_count;
                        locals.currMap
                        )
        else NameMap.add id ty locals.currMap}
  in
  if ty = "Image*" || ty = "Video" || ty = "Pixel" then (print_endline "open is used for images
and videos not input."; incr Ast.error_count);
        locals,(
                if List.length act > 1 then (print_endline "Invalid arguments to open" ; incr
Ast.error_count);
                if List.length act <> 0 then (
                let typ,strng = check_expr (globals,locals) (List.hd act) in
                        if typ <> "string" then (print_endline "Invalid arguments to open" ; incr
Ast.error_count);
                        "cout << ("^strng^")<< endl;\n cin.clear();\ncin >> "^ id ^";\n"
                        )
                else ("\n cin.clear();\ncin >> "^ id ^";\n"))
```

```
|Vassign("Pixel",Single(id),Int_list([Integers(i1);Integers(i2);Integers(i3)])) ->
{locals with currMap =
                    if NameMap.mem id locals.currMap then (
                              print_endline ("ERROR : variable "^id^" has already been declared in this
scope.");
                              incr Ast.error_count;
                              locals.currMap
                              )
          else NameMap.add id "Pixel" locals.currMap},(
                              if i1>255 || i1<0 || i2>255 || i2<0 || i3>255 || i3<0 then
                                                  (print_endline "This is a pixel. Assign an array of 3 integers
whose value is less than 255 to it."; incr Ast.error_count);
                                                  "Pixel "^id^"("^string_of_int i1^","^string_of_int
i2^","^string_of_int i3^");\n")




|Vassign("Pixel",Single(id),Int_list([e1;e2;e3])) ->
{locals with currMap =
                    if NameMap.mem id locals.currMap then (
                              print_endline ("ERROR : variable "^id^" has already been declared in this
scope.");
                              incr Ast.error_count;
                              locals.currMap
                              )
          else NameMap.add id "Pixel" locals.currMap},(
                    let t1,s1 = check_expr (globals,locals) e1 and t2,s2 = check_expr (globals,locals)
e1 and t3,s3 = check_expr (globals,locals) e1 in
```

```
                    if t1 <> "int" || t2 <> "int" || t2 <> "int" then
                                (print_endline "This is a pixel. Assign an array of 3 integers
to it."; incr Ast.error_count);
                                "Pixel "^id^"("^s1^","^s2^","^s3^");\n")




|Vdefn(ty,id_list) -> {locals with currMap =
                if ty <> "void" then (
                    List.fold_left (
                        fun vmap id_array ->
                            (match id_array with
                                |Single(id) -> if NameMap.mem id vmap then (
                                                print_endline
("ERROR : variable "^id^" has already been declared in this scope.");
                                                incr Ast.error_count;
                                                vmap
                                                )
                                            else NameMap.add id ty
vmap
                                |Array(id,e1,e2,e3) -> if NameMap.mem id vmap
then (
                                                    print_endline
("ERROR : variable "^id^" has already been declared in this scope.");
                                                    incr
Ast.error_count;
                                                    vmap
```

```
                                                                                                    )
                                                                                    else if fst(check_expr
(globals,locals) e1) <> "int" || fst(check_expr (globals,locals) e2) <> "int" ||        fst(check_expr
(globals,locals) e3) <> "int" then  (

        print_endline ("ERROR : variable "^id^" has already been declared in this scope.");

        incr Ast.error_count;

        vmap
                                                                                                            )
                                                                                         else
                                                                                    NameMap.add id ty vmap
                                                    )
                                    ) locals.currMap id_list
                            )
                            else (print_endline ("ERROR : Variable(s) "^Ast.string_of_id_list id_list^" cannot
have type void.");
                                        incr Ast.error_count;
                                        locals.currMap)
                            }, (String.concat ";\n" (List.map (fun id ->(match id with
                                                | Array(ident,a,b,Integers(0)) when ty = "Image*"->ty^" "^
ident^" = new Image("^string_of_expr a^","^string_of_expr b^ ");\n"
                                                | _ ->   (ty^" "^string_of_id id ))) id_list)^";\n")




|Vassign(ty,id_array,expr) ->
        let rhs,str = check_expr (globals,locals) expr in
```

```
            ({locals with currMap =
                if ty <> "void" then (
                    if ty = rhs then (
                        (match id_array with
                            |Single(id) -> if NameMap.mem id locals.currMap then (
                                                print_endline ("ERROR :
variable "^id^" has already been declared in this scope.");
                                                incr Ast.error_count;
                                                locals.currMap
                                                )
                                            else NameMap.add id ty
locals.currMap
                            |Array(id,e1,e2,e3) -> if NameMap.mem id locals.currMap
then (
                                                print_endline ("ERROR :
variable "^id^" has already been declared in this scope.");
                                                incr Ast.error_count;
                                                locals.currMap
                                                )
                                                else if fst(check_expr
(globals,locals) e1) <> "int" || fst(check_expr (globals,locals) e2) <> "int" ||        fst(check_expr
(globals,locals) e3) <> "int" then  (
                                                    print_endline
("ERROR : variable "^id^" has already been declared in this scope.");
                                                    incr Ast.error_count;
                                                    locals.currMap
                                                )

                                            else NameMap.add id ty
locals.currMap
```

```
                                        )
                                )
                                else (print_endline ("ERROR : Variable(s) has(have) type "^ty^" and RHS
has type "^rhs^". Variables could not be declared.");
                                         incr Ast.error_count;
                                         locals.currMap)
                        )
                        else (print_endline ("ERROR : Variable(s) cannot have type void.");
                                 incr Ast.error_count;
                                 locals.currMap)
                        },
                                (match id_array with
                                        | Array(ident,a,b,Integers(0)) when ty = "Image*"->ty^" "^
ident^"("^string_of_expr a^","^string_of_expr b^ ");\n"
                                        | _ ->   (ty^" "^string_of_id id_array ^ " = "^ str^";\n" )
                                                                                        )         )




(*check_stmt function takes global and local environments and current return type and returns
a modified local environment environment*)


let rec check_stmt (globals,locals) current_func = (function



| Vexpr(Expr(Assign(e,Call("input",act))))->
        let ty,str = check_expr (globals,locals) e in
```

```
		if ty = "Image*" || ty = "Video" || ty = "Pixel" then (print_endline "input is used for images and videos not input."; incr Ast.error_count);
		locals,(
			if List.length act > 1 then (print_endline "Invalid arguments to input" ; incr Ast.error_count);
			if List.length act <> 0 then (
			let typ,strng = check_expr (globals,locals) (List.hd act) in
				if typ <> "string" then (print_endline "Invalid arguments to input" ; incr Ast.error_count);
				"cout << ("^strng^")<< endl;\n cin.clear();\ncin >> "^ str ^";\n"
				)
			else ("\n cin >> "^ str ^";\n"))


	| Vexpr(Expr(Assign(e,Call("open",act))))->
		let ty,str = check_expr (globals,locals) e in
		if ty <> "Image*" && ty <> "Video" then (print_endline "open is used for images and videos only."; incr Ast.error_count);
		if List.length act <> 1  then (print_endline "Invalid arguments to open" ; incr Ast.error_count);
		let typ,strng = check_expr (globals,locals) (List.hd act) in
			if typ <> "string" then (print_endline "Invalid arguments to open" ; incr Ast.error_count);
			locals,("\n{\nstring S = "^strng^";\nFILE *F = fopen(S.c_str(),\"r\");\n "^str^" = BMPReadImage(F);\nfclose(F);\n}\n")


	| Vexpr(Expr(Call("save",act)))->
			if List.length act <> 2 then (print_endline "Invalid arguments to save" ; incr Ast.error_count);
```

```
                let frst::scnd::rem = act in

                let ty1,str1 = check_expr (globals,locals) frst and ty2,str2 = check_expr (globals,locals)
scnd in

                                if ty2 <> "string" || ty1 <> "Image*" then (print_endline "Invalid
arguments to save" ; incr Ast.error_count);

                                locals,("\n{\nstring S = "^str2^";\nFILE *F =
fopen(S.c_str(),\"w\");\nBMPWriteImage(("^str1^"),F);\nfclose(F);\n}\n")




| Vexpr(Expr(Call("input",act)))->

          locals,(

                    if List.length act > 1 then (print_endline "Invalid arguments to input" ; incr
Ast.error_count);

                    if List.length act <> 0 then

                    let ty,str = check_expr (globals,locals) (List.hd act) in

                              (

                                        if ty <> "string" then (print_endline "Invalid arguments to input" ;
incr Ast.error_count);

                                        "cout << ("^str^")<< endl;\n \n cin.clear();\ncin.ignore(2);\n"

                               )

                              else "\ncin.clear();\ncin.ignore(2);\n"

                    )


| Vexpr(Expr(Call("display",actuals))) ->

locals,(

          if List.length actuals = 1 then

          (let ty,str = check_expr (globals,locals) (List.hd actuals) in

                    (
                              if ty = "Image*" || ty = "Video" then "display("^str^");\n"
                                                  else if ty = "Pixel" then (print_endline "cant display
a pixel!!";incr Ast.error_count;"cout<<endl;")
                                                  else ("cout<<"^str^"<<endl;\n")
                                            )
      )
      else
      ( "cout<<" ^ (String.concat "<<" (List.map (fun expr ->
            let ty,str = check_expr (globals,locals) expr in
            (if ty = "Image*" || ty = "Video" || ty = "Pixel" then (print_endline "cant display
a pixel!!";incr Ast.error_count));
            str
            ) actuals))  ^ "<< endl;\n"
      )


)





| For(v_initialize,condition,step,stmt_blk) -> (*modify locals and then check statements then
return modified local before stmt checks*)
(*v_initialize; while (condition) stmt_blk*)
 let str =
 (let locals,strv = check_stmt (globals,locals) current_func (Vexpr(v_initialize)) in
            let cond,stre =  check_expr (globals,locals) condition in
                        if cond <> "bool" then (
                                                  print_endline ("Warning : Condition in for statement
should be a boolean expression, not "^cond^".")
                                        );
                    let _,strs = check_stmt (globals,locals) current_func stmt_blk in
                              ("for ("^String.sub strv 0 (String.length strv -
2)^";"^stre^";"^snd(check_expr (globals,locals) step)^")\n"^strs^"\n")
          )
 in
 locals,str
| For_list(v_initialize,array_id,stmt_blk) -> (*Same as other for loop*)
          locals,"\n\n\n\n\n\n\n\n"
| Vexpr(Vdecl(vdecl)) ->  let locals,str = check_vdecl (globals,locals) vdecl in locals,str
| Vexpr(Expr(expr)) ->  locals,(snd(check_expr (globals,locals) expr)^";\n")
| Block(stmt_list)  ->
          let str_stmt_blk = ("{\n"^
                              (let locals = {outMap = addMap locals.outMap locals.currMap; currMap =
NameMap.empty} in
                                        let _,str_blk =
                    List.fold_left (fun (local,str) stmt ->
                                                            (let lcl,strs = check_stmt (globals,local)
current_func stmt in

          lcl,(str^strs)

                                                            )
                                                  ) (locals,"") stmt_list
                              in str_blk
                              )^ "}\n") in
                    locals,str_stmt_blk
```

```
| Return(expr) ->
        let ty,str = check_expr (globals,locals) expr and current_func = NameMap.find
current_func globals.funMap in
                (if current_func.rtype <> ty then
                        (print_endline ("ERROR: In function '"^current_func.fname^"'
return expression is of type "^ty^" but expected return type is "^current_func.rtype^".\n");
                        incr Ast.error_count));
                    locals,("return "^str^";\n")
| Break -> locals,"break;\n"
| Continue -> locals,"continue;\n"
| If(predicate,then_blk,Block([])) ->

        let cond,strp =  check_expr (globals,locals) predicate in
                    (if cond <> "bool" then (
                        print_endline ("Warning : The predicate of an If statement should
be a boolean expression, not "^cond^"."))
                    );
        let _,strthen    = check_stmt (globals,locals) current_func then_blk in
        locals,("if ("^strp^")"^strthen^"\n")
| If(predicate,then_blk,else_blk) ->
        let cond,strp =  check_expr (globals,locals) predicate in
                    if cond <> "bool" then (
                        print_endline ("Warning : The predicate of an If statement should
be a boolean expression, not "^cond^"."));
        let _,strthen    = check_stmt (globals,locals) current_func then_blk and _,strelse =
check_stmt (globals,locals) current_func else_blk in
        locals,("if ("^strp^") "^strthen^" else "^strelse^"\n")
| While(predicate,stmt_blk) ->
        let cond,strp =  check_expr (globals,locals) predicate in
```

```
                              (if cond <> "bool" then (
                                        print_endline ("Warning : The predicate of a While statement
should be a boolean expression, not "^cond^"."))
                              );
                    let  _,strs = check_stmt (globals,locals) current_func stmt_blk in
          locals,("while (" ^ strp ^ ") "^ strs ^ "\n")
| Do_while(predicate,stmt_blk) ->
          let cond,strp =  check_expr (globals,locals) predicate in
                    (if cond <> "bool" then (
                              print_endline ("Warning : The predicate of a Do-While statement should
be a boolean expression, not "^cond^"."))
                    );
          let _,strs = check_stmt (globals,locals) current_func stmt_blk in

          locals,("do " ^ strs ^ " while (" ^ strp ^ ");\n")
)




let check_function (globals,locals) fdecl =
          let locals = {outMap = globals.varMap;
                                        currMap =
                                                  List.fold_left (fun vmap (ty,id_array) ->
                                                            NameMap.add (
                                                                                (match id_array with
                                                                                          |Single(id) -> id
                                                                                          |Array(id,_,_,_) -> id
                                                            )
                                                      ) ty vmap
                                                ) NameMap.empty fdecl.formals
                  }
            in
                  let _,str =         List.fold_left (fun (local,str) stmt ->
                        let lcl,strs = check_stmt (globals,local) fdecl.fname stmt in
                                    (lcl,(str^strs))
                              ) (locals,"") fdecl.body

            in
            (globals,locals,str)







(*We will take take the Ast and return a global varmap, a funmap, the modified Ast*)
let check_program program =
let globals = {varMap = NameMap.empty;funMap = NameMap.empty} and locals = {outMap =
NameMap.empty;currMap = NameMap.empty} in
let inbuilt_functions = ["display";"main";"open";"save";"input"] in
let (globals,_,program,main) =
      List.fold_left (
            fun (globals,locals,program,main) prog ->
                  (match prog with
                        |Fdefn(f) ->
                            let (globals,locals) =
                                    ({globals with funMap =
                                        if List.mem f.fname
inbuilt_functions then (
                                            print_endline ("ERROR
:'"^f.fname^"' is an inbuilt function.");
                                            incr Ast.error_count;
                                            globals.funMap
                                    )
                                    else if NameMap.mem f.fname
globals.funMap then (
                                            print_endline ("ERROR : The
function '"^f.fname^"' has already been declared.");
                                            incr Ast.error_count;
                                            globals.funMap)
                                    else NameMap.add f.fname f
globals.funMap},locals)
                            in
                            let (_,_,str)=
                                (check_function (globals,locals) f)
                            in
                            (globals,locals,(program^"\n"^f.rtype^" "^f.fname^"("^
(String.concat ", " (List.map Ast.string_of_formal f.formals)) ^")\n{ "^str^" \n}"),main)
                            (*check the statements in the function block here after
saving globals. *)


                        | Stmt(Vexpr(Vdecl(Vassign(ty,ida,expr)))) ->
```

```
                              let local,str = check_vdecl
(globals,{outMap=NameMap.empty;currMap=globals.varMap}) (Vassign(ty,ida,expr)) in

                                                      ({globals with varMap =
local.currMap},locals,program,main^str)

                                        | Stmt(Vexpr(Vdecl(vdecl))) ->

                              let local,str = check_vdecl
(globals,{outMap=NameMap.empty;currMap=globals.varMap}) vdecl in

                                                      ({globals with varMap =
local.currMap},locals,program,main)

                                        | Stmt(Return(_)) ->

                                                print_endline "ERROR: What are you trying to accomplish?
'return' only makes sense when used inside functions";

                                                incr Ast.error_count;

                                                (globals,locals,program,main)

                                        | Stmt(Break) ->

                                                print_endline "ERROR: What are you trying to accomplish?
'break' only makes sense when used inside loops";

                                                incr Ast.error_count;

                                                (globals,locals,program,main)

                                        | Stmt(Continue) ->

                                                print_endline "ERROR: What are you trying to accomplish?
'continue' only makes sense when used inside loops";

                                                incr Ast.error_count;

                                                (globals,locals,program,main)
```

```
                                | Stmt(s) ->
                                        let locals = {outMap = globals.varMap ; currMap =
NameMap.empty}  in
                                                let _,str = check_stmt (globals,locals) "main" s in
                                                        (globals,locals,program,main^str)


                        )
                ) (globals,locals,"","") program
in
(globals.varMap,globals.funMap,program,main)
```

****

ast.ml – Authored by Shubhanshu and Vaibhav

****

```
let error_count = ref 0


type op =
  Add | Sub | Mult | Div | Mod | Pow
 | Equal | Neq | Less | Leq | Greater | Geq | Not



type expr =
  Integers of int
 | Floats of float
 | Strings of string
 | Id of string
 | Var_access of int * expr * (bool * expr * expr) * (bool * expr * expr) * (bool * expr * expr)
 | Boolean of bool
 | Uop of op * expr
 | Binop of expr * op * expr
 | Paren of expr
 | Assign of expr * expr
 | Call of string * expr list
 | Objcall of expr * expr list
 | Objid of expr * string
 | Int_list of expr list
 | Noexpr


type id =
|Single of string
|Array of string * expr * expr * expr


type vdecl =
  Vdefn of string * (id list)
 | Vassign of string * id * expr

type vexpr =
  Vdecl of vdecl
 | Expr of expr

type stmt =
 | Block of stmt list
 | Vexpr of vexpr
 | Return of expr
 | Break
 | Continue
 | If of expr * stmt * stmt
 | For of vexpr * expr * expr * stmt
 | For_list of vexpr * string * stmt
 | While of expr * stmt
 | Do_while of expr * stmt

type func_decl = {
  rtype : string;
  fname : string;
  formals : (string * id) list;
  body : stmt list;
 }

type possibilities =
| Stmt of stmt
| Fdefn of func_decl

type program = possibilities list
```

```
let rec string_of_expr = function
   Integers(l) -> string_of_int l
  | Floats(f) -> string_of_float f
  | Strings(s) -> "\"" ^ s ^ "\""
  | Boolean(b) -> if b then "true" else "false"
  | Id(s) -> s
  | Int_list(l) -> "{"^String.concat "" (List.map string_of_expr l)^"}"
  | Var_access(i,e,(b1,e11,e12),(b2,e21,e22),(b3,e31,e32)) ->
   string_of_expr e ^ (if b1 then "["^string_of_expr e11^":"^string_of_expr e11^"]" else
"["^string_of_expr e11^"]") ^
   if i=1 then ""
   else if i=2 then (if b1 then "["^string_of_expr e11^":"^string_of_expr e12^"]" else
"["^string_of_expr e11^"]")
   else if i=3 then (if b1 then "["^string_of_expr e21^":"^string_of_expr e22^"]" else
"["^string_of_expr e21^"]") ^ (if b1 then "["^string_of_expr e31^":"^string_of_expr e32^"]"
else "["^string_of_expr e31^"]")
   else "Never Occurrs"
  | Binop(e1, o, e2) ->
     string_of_expr e1 ^ " " ^
     (match o with
 Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/" | Mod -> "%"
     | Equal -> "==" | Neq -> "!="
     | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=" | _ -> "") ^ " " ^
     string_of_expr e2
  | Uop(o,e) ->  (match o with
       Sub -> "-" | Not -> "!" | _ -> "") ^ " " ^
```

```
    string_of_expr e
  | Paren(e) -> "(" ^ string_of_expr e ^")"
  | Assign(e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
  | Call(f, el) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Objid(obj,var) -> string_of_expr obj ^ "." ^ var
  | Objcall(obj,act) -> string_of_expr obj ^ "(" ^ String.concat ", " (List.map string_of_expr act) ^
")"
  | Noexpr -> ""



let string_of_id = function
| Single(id) -> id
| Array(id,e1,e2,e3) -> id ^ "[" ^string_of_expr e1 ^"]" ^ "[" ^ string_of_expr e2 ^ "]" ^ "[" ^
string_of_expr e2 ^ "]"

let string_of_id_list id = String.concat "," (List.map string_of_id id)

let string_of_vdecl = function
   Vdefn(ty,id) -> ty ^ " " ^ string_of_id_list id ^ " "
  | Vassign(ty,id,ex) -> ty ^ " " ^ string_of_id id ^ " = " ^ (string_of_expr ex) ^ " "


let string_of_vexpr = function
  Vdecl(vdecl) -> string_of_vdecl vdecl
  |Expr(expr) -> string_of_expr expr
```

```
let rec string_of_stmt = function

    Block(stmts) ->

      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"

  | Vexpr(vexpr) -> string_of_vexpr vexpr ^ ";\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(v1, e2, e3, s) ->

      "for (" ^ string_of_vexpr v1  ^ " ; " ^ (string_of_expr e2) ^ " ; " ^

      (string_of_expr e3)  ^ ") " ^ string_of_stmt s

(*  | For_list(v1,id,s) ->

      "for (" ^ string_of_vexpr v1  ^ " in " ^ string_of_id_access id ^ ") " ^ string_of_stmt s*)

  | While(e, s) -> "while (" ^ (string_of_expr e) ^ ") " ^ string_of_stmt s

  | Do_while(e,s) -> "do " ^ string_of_stmt s ^ " while (" ^ (string_of_expr e) ^ ");\n"

  | Return(ex) -> "return "^ (string_of_expr ex) ^ ";\n"

  | Break -> "break ;\n"

  | Continue -> "continue ;\n"



let string_of_formal (ty,id) = ty ^ " " ^ string_of_id id



let string_of_fdefn fdefn =

  if fdefn.fname <> "" then

  fdefn.rtype ^ " " ^ fdefn.fname ^ "(" ^ String.concat ", " (List.map string_of_formal
fdefn.formals) ^ ")\n{\n" ^
```

```
 String.concat "" (List.map string_of_stmt fdefn.body) ^
 "}\n"
 else ""


let string_of_prog = function
   Stmt(s) -> string_of_stmt s
  | Fdefn(f) -> string_of_fdefn f

let string_of_program program =
 String.concat "" (List.map string_of_prog program ) ^ "\n"
```

****

codegen.ml – Authored by Vaibhav

****

```
open Printf
open Sast
open Ast
```

```
let string_of_fdecl fdefn =
  fdefn.rtype ^ " " ^ fdefn.fname ^ "(" ^ String.concat ", " (List.map string_of_formal
fdefn.formals) ^ ")"




let make_program (varmap,funmap,program,main,filename) =
let convert_program program main global_vars funcs =
("
#include \"/home/vaibhav/plt_project/Vaibhav/c_source/image.h\"
#include \"/home/vaibhav/plt_project/Vaibhav/c_source/bmp.h\"
#include \"CImg.h\"
#include <assert.h>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include \"/home/vaibhav/plt_project/Vaibhav/c_source/extra.h\"

using namespace std;

"^

 String.concat ";\n" (List.rev (NameMap.fold (
            fun id ty a -> ( ty ^" "^ id )::a
                      ) global_vars []))   ^";\n\n"^
 String.concat ";\n" (List.rev (NameMap.fold (
          fun fname fdecl a -> (string_of_fdecl fdecl)::a
                  ) funcs []))   ^";\n\n"^

 "int main(void){\n"^

 main
(* String.concat "" (List.map string_of_stmt program) ^ *)

 ^"\nreturn 0;\n}\n\n"^

 program



)



in
if !Ast.error_count <> 0  then
          raise (Failure ("Program could not be compile because there are "^string_of_int
!Ast.error_count^"  errors"))
else
let file = String.sub filename 0 (String.length filename - 5) ^".cpp"  in
let listing = convert_program program main varmap funmap  in
          let outChannel = open_out file in
                  (fprintf outChannel "%s" (listing);
                        close_out outChannel;

                        ignore (Sys.command ("g++  -o " ^  (String.sub filename 0 (String.length
filename -5)) ^ " " ^ file ^" -O2 -L/usr/X11R6/lib -lm -lpthread -lX11 -DUSE_UNIX -g -ansi -Wall
./c_source/bmp.cpp ./c_source/pixel.cpp ./c_source/image.cpp ./c_source/vector.cpp " ));

                        print_endline "Program has compiled successfully"

                        )
```

****

svipc.ml – Authored by Shubhanshu Yadav

****

```
type action = Ast | Compile | Test

exception Illegal_character of string * int


let _ =

  let (filename,action) =

  if Array.length Sys.argv > 2 then

    (Sys.argv.(2),List.assoc Sys.argv.(1) [ ("-a", Ast);

         ("-c", Compile);

         ("-t", Test)])

  else (Sys.argv.(1),Compile)

  in

  let len = String.length filename in

  if String.sub filename (len - 5) 5  = ".svip" then

    let lexbuf = Lexing.from_channel (open_in filename) in
```

```
 let program = Parser.program Scanner.token lexbuf   in

 match action with

  | Ast -> let listing = Ast.string_of_program program
        in print_endline listing;
          print_endline ("\nProgram has been parsed with " ^ string_of_int !Ast.error_count ^ " errors.");
  | Test ->   let (_,_,_,_) = Sast.check_program program in
          if !Ast.error_count <> 0  then
          raise (Failure ("Program has "^string_of_int !Ast.error_count^"  errors"))
          else
          print_endline ("Program has no errors.")

  | Compile -> let (a,b,c,d) = Sast.check_program program in
            Codegen.make_program (a,b,c,d,filename)

else print_endline "Error reading the file. It is only possible to compile a '.svip' file"
```

****

makefile – Authored by Bhargav

****

```
OBJS = ast.cmo sast.cmo codegen.cmo parser.cmo scanner.cmo svipc.cmo

TARFILES = Makefile testall.sh scanner.mll parser.mly \
	ast.ml execute.ml svipc.ml \
	$(TESTS:%=tests/test-%.mc) \
	$(TESTS:%=tests/test-%.out)

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

.PHONY : test
test : svipc testall.sh
	./testall.sh

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc -v parser.mly

%.cmo : %.ml
```

```
	ocamlc -c $<

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

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

.PHONY : clean
clean :
	rm -f parser.output parser.ml parser.mli scanner.ml testall.log \
	*.cmo *.cmi *.out *.diff

.PHONY : clean_all
clean_all :
	rm -f svipc parser.output parser.ml parser.mli scanner.ml testall.log \
	*.cmo *.cmi *.out *.diff && find ./tests/ ! -name "*.svip" -type f -delete

# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
sast.cmo: ast.cmo
sast.cmx: ast.cmx
codegen.cmo: bytecode.cmo ast.cmo
codegen.cmx: bytecode.cmx ast.cmx
svipc.cmo: scanner.cmo parser.cmi sast.cmo codegen.cmo \
    ast.cmo
```

```
svipc.cmx: scanner.cmx parser.cmx sast.cmx codegen.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
sast.cmo: ast.cmo
sast.cmx: ast.cmx
```

****

Bmp.cpp – Authored by Bhargav

****

```
#include "bmp.h"
#include <assert.h>
#include <stdlib.h>



#define DEBUG_HEADERS 0



typedef char BYTE;              /* 8 bits */
typedef unsigned short int WORD;  /* 16-bit unsigned integer. */
typedef unsigned int DWORD;       /* 32-bit unsigned integer */
typedef int LONG;                 /* 32-bit signed integer */



typedef struct tagBITMAPFILEHEADER {
    WORD bfType;
    DWORD bfSize;
    WORD bfReserved1;
    WORD bfReserved2;
    DWORD bfOffBits;
} BITMAPFILEHEADER;



typedef struct tagBITMAPINFOHEADER {
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;
    WORD biPlanes;
    WORD biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    LONG biXPelsPerMeter;
    LONG biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER;



/* constants for the biCompression field */
#define BI_RGB        0L
#define BI_RLE8       1L
#define BI_RLE4       2L
#define BI_BITFIELDS  3L



typedef struct tagRGBTRIPLE {
    BYTE rgbtBlue;
    BYTE rgbtGreen;
    BYTE rgbtRed;
} RGBTRIPLE;



typedef struct tagRGBQUAD {
    BYTE rgbBlue;
    BYTE rgbGreen;
    BYTE rgbRed;
    BYTE rgbReserved;
} RGBQUAD;
```

```
/* Some magic numbers */

#define BMP_BF_TYPE 0x4D42
/* word BM */

#define BMP_BF_OFF_BITS 54
/* 14 for file header + 40 for info header (not sizeof(), but packed size) */

#define BMP_BI_SIZE 40
/* packed size of info header */



/* Reads a WORD from a file in little endian format */
static WORD WordReadLE(FILE *fp)
{
   WORD lsb, msb;

   lsb = getc(fp);
   msb = getc(fp);
   return (msb << 8) | lsb;
}
```

```
/* Writes a WORD to a file in little endian format */
static void WordWriteLE(WORD x, FILE *fp)
{
   BYTE lsb, msb;

   lsb = (BYTE) (x & 0x00FF);
   msb = (BYTE) (x >> 8);
   putc(lsb, fp);
   putc(msb, fp);
}



/* Reads a DWORD word from a file in little endian format */
static DWORD DWordReadLE(FILE *fp)
{
   DWORD b1, b2, b3, b4;

   b1 = getc(fp);
   b2 = getc(fp);
   b3 = getc(fp);
   b4 = getc(fp);
   return (b4 << 24) | (b3 << 16) | (b2 << 8) | b1;
}
```

```
/* Writes a DWORD to a file in little endian format */
static void DWordWriteLE(DWORD x, FILE *fp)
{
   unsigned char b1, b2, b3, b4;

   b1 = (x & 0x000000FF);
   b2 = ((x >> 8) & 0x000000FF);
   b3 = ((x >> 16) & 0x000000FF);
   b4 = ((x >> 24) & 0x000000FF);
   putc(b1, fp);
   putc(b2, fp);
   putc(b3, fp);
   putc(b4, fp);
}



/* Reads a LONG word from a file in little endian format */
static LONG LongReadLE(FILE *fp)
{
   LONG b1, b2, b3, b4;

   b1 = getc(fp);
   b2 = getc(fp);
   b3 = getc(fp);
   b4 = getc(fp);
```

```
    return (b4 << 24) | (b3 << 16) | (b2 << 8) | b1;
}



/* Writes a LONG to a file in little endian format */
static void LongWriteLE(LONG x, FILE *fp)
{
    char b1, b2, b3, b4;

    b1 = (x & 0x000000FF);
    b2 = ((x >> 8) & 0x000000FF);
    b3 = ((x >> 16) & 0x000000FF);
    b4 = ((x >> 24) & 0x000000FF);
    putc(b1, fp);
    putc(b2, fp);
    putc(b3, fp);
    putc(b4, fp);
}



Image *BMPReadImage(FILE *fp)
{
    Image *img = NULL;
    BITMAPFILEHEADER bmfh;
    BITMAPINFOHEADER bmih;
    int x, y;
    int lineLength;

    assert(fp != NULL);

    /* Read file header */

    /* fread(&bmfh, sizeof(bmfh), 1, fp); */
    /* fread won't work on different platforms because of endian
     * issues.  Sigh... */

    bmfh.bfType = WordReadLE(fp);
    bmfh.bfSize = DWordReadLE(fp);
    bmfh.bfReserved1 = WordReadLE(fp);
    bmfh.bfReserved2 = WordReadLE(fp);
    bmfh.bfOffBits = DWordReadLE(fp);

    /* Debug file header */
#if DEBUG_HEADERS
    fprintf(stderr, "file header:\n");
    fprintf(stderr, "\tbfType: %x\n", (int) bmfh.bfType);
    fprintf(stderr, "\tbfSize: %d\n", (int) bmfh.bfSize);
    fprintf(stderr, "\tbfReserved1: %d\n", (int) bmfh.bfReserved1);
    fprintf(stderr, "\tbfReserved2: %d\n", (int) bmfh.bfReserved2);
    fprintf(stderr, "\tbfOffBits: %d\n", (int) bmfh.bfOffBits);
#endif
```

```
    /* Check file header */
    assert(bmfh.bfType == BMP_BF_TYPE);
    /* ignore bmfh.bfSize */
    /* ignore bmfh.bfReserved1 */
    /* ignore bmfh.bfReserved2 */
    assert(bmfh.bfOffBits == BMP_BF_OFF_BITS);

    /* Read info header */

    /* fread(&bmih, sizeof(bmih), 1, fp); */
    /* same problem as above... */

    bmih.biSize = DWordReadLE(fp);
    bmih.biWidth = LongReadLE(fp);
    bmih.biHeight = LongReadLE(fp);
    bmih.biPlanes = WordReadLE(fp);
    bmih.biBitCount = WordReadLE(fp);
    bmih.biCompression = DWordReadLE(fp);
    bmih.biSizeImage = DWordReadLE(fp);
    bmih.biXPelsPerMeter = LongReadLE(fp);
    bmih.biYPelsPerMeter = LongReadLE(fp);
    bmih.biClrUsed = DWordReadLE(fp);
    bmih.biClrImportant = DWordReadLE(fp);

    /* Debug info header */
#if DEBUG_HEADERS
    fprintf(stderr, "info header:\n");
    fprintf(stderr, "\tbiSize: %d\n", (int) bmih.biSize);
    fprintf(stderr, "\tbiWidth: %d\n", (int) bmih.biWidth);
    fprintf(stderr, "\tbiHeight: %d\n", (int) bmih.biHeight);
    fprintf(stderr, "\tbiPlanes: %d\n", (int) bmih.biPlanes);
    fprintf(stderr, "\tbiBitCount: %d\n", (int) bmih.biBitCount);
    fprintf(stderr, "\tbiCompression: %d\n", (int) bmih.biCompression);
    fprintf(stderr, "\tbiSizeImage: %d\n", (int) bmih.biSizeImage);
    fprintf(stderr, "\tbiXPelsPerMeter: %d\n", (int) bmih.biXPelsPerMeter);
    fprintf(stderr, "\tbiYPelsPerMeter: %d\n", (int) bmih.biYPelsPerMeter);
    fprintf(stderr, "\tbiClrUsed: %d\n", (int) bmih.biClrUsed);
    fprintf(stderr, "\tbiClrImportant: %d\n", (int) bmih.biClrImportant);
#endif

    /* Check info header */
    assert(bmih.biSize == BMP_BI_SIZE);
    assert(bmih.biWidth > 0);
    assert(bmih.biHeight > 0);
    assert(bmih.biPlanes == 1);
    assert(bmih.biBitCount == 24);  /* RGB */
    assert(bmih.biCompression == BI_RGB);   /* RGB */
    lineLength = bmih.biWidth * 3;  /* RGB */
    if ((lineLength % 4) != 0) {
    lineLength = (lineLength / 4 + 1) * 4;
    }
    assert(bmih.biSizeImage == (DWORD) lineLength * (DWORD) bmih.biHeight);
    /* Creates the image */
    img = new Image(bmih.biWidth, bmih.biHeight);

    /* Read triples */
    /* RGB */
    {
    RGBTRIPLE *triples;
    triples = (RGBTRIPLE *)malloc(lineLength);
    assert(triples != NULL);
    fseek(fp, (long) bmfh.bfOffBits, SEEK_SET);

    for (y = 0; y < img->height; y++) {
        fread(triples, 1, lineLength, fp);
        assert(ferror(fp) == 0);

        /* Copy triples */
        for (x = 0; x < img->width; x++)
            img->GetPixel(x, img->height-y-1).Set(
                triples[x].rgbtRed,
                triples[x].rgbtGreen,
                triples[x].rgbtBlue,
                255);
    }
    free(triples);
    }
    return img;
}
```

```
void BMPWriteImage(Image *img, FILE *fp)
{
    assert(img);

    BITMAPFILEHEADER bmfh;
    BITMAPINFOHEADER bmih;
    int x, y;
    int lineLength;

    lineLength = img->width * 3;    /* RGB */
    if ((lineLength % 4) != 0) {
    lineLength = (lineLength / 4 + 1) * 4;
    }
    /* Write file header */

    bmfh.bfType = BMP_BF_TYPE;
    bmfh.bfSize = BMP_BF_OFF_BITS + lineLength * img->height;
    bmfh.bfReserved1 = 0;
    bmfh.bfReserved2 = 0;
    bmfh.bfOffBits = BMP_BF_OFF_BITS;

    /* Debug file header */
#if DEBUG_HEADERS
    fprintf(stderr, "file header:\n");
    fprintf(stderr, "\tbfType: %x\n", bmfh.bfType);

    fprintf(stderr, "\tbfSize: %d\n", (int) bmfh.bfSize);

    fprintf(stderr, "\tbfReserved1: %d\n", (int) bmfh.bfReserved1);

    fprintf(stderr, "\tbfReserved2: %d\n", (int) bmfh.bfReserved2);

    fprintf(stderr, "\tbfOffBits: %d\n", (int) bmfh.bfOffBits);

#endif


    WordWriteLE(bmfh.bfType, fp);

    DWordWriteLE(bmfh.bfSize, fp);

    WordWriteLE(bmfh.bfReserved1, fp);

    WordWriteLE(bmfh.bfReserved2, fp);

    DWordWriteLE(bmfh.bfOffBits, fp);


    /* Write info header */


    bmih.biSize = BMP_BI_SIZE;

    bmih.biWidth = img->width;

    bmih.biHeight = img->height;

    bmih.biPlanes = 1;

    bmih.biBitCount = 24;       /* RGB */

    bmih.biCompression = BI_RGB;    /* RGB */

    bmih.biSizeImage = lineLength * (DWORD) bmih.biHeight;  /* RGB */

    bmih.biXPelsPerMeter = 2925;

    bmih.biYPelsPerMeter = 2925;

    bmih.biClrUsed = 0;

    bmih.biClrImportant = 0;
    /* Debug info header */
#if DEBUG_HEADERS
    fprintf(stderr, "info header:\n");
    fprintf(stderr, "\tbiSize: %d\n", (int) bmih.biSize);
    fprintf(stderr, "\tbiWidth: %d\n", (int) bmih.biWidth);
    fprintf(stderr, "\tbiHeight: %d\n", (int) bmih.biHeight);
    fprintf(stderr, "\tbiPlanes: %d\n", (int) bmih.biPlanes);
    fprintf(stderr, "\tbiBitCount: %d\n", (int) bmih.biBitCount);
    fprintf(stderr, "\tbiCompression: %d\n", (int) bmih.biCompression);
    fprintf(stderr, "\tbiSizeImage: %d\n", (int) bmih.biSizeImage);
    fprintf(stderr, "\tbiXPelsPerMeter: %d\n", (int) bmih.biXPelsPerMeter);
    fprintf(stderr, "\tbiYPelsPerMeter: %d\n", (int) bmih.biYPelsPerMeter);
    fprintf(stderr, "\tbiClrUsed: %d\n", (int) bmih.biClrUsed);
    fprintf(stderr, "\tbiClrImportant: %d\n", (int) bmih.biClrImportant);
#endif

    DWordWriteLE(bmih.biSize, fp);
    LongWriteLE(bmih.biWidth, fp);
    LongWriteLE(bmih.biHeight, fp);
    WordWriteLE(bmih.biPlanes, fp);
    WordWriteLE(bmih.biBitCount, fp);
    DWordWriteLE(bmih.biCompression, fp);
    DWordWriteLE(bmih.biSizeImage, fp);
    LongWriteLE(bmih.biXPelsPerMeter, fp);
    LongWriteLE(bmih.biYPelsPerMeter, fp);
    DWordWriteLE(bmih.biClrUsed, fp);
    DWordWriteLE(bmih.biClrImportant, fp);
    /* Write pixels */
    for (y = 0; y < img->height; y++) {
    int nbytes = 0;
    for (x = 0; x < img->width; x++) {
        Pixel p = img->GetPixel(x, img->height-y-1);
        putc(p.b, fp), nbytes++;
        putc(p.g, fp), nbytes++;
        putc(p.r, fp), nbytes++;
        /* putc(p.a, fp), nbytes++; */
        /* RGB */
    }
    /* Padding for 32-bit boundary */
    while ((nbytes % 4) != 0) {
        putc(0, fp);
        nbytes++;
    }
    }
}
```

****

Bmp.h – Authored by Bhargav

****

```
#ifndef BMP_INCLUDED
#define BMP_INCLUDED

#include "image.h"


// Reads an image from a BMP file.
Image *BMPReadImage(FILE *fp);

// Writes an image to a BMP file.
void BMPWriteImage(Image *img, FILE *fp);


#endif
```

****

image.cpp – Authored by Bhargav

****

```
#include "image.h"

#include "bmp.h"

#include <math.h>

#include <stdlib.h>

#include <string.h>

#include <float.h>



/**

 * Image

 **/

Image::Image (int width_, int height_)

{

    assert(width_ > 0);

    assert(height_ > 0);


    width          = width_;

    height         = height_;

    num_pixels     = width * height;

    pixels         = new Pixel[num_pixels];

    sampling_method = IMAGE_SAMPLING_POINT;
    assert(pixels != NULL);
}


Image::Image (const Image& src)
{
    width          = src.width;
    height         = src.height;
    num_pixels     = width * height;
    pixels         = new Pixel[num_pixels];
    sampling_method = IMAGE_SAMPLING_POINT;

    assert(pixels != NULL);
    memcpy(pixels, src.pixels, src.width * src.height * sizeof(Pixel));
}


Image::~Image ()
{
    delete [] pixels;
    pixels = NULL;
}



/* Error-diffusion parameters */
```

```
const double
    ALPHA = 7.0 / 16.0,
    BETA  = 3.0 / 16.0,
    GAMMA = 5.0 / 16.0,
    DELTA = 1.0 / 16.0;
```

****

image.h – Authored by Bhargav. Assisted by Shubhanshu and Vaibhav

****

```
#ifndef IMAGE_INCLUDED
#define IMAGE_INCLUDED

#include <assert.h>
#include <stdio.h>
#include "pixel.h"
```

```
#include "vector.h"



/**
 * constants
 **/
enum {
   IMAGE_SAMPLING_POINT,
   IMAGE_SAMPLING_BILINEAR,
   IMAGE_SAMPLING_GAUSSIAN,
   IMAGE_N_SAMPLING_METHODS
};

enum {
   IMAGE_CHANNEL_RED,
   IMAGE_CHANNEL_GREEN,
   IMAGE_CHANNEL_BLUE,
   IMAGE_CHANNEL_ALPHA,
   IMAGE_N_CHANNELS
};



/**
 * Line defined as two points (vectors).
 **/
struct Line
{
    Vector p, q;
};



/**
 * Image
 **/
class Image
{
public:
    Pixel *pixels;
    int width, height, num_pixels;
    int sampling_method;

public:
    // Creates a blank image with the given dimensions
    Image (int width, int height);

    // Copy iamage
    Image (const Image& src);

    // Destructor
    ~Image ();

    // Pixel access
    int ValidCoord (int x, int y)  { return x>=0 && x<width && y>=0 && y<height; }
```

```
  Pixel& GetPixel (int x, int y) { assert(ValidCoord(x,y)); return pixels[y*width + x]; }

  // Dimension access
  int Width     () { return width; }
  int Height    () { return height; }
  int NumPixels () { return num_pixels; }


#endif
```

****

pixel.h – Authored by Bhargav. Assisted by Shubhanshu and Vaibhav

****

```
#ifndef PIXEL_INCLUDED
#define PIXEL_INCLUDED


/**
 * Component, fundamental type
 **/
typedef unsigned char Component;

// Confines a component in the range [0..255]
inline Component ComponentClamp(int i)
{ return (i<0) ? 0 : (i>255) ? 255 : i; }

// Returns a random number in the range [0.255]
Component ComponentRandom(void);

// Scales the component by the given factor
Component ComponentScale(Component c, double f);

// Linear interpolation of the components
// Returns (1 - t) * c + t * d
Component ComponentLerp(Component c, Component d, double t);
```

```
/**
 * Pixel
 **/
struct Pixel
{
   // Data
   Component r, g, b, a;

   // Constructor
   Pixel (Component r_=0, Component g_=0, Component b_=0, Component a_=255) : r(r_),
g(g_), b(b_), a(a_) {}

   // Set
   void Set (Component  r_, Component  g_, Component  b_, Component  a_) { r=r_; g=g_; b=b_;
a=a_; }
   void Set (Component  r_, Component  g_, Component  b_)              { r=r_; g=g_; b=b_; }

   void SetClamp (double r_, double g_, double b_);
   void SetClamp (double r_, double g_, double b_, double a_);

   // Returns the luminance of the pixel.
   Component Luminance ();
};

// Returns a random pixel.
Pixel PixelRandom (void);
```

```
// Component-wise addition of pixels.

Pixel operator+ (const Pixel& p, const Pixel& q);


   // Component-wise multiplication of pixels.

Pixel operator* (const Pixel& p, const Pixel& q);


// Component-wise multiplication of pixel by scalar.

Pixel operator* (const Pixel& p, double f);

#endif
```