Columbia University

Programming Languages and Translators

COMS W4115, Fall 2016

Final Report: LéPix

Members:
Fatima Koli
Gabrielle A. Taylor
Jackie Lin
Akshaan Kakar

Supervisor:
Stephen A. Edwards

December 21, 2016

1 Introduction

LéPix is a graphics processing language based loosely on a subset of the C language. Using an imperative style with strong static typing, we support primitives that enable quick and concise programs for image creation and manipulation. The LéPix programming language enables the writing of computer vision and computer graphics algorithms in LéPix with relative ease compared to other languages.

1.1 Background

Image editing and manipulation algorithms rely heavily on array and matrix data structures. Images are represented as 2 dimensional arrays with multiple channels (e.g. RGB) or as higher dimensional arrays. Most languages have in-built syntax for the construction of 1-D arrays, but require complicated syntactic constructions for 2-D arrays. With the LéPix language, we aim to provide an easy-to-use syntax to initialize 1-D and 2-D arrays along with simple expressions for manipulating and printing them.

1.2 Goals

1.2.1 Ease of use

The primary goal of the LéPix language is to enable array and matrix based image editing in an easy to use environment. They syntax of LéPix reminiscent of Swift and C/C++, allowing a user to learn the language and express complex constructs easily and rapidly. The LéPix language also provides syntactic sugar that makes array and matrix declaration and manipulation intuitive.

1.2.2 Flexibility

We have designed the LéPix compiler keeping in mind the extensibility of the system to include features that we do no currently support, such as objects, structures etc.

2 Language Tutorial

2.1 Hello World!

This is an example of a Hello World program in LéPix. It creates an array from an initializer, and then proceeds to print it to the command line.

```

fun main () : int {
    print(24);
    return 0;
}
```

Listing 1: hello world

2.2 Variables and Declarations

Variables are made with the var declaration. You can declare and assign variables by giving them a name and then referencing that name in other places.

```
fun main () : int {
    var a : int = 24 * 2 + 1;
    // a == 49
    var b : int = a - 48;
    // b == 1
    var c : int[[5, 2]] = [
    0, 2, 4, 6, 8, 10;
        1, 3, 5, 7, 9, 11;
   ];
    var value : int = a + b + c[0, 4];
    // value == 58
    return value;
```

```
}
```

Listing 2: Variable Declaration

2.3 Functions

Functions can be called with a simple syntax. The goal is to make it easy to pass arguments and specify types on those arguments, as well as the return type. All functions are defined by starting with the fun keyword, followed by an identifier including the name, before an optional list of parameters.

```

fun sum (a: int, b: int) : int {
      int a = 2;
      int b = 3;
      return a + b;
}

fun main () : int {
      return sum(a,b);
}
```

Listing 3: Function Declaration and Invocation

2.4 Control Flow

Control flow is important for programs to exhibit more complex behaviors. LéPix has for and while constructs for looping, as well as if, else if, else statements. They can be used as in the following sample:

```
fun main(): int
{
  var x:int;
  x = 5;
  if (x < 6) {
      print(42);
  }
  else {
      print(17);
  }
  return 0;
}
```

Listing 4: Control Flow

3 Language Manual

3.1 Expressions, Operations and Types

3.1.1 Variable Names and Identifiers

Identifiers

1. All names for all identifiers in a LéPix program must be composed of a single start alpha codepoint followed by either zero or more of a digit or an alpha codepoint. Any identifier that does not follow this scheme and does not form a valid keyword, literal or definition is considered ill-formed.

3.1.2 Literals

Kinds of Literals

LéPix supports the following literals:

literal:

boolean-literal
integer-literal
floating-literal

Boolean Literals

1. A boolean literal are the keywords `true` or `false`.

Integer Literals

1. An integer literal is a valid sequence of digits.

2. A decimal integer literal uses digits '0' through '9' to define a base-10 number.

Floating Literals

1. A floating literal has two primary forms, utilizing digits as defined in 3.1.2.

2. The first form must have a dot '.' preceded by an integer literal and/or suffixed by an integer literal. It must have one or the other, and may not omit both the prefixing or suffixing integer literal.

3. The second form follows 2, but includes the exponent symbol `e` and another integer literal describing that exponent. Both the exponent and integer literal must be present in this form, but if the exponent is included then the dot is not necessary and may be prefixed with only an integer literal or just an integer literal and a dot.

3.1.3 Variable Declarations

`var` declarations

variable-initialization:

var <identifier> : <type>;

1. A variable can be declared using the `var` keyword, an identifier as defined in 3.1.1 and optionally followed by a colon ':' and type name. This is called a `variable declaration`.

2. A variable declared with `var` is mutable. Mutable variables can have their values re-assigned after declaration and initialization.

3. A declaration can appear inside function bodies or as globals. It cannot appear in the scope of control flow blocks.

3.1.4 Initialization

Variable Initialization

variable-declaration:

var *<identifier> : <type>; = expression ;*

1. Initialization is the assignment of an expression on the right side to a variable declaration.
2. If the expression cannot directly initialize the type on the left, then the program is ill-formed.

3.2 Assignment

assignment-expression:

expression = expression

3.2.1 Arithmetic Expressions

Binary Arithmetic Operations

addition-expression:

expression + expression

subtraction-expression:

expression − expression

division-expression:

expression / expression

multiplication-expression:

*expression * expression*

1. Symbolic expression to perform the commonly understood mathematical operations on two operands.
2. All operations are left-associative.

3.3 Unary Arithmetic Operations

unary-minus-expression:

$-$*expression*

1. Unary minus is typically interpreted as negation of the single operand.
2. All operations are left-associative.

3.3.1 Logical Expressions

Binary Compound Boolean Operators

and-expression:

expression and expression

expression && expression

or-expression:

expression or expression

expression || expression

1. Symbolic expressions to check for logical conjunction and disjunction.
2. For the and−expression, short-circuiting logic is applied if the expression on the left evaluates to false. The right hand expression will not be evaluated.
3. For the or−expression, short-circuiting logic is applied if the expression on the left evaluates to true. The right hand expression will not be evaluated.
4. All operations are left associative.

Binary Relational Operators

equal-to-expression:

expression == *expression*

not-equal-to-expression:

expression != *expression*

less-than-expression:

expression < *expression*

greater-than-expression:

expression > *expression*

less-than-equal-to-expression:

expression <= *expression*

greater-than-equal-to-expression:

expression >= *expression*

1. Symbolic expression to perform relational operations meant to do comparisons.

2. All operations are left-associative.

3.4 Functions

3.4.1 Functions and Function Declarations

Functions are independent code that perform a particular task. They can appear in any order and in one or many source files, but cannot be split among source files.

Function Definitions

```
fun <identifier> ([<parameter_declarations>]) : <return_type
    > {
      <function_body>
      [return <expression>;]
}
```

1. All function definitions in LéPix are of the above form where they begin with the keyword fun, followed by the identifier, a list of optional parameter declarations enclosed in parentheses,the return type, and the function body with an optional return statement.

2. return types can be variable types or void.

3. Functions that return void can either omit the return statement or leave it in or return the value.

4. Functions that return any other variable type must include a return statement and the expression in the return statement must evaluate to the same type as the return type.

5. Function input parameters are passed by value.

3.5 Function Scope and Parameters

1. Variables are declared as usual within the body of a function. The variables declared within the body of a function exist only in the scope of the function and are discarded when they go out of scope.

2. External variables are passed into functions as parameters. All variable types are passed by value.

3. Passing value copies the object, meaning changes are made to the copy within the function and not the original.

4. To pass by value to a function, use the variable name: `add ( x, y );`

3.6 Data Types

3.6.1 Primitive and Derived Types

The types of the language are divided into two categories: primitive types and data types derived from those primitive types. The primitive types are the boolean type, the integral type int, and the floating-point type float. The derived type is Array.

Primitive Data Types

1. `int`

   By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of 2^{32}.

2. `float`

   The float data type is a single precision 32-bit IEEE 754 floating point.

3. `boolean`

   The boolean data type has possible values true and false.

Derived Data Types

1. `array`

   An array is a container object that holds a fixed number of values of a single type. Multi-dimensional arrays are also supported. They need to have arrays of the same length at each level.

3.7 Program Structure and Control Flow

3.7.1 Statements

1. Any expression followed by a semicolon becomes a statement. For example, the expressions x = 2, return x become statements:

```
x = 2;
foo(x);
return x;
```

2. The semicolon is used in this way as a statement terminator.

3.7.2 Blocks and Scope

Braces { and } are used to group statements in to blocks. Braces that surround the contents of a function are an example of grouping statements like this. Statements in the body of a `for`, `while`, or `if` are also surrounded in braces, and therefore also contained in a block. Variables declared within a block exist only in that block. A semicolon is not required after the right brace.

3.7.3 Scope

1. Scopes are defined as the collection of identifiers and available within the current lexicographic block[1].

2. Every program is implicitly surrounded by braces, which define the **global block**.

3.7.4 Variable Scope

1. Variables are in scope only within their own block[2].

2. Variables declared within blocks last only within lifetime of that block.

3. If a variable with a particular identifier has been declared and the identifier is re-used within a nested block, the original definition of the identifier is **shadowed** and the new one is used until the end of the block.

[1] This is usually between two curly braces {}

[2] E.g., between the brackets {}

4. Variables are constructed, that is, stored in memory when they are first encountered in their scope, and destructed at the scope's end in the reverse order they were encountered in.

3.8 Function Scope

1. Function definitions define a new block, which each have their own scope.

2. Function definitions have access to any variables within their surrounding scope, however anything defined in the function definition's block is not accessible in the surrounding blocks.

3. Variables defined in a parameter list belong to the definition-scope of the function.

3.9 Control Flow Scope

1. Control flow also introduces a new block with its own scope.

3.9.1 if

```
if (expression; expression; ...) {
    statements
} else {
    alternative-statements
}
```

1. if statements are used to make decisions in control flow.

2. Variations on this syntax are permitted, e.g. The else block of the if statement is optional.

3. If the expression is evaluated and returns true, then the first portion of the if statement is executed. Otherwise, if there is an else the portion after it is executed, and if there is none then the function continues at the next statement.

4. If statements can also be nested so that multiple conditions can be tested.

3.9.2 while

```
while (condition) {
      statements
}
```

1. while loops are used to repeat a block of code until some condition is met.
2. Every time a loop condition evaluates to true, the while loop's block and statements are executed.
3. When the condition evaluates to false, the while loop's execution is stopped.
4. Loops are dangerous because they can potentially run forever. Make sure your conditions are done properly.

3.9.3 for

1. For loops are another way to repeat a group of statements multiple times. In LéPix, for loops use C-style declarations.

```
for (x = 1; x <= 10; x = x + 1) {
      arr[x] = 1;
}
```

4 Project Plan

4.1 Timeline

- September 21: Decide what kind of language we will be creating and what we expect the syntax to look like so we can write up the proposal.

- September 28: Proposal Due
- October 10: Decided whether we will be adding multicore support or programming to a GPU.
- October 16: Finalize the syntax of the language.
- October 26: Language Reference Manual Due
- November 10: Complete the AST and Parser, which should have no shift-reduce conflicts. Start working on Semantic Analyzer. Start creating test files and a regression test suite.
- November 21: Have working codegen to be able to run the Hello World Program
- December 15: Regression testing. Continue working on Semantic Analyzer, Semantic AST, and Codegen.
- December 19: Project Demo

4.2 Responsibilities

Roles were shifted around since we had a group member leave, but listed below are the roles we initially took on and the responsibilities we ended up having.

- Manager: Fatima.

  Fatima wrote the Parser and AST with Akshaan. She collaborated on codegen with the rest of the members. She wrote up functions for filtering images, such as grey-scale, blurring, etc along with Akshaan.

- System Architect: Akshaan.

  Akshaan wrote the Lexer. He worked on the Parser and AST with Fatima. He wrote the Semantic Analyzer and Semantic AST.

- Language Guru: Gabrielle.

  Gabrielle helped decide on syntax of the language and worked on codegen through developing C programs and using their LLVM output to work backwards and figure out what to put into codegen. She wrote up functions for filtering images, such as filtering by color etc and ran them on test images to establish our demo.

- Tester: Jackie.

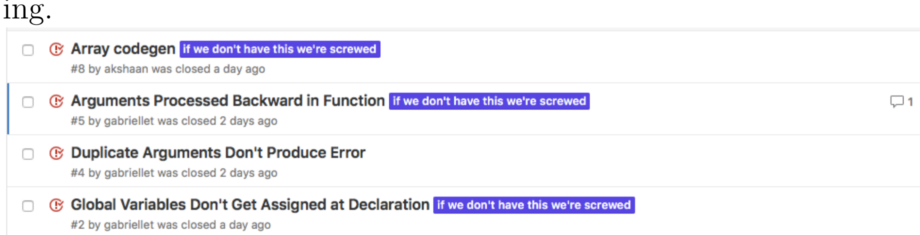
  Jackie wrote all the test cases and set up testing on Travis with Gabrielle. She checked to make sure our code gave the right output for well-formed code and also made sure to check that it failed on code that it shouldn't run on.

- Codegen: All the members collaborated together on codegen.

4.2.1 Development Tools

For this project, we used:

- OCaml
- Github for version control, collaborative development and issue tracking.



- Travis for testing and continuous integration.
- Clang for generating LLVM IR from C programs which we then tried to emulate in our codegen.

Our testing environment on Travis used Ubuntu Trusty (14.04).

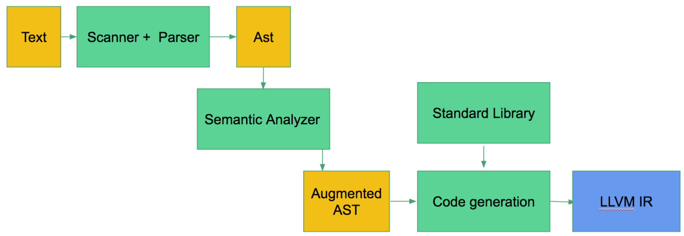
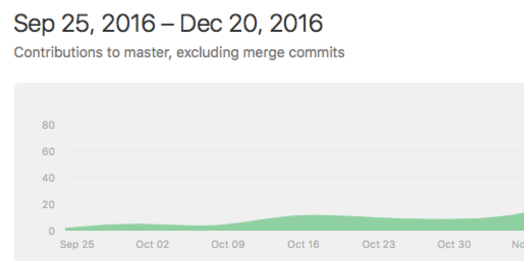


Figure 1: Schematic diagram of the major subsystems in Sylvan

4.2.2 Project Log

For approximately half of this project, we were on schedule. After a setback in November where we were forced to revert several changes, we were behind schedule but completed the minimum goals of our language.



5 Architectural Design

The LéPixcompiler system is composed of the lexer and parser, semantic analyzer and code generation subsystems. We also use the custom Abstract Syntax Tree (AST) and Semantic Abstract Syntax Tree (SEMAST) interfaces. Figure 1. shows the major subsystems (in green) with their interfaces (yellow) as well as the flow of a program through the system, upto its the compilation into LLVM IR.

5.1 Lexer (Akshaan Kakar)

We implemented the lexer in OCaml for use with the ocamllex lexer generator. The lexer accepts the program as a stream of whitespace separated tokens. Using a finite state machine generated by ocamllex, the lexer matches input characters with a defined set of permissible tokens and fails to accept in case that a prohibited symbol is seen.

5.2 Parser and Abstract Syntax Tree (Akshaan Kakar and Fatima Koli)

The parser for LéPixis also implemented in OCaml and used with the ocamlyacc LR(1) parser generator. In the parser module, we define the context free grammar for the LéPixlanguage. We structure our grammar into expressions and statements. Expressions include all the constructs that return a value, such as mathematical expressions, function calls, array accesses and assignments, variable references and assignments, and literals (integer, floating point, Boolean and Array). Statements comprise all the constructs that are used to define the sequences of expressions (control flow : if-else, for, while) and also declarations (variable declarations, function declarations).

The parser generated by ocamlyacc is an LR(1) parser that matches sequences of input tokens from the lexer with the defined grammar rules. Each of these sets of tokens that is reduced according a rule, is mapped to an instance of a type that is defined in the abstract syntax tree (Ast) interface. The AST has recursive types for expressions and statements, which encompass all the language constructs. The topmost construct in the Ast is the program, which is represented as a list of declaration statements.

5.3 Semantic Analyzer and Semantic Abstract Syntax Tree (Akshaan Kakar)

The semantic analyzer subsystem check whether the constructs expressed in the language are semantically sound. Since our language is strongly typed, the semantic analyzer makes sure that all the types in the program statements are in agreement. For instance, variable assignments are checked to ensure that the left and right sides yield the same type. Similarly function calls are checked for correct parameter types. The semantic analyzer also checks that the program follows coping rules. it enforces static coping by checking that all referenced variables and functions are defined in the regions where they are referenced.

The semantic analyzer conducts a depth first traversal of the abstract syntax tree, checking each node for type agreement and scoping rules, in a bottom-up fashion. Once the type agreement for an expression or statement is checked and its type is inferred, an instance of a new, semantically checked counterpart of the corresponding ast type in instantiated. These types comprise the Semantic Abstract Syntax Tree Semast) interface. These new types carry all the same information as the AST types but also include type information for each construct as needed. In addition, the Semast interface also defined a recursive environment type, which represents nested scope information in the program. Each scope contains a symbol table and list of defined functions along with an optional reference to its parent scope. The topmost construct of the semantic AST is the program, which is represented as a list of semantically checked variable declarations and a list of semantically checked function definitions.

5.4 Code Generation (Akshaan Kakar, Fatima Koli, Gabrielle Taylor, Jackie Lin)

The code generation (codegen) subsystem is responsible for processing the information in the semantic abstract syntax tree and generating corresponding LLVM intermediate representations, which can be converted to machine code. The codegen system initializes an LLVM builder, using the LLVM module in ocaml. A depth first traversal of the semantic AST is performed, and the LLVM instructions for each node in the tree are constructed in a top-down manner.

6 Test Plan

6.1 Representative Language Programs with Target Language Programs

Source Program (extremecontrast.lepix):

```
fun main() : int
{
      var img : int[15552] = [   ]; // truncated for length
      var w: int = 72;
      var h: int = 72;
      var size: int = w*h*3;
      var i : int;
      for(i = 0; i < size; i = i + 3)
      {
            if (img[i]>127){ img[i]=255; } else {img[i]=0;}
            if (img[i+1]>127){ img[i+1]=255; } else {img[i+1]=0;}
            if (img[i+2]>127){ img[i+2]=255; } else {img[i+2]=0;}
      }
      printppm(w);
      var j : int;
      for (j=0; j<15552; j=j+1){
            print(img[j]);
      }
      return 0;
}
```

Target Result:

```
; ModuleID = 'Lepix'

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@str1 = private unnamed_addr constant [13 x i8] c"P3\0A72 72\0
    A255\00"
@charfmt = private unnamed_addr constant [4 x i8] c"%s\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main() {
entry:
  %img = alloca [15552 x i32]
  %w = alloca i32
```

```
  %h = alloca i32
  %size = alloca i32
  %i = alloca i32
  %j = alloca i32
  store [15552 x i32] [ ], [15552 x i32]* %img ;truncated for length
  store i32 72, i32* %w
  store i32 72, i32* %h
  %w1 = load i32* %w
  %h2 = load i32* %h
  %tmp = mul i32 %w1, %h2
  %tmp3 = mul i32 %tmp, 3
  store i32 %tmp3, i32* %size
  store i32 0, i32* %i
  br label %cond

loop:                                                  ; preds = %cond
  %i4 = load i32* %i
  %tmp5 = add i32 %i4, 0
  %tmp6 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp5
  %tmp7 = load i32* %tmp6
  %tmp8 = icmp sgt i32 %tmp7, 127
  br i1 %tmp8, label %then, label %else

then:                                                  ; preds = %loop
  %i9 = load i32* %i
  %tmp10 = add i32 %i9, 0
  %tmp11 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp10
  store i32 255, i32* %tmp11
  br label %ifcont

else:                                                  ; preds = %loop
  %i12 = load i32* %i
  %tmp13 = add i32 %i12, 0
  %tmp14 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp13
  store i32 0, i32* %tmp14
  br label %ifcont

ifcont:                                                ; preds = %else, %then
  %i15 = load i32* %i
  %tmp16 = add i32 %i15, 1
```

```
  %tmp17 = add i32 %tmp16, 0
  %tmp18 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp17
  %tmp19 = load i32* %tmp18
  %tmp20 = icmp sgt i32 %tmp19, 127
  br i1 %tmp20, label %then21, label %else26

then21:                                           ; preds = %ifcont
  %i22 = load i32* %i
  %tmp23 = add i32 %i22, 1
  %tmp24 = add i32 %tmp23, 0
  %tmp25 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp24
  store i32 255, i32* %tmp25
  br label %ifcont31

else26:                                           ; preds = %ifcont
  %i27 = load i32* %i
  %tmp28 = add i32 %i27, 1
  %tmp29 = add i32 %tmp28, 0
  %tmp30 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp29
  store i32 0, i32* %tmp30
  br label %ifcont31

ifcont31:                                         ; preds = %else26, %then21
  %i32 = load i32* %i
  %tmp33 = add i32 %i32, 2
  %tmp34 = add i32 %tmp33, 0
  %tmp35 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp34
  %tmp36 = load i32* %tmp35
  %tmp37 = icmp sgt i32 %tmp36, 127
  br i1 %tmp37, label %then38, label %else43

then38:                                           ; preds = %ifcont31
  %i39 = load i32* %i
  %tmp40 = add i32 %i39, 2
  %tmp41 = add i32 %tmp40, 0
  %tmp42 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp41
  store i32 255, i32* %tmp42
  br label %ifcont48

else43:                                           ; preds = %ifcont31
```

```
  %i44 = load i32* %i
  %tmp45 = add i32 %i44, 2
  %tmp46 = add i32 %tmp45, 0
  %tmp47 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp46
  store i32 0, i32* %tmp47
  br label %ifcont48

ifcont48:                                          ; preds = %else43, %then38
  br label %inc

inc:                                               ; preds = %ifcont48
  %i49 = load i32* %i
  %tmp50 = add i32 %i49, 3
  store i32 %tmp50, i32* %i
  br label %cond

cond:                                              ; preds = %inc, %entry
  %i51 = load i32* %i
  %size52 = load i32* %size
  %tmp53 = icmp slt i32 %i51, %size52
  br i1 %tmp53, label %loop, label %afterloop

afterloop:                                         ; preds = %cond
  %uhhh = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @charfmt, i32 0, i32 0), i8* getelementptr inbounds ([13 x i8]* @str1, i32 0, i32 0))
  store i32 0, i32* %j
  br label %cond56

loop54:                                            ; preds = %cond56
  %j58 = load i32* %j
  %tmp59 = add i32 %j58, 0
  %tmp60 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp59
  %tmp61 = load i32* %tmp60
  %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmt, i32 0, i32 0), i32 %tmp61)
  br label %inc55

inc55:                                             ; preds = %loop54
```

```
  %j62 = load i32* %j
  %tmp63 = add i32 %j62, 1
  store i32 %tmp63, i32* %j
  br label %cond56

cond56:                                           ; preds = %inc55, %afterloop
  %j64 = load i32* %j
  %tmp65 = icmp slt i32 %j64, 15552
  br i1 %tmp65, label %loop54, label %afterloop57

afterloop57:                                      ; preds = %cond56
  ret i32 0
}
```

Source Program (flip.lepix):

```
fun main() : int
{
    var img : int[15552] = [  ]; // truncated for length
    var w : int = 72;
    var h : int = 72;
    var i : int = 0;
    var j : int = 213;
    var x: int = 0;
    var temp: int;
    for (x=0; x<w; x=x+1)
    {
        i=x*216;
        j=x*216;
        j=j+213;
        while (i<j){
            temp = img[j];
            img[j] = img[i];
            img[i] = temp;

            temp = img[j+1];
                img[j+1] = img[i+1];
                img[i+1] = temp;

            temp = img[j+2];
                    img[j+2] = img[i+2];
                img[i+2] = temp;

```

```
                i = i + 3;
                j = j - 3;
            }
        }

        printppm(w);
        var size: int = w*h*3;
        var l : int;
        for (l=0; l<size; l=l+1){
            print(img[l]);
        }
        return 0;
}
```

Target Result:

```
; ModuleID = 'Lepix'

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@str1 = private unnamed_addr constant [13 x i8] c"P3\0A72 72\0
    A255\00"
@charfmt = private unnamed_addr constant [4 x i8] c"%s\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main() {
entry:
  %img = alloca [15552 x i32]
  %w = alloca i32
  %h = alloca i32
  %i = alloca i32
  %j = alloca i32
  %x = alloca i32
  %temp = alloca i32
  %size = alloca i32
  %l = alloca i32
  store [15552 x i32] [ ], [15552 x i32]* %img ;truncated for
    length
  store i32 72, i32* %w
  store i32 72, i32* %h
  store i32 0, i32* %i
  store i32 213, i32* %j
  store i32 0, i32* %x
  store i32 0, i32* %x
  br label %cond
```

```

loop:                                        ; preds = %cond
  %x1 = load i32* %x
  %tmp = mul i32 %x1, 216
  store i32 %tmp, i32* %i
  %x2 = load i32* %x
  %tmp3 = mul i32 %x2, 216
  store i32 %tmp3, i32* %j
  %j4 = load i32* %j
  %tmp5 = add i32 %j4, 213
  store i32 %tmp5, i32* %j
  br label %cond8

loop6:                                       ; preds = %cond8
  %j10 = load i32* %j
  %tmp11 = add i32 %j10, 0
  %tmp12 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp11
  %tmp13 = load i32* %tmp12
  store i32 %tmp13, i32* %temp
  %j14 = load i32* %j
  %tmp15 = add i32 %j14, 0
  %tmp16 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp15
  %i17 = load i32* %i
  %tmp18 = add i32 %i17, 0
  %tmp19 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp18
  %tmp20 = load i32* %tmp19
  store i32 %tmp20, i32* %tmp16
  %i21 = load i32* %i
  %tmp22 = add i32 %i21, 0
  %tmp23 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp22
  %temp24 = load i32* %temp
  store i32 %temp24, i32* %tmp23
  %j25 = load i32* %j
  %tmp26 = add i32 %j25, 1
  %tmp27 = add i32 %tmp26, 0
  %tmp28 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp27
  %tmp29 = load i32* %tmp28
  store i32 %tmp29, i32* %temp
  %j30 = load i32* %j
  %tmp31 = add i32 %j30, 1
  %tmp32 = add i32 %tmp31, 0
  %tmp33 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp32
  %i34 = load i32* %i
```

```
  %tmp35 = add i32 %i34, 1
  %tmp36 = add i32 %tmp35, 0
  %tmp37 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp36
  %tmp38 = load i32* %tmp37
  store i32 %tmp38, i32* %tmp33
  %i39 = load i32* %i
  %tmp40 = add i32 %i39, 1
  %tmp41 = add i32 %tmp40, 0
  %tmp42 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp41
  %temp43 = load i32* %temp
  store i32 %temp43, i32* %tmp42
  %j44 = load i32* %j
  %tmp45 = add i32 %j44, 2
  %tmp46 = add i32 %tmp45, 0
  %tmp47 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp46
  %tmp48 = load i32* %tmp47
  store i32 %tmp48, i32* %temp
  %j49 = load i32* %j
  %tmp50 = add i32 %j49, 2
  %tmp51 = add i32 %tmp50, 0
  %tmp52 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp51
  %i53 = load i32* %i
  %tmp54 = add i32 %i53, 2
  %tmp55 = add i32 %tmp54, 0
  %tmp56 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp55
  %tmp57 = load i32* %tmp56
  store i32 %tmp57, i32* %tmp52
  %i58 = load i32* %i
  %tmp59 = add i32 %i58, 2
  %tmp60 = add i32 %tmp59, 0
  %tmp61 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp60
  %temp62 = load i32* %temp
  store i32 %temp62, i32* %tmp61
  %i63 = load i32* %i
  %tmp64 = add i32 %i63, 3
  store i32 %tmp64, i32* %i
  %j65 = load i32* %j
  %tmp66 = sub i32 %j65, 3
  store i32 %tmp66, i32* %j
  br label %inc7

inc7:                                               ; preds = %
    loop6
  br label %cond8

```

```
cond8:                                            ; preds = %inc7, %loop
  %i67 = load i32* %i
  %j68 = load i32* %j
  %tmp69 = icmp slt i32 %i67, %j68
  br i1 %tmp69, label %loop6, label %afterloop9

afterloop9:                                       ; preds = %cond8
  br label %inc

inc:                                              ; preds = %afterloop9
  %x70 = load i32* %x
  %tmp71 = add i32 %x70, 1
  store i32 %tmp71, i32* %x
  br label %cond

cond:                                             ; preds = %inc, %entry
  %x72 = load i32* %x
  %w73 = load i32* %w
  %tmp74 = icmp slt i32 %x72, %w73
  br i1 %tmp74, label %loop, label %afterloop

afterloop:                                        ; preds = %cond
  %uhhh = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @charfmt, i32 0, i32 0), i8* getelementptr inbounds ([13 x i8]* @str1, i32 0, i32 0))
  %w75 = load i32* %w
  %h76 = load i32* %h
  %tmp77 = mul i32 %w75, %h76
  %tmp78 = mul i32 %tmp77, 3
  store i32 %tmp78, i32* %size
  store i32 0, i32* %l
  br label %cond81

loop79:                                           ; preds = %cond81
  %l83 = load i32* %l
  %tmp84 = add i32 %l83, 0
  %tmp85 = getelementptr [15552 x i32]* %img, i32 0, i32 %tmp84
  %tmp86 = load i32* %tmp85
  %printf = call i32 (i8*, ...)* @printf(i8* getelementptr
```

```
    inbounds ([4 x i8]* @fmt, i32 0, i32 0), i32 %tmp86)
  br label %inc80

inc80:                                            ; preds = %
    loop79
  %l87 = load i32* %l
  %tmp88 = add i32 %l87, 1
  store i32 %tmp88, i32* %l
  br label %cond81

cond81:                                           ; preds = %
    inc80, %afterloop
  %l89 = load i32* %l
  %size90 = load i32* %size
  %tmp91 = icmp slt i32 %l89, %size90
  br i1 %tmp91, label %loop79, label %afterloop82

afterloop82:                                      ; preds = %
    cond81
  ret i32 0
}
```

6.2 Test Suite

6.2.1 Tests

For each new feature added to the compiler, at least one test-to-pass and one test-to-fail test program were written and added to the test suite to ensure that the feature worked correctly and that future changes to the codebase that broke these existing features would be caught. There are many small tests that test only one feature, such as arithmetic operations, unary operations, array access, array access and assign, etc. There are also larger tests that combine features, such as nested loops with array access.

fail-arr1.err

fail-arr1.lepix

fail-arr2.err

fail-arr2.lepix

fail-arr3.err

fail-arr3.lepix

fail-assign1.err

fail-assign1.lepix

fail-assign2.err

fail-assign2.lepix
fail-assign3.err
fail-assign3.lepix
fail-dead2.err
fail-dead2.lepix
fail-expr1.err
fail-expr1.lepix
fail-expr2.err
fail-expr2.lepix
fail-for1.err
fail-for1.lepix
fail-for2.err
fail-for2.lepix
fail-for3.err
fail-for3.lepix
fail-for4.err
fail-for4.lepix
fail-for5.err
fail-for5.lepix
fail-func1.err
fail-func1.lepix
fail-func2.err
fail-func2.lepix
fail-func3.err
fail-func3.lepix
fail-func5.err
fail-func5.lepix
fail-func6.err
fail-func6.lepix
fail-func7.err
fail-func7.lepix
fail-func8.err
fail-func8.lepix
fail-func9.err
fail-func9.lepix
fail-global1.err
fail-global1.lepix
fail-global2.err
fail-global2.lepix
fail-if1.err
fail-if1.lepix
fail-if2.err
fail-if2.lepix
fail-if3.err
fail-if3.lepix
fail-local1.err
fail-local1.lepix
fail-nestloop1.err
fail-nestloop1.lepix
fail-nomain.err
fail-nomain.lepix
fail-return1.err
fail-return1.lepix
fail-return2.err
fail-return2.lepix
fail-while1.err
fail-while1.lepix
fail-while2.err
fail-while2.lepix
test-2arr1.lepix
test-2arr2.lepix
test-2arr3.lepix
test-2arr4.lepix
test-add1.lepix
test-arith1.lepix
test-arith2.lepix

test-arith3.lepix
test-arr1.lepix
test-arr2.lepix
test-arr3.lepix
test-arr4.lepix
test-arr5.lepix
test-arr6.lepix
test-basic1.lepix
test-div1.lepix
test-for1.lepix
test-for3.lepix
test-for4.lepix
test-func1.lepix
test-func2.lepix
test-func3.lepix
test-func4.lepix
test-func5.lepix
test-func6.lepix
test-func7.lepix
test-func8.lepix
test-gcd.lepix
test-gcd2.lepix
test-global1.lepix
test-global2.lepix
test-global3.lepix
test-global4.lepix
test-global5.lepix
test-hello.lepix
test-helloworld.lepix
test-if1.lepix
test-if2.lepix
test-if3.lepix
test-if4.lepix
test-if5.lepix
test-if6.lepix
test-if7.lepix
test-if8.lepix
test-if9.lepix
test-if10.lepix
test-if11.lepix
test-if12.lepix
test-if13.lepix
test-if14.lepix
test-local2.lepix
test-local3.lepix
test-mod.lepix
test-nestif1.lepix
test-nestif2.lepix
test-nestloop1.lepix
test-nestloop2.lepix
test-nestloop3.lepix
test-nestloop4.lepix
test-ops1.lepix
test-ops2.lepix
test-ops3.lepix
test-ops4.lepix
test-prime.lepix
test-print.lepix
test-sqrt.lepix
test-var1.lepix
test-var2.lepix
test-while1.lepix
test-while2.lepix
test-while3.lepix
test-while4.lepix

6.2.2 Test Script

See appendix.

6.2.3 Test Automation

In order to run our test suite, we wrote a test script, `testall.sh`, which ran each test, compared its output to the expected output, and printed a pass/fail status message to the screen. If a test's output fails to match the expected output, the script prints both the output and expected output to the screen to allow for easy debugging. The script also writes information about each test to a log to further aid in debugging

6.2.4 Continuous Integration

In addition to an automated test script, we also incorporated the continuous integration tool Travis with our GitHub.

After each commit, Travis built our compiler, ran the test suite, and notified us if any commit broke the build.

This allowed us to quickly catch any mistakes immediately after they were committed and pinpoint the source of any errors.

7 Lessons Learned

7.1 Lessons : Akshaan

- One should think deeply about semantic analysis and codegen process before designing ones AST and Semantic AST interfaces
- One should write cleanly structured and modular code with expressive error messages to enable effective debugging and trouble shooting.
- One should test regularly and copiously

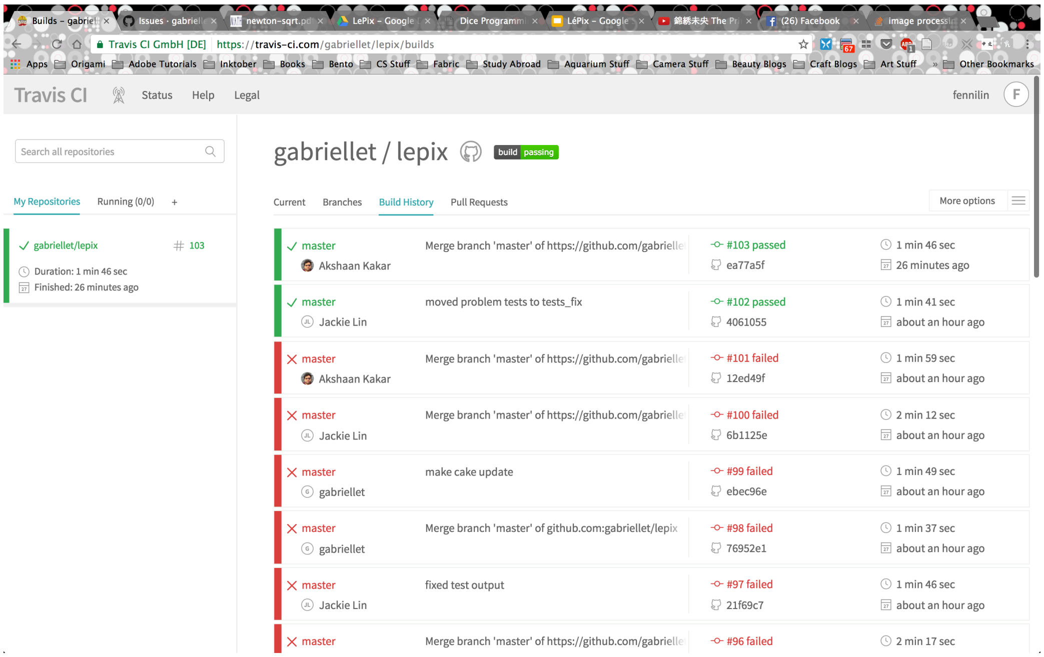


Figure 2: Travis

- One should start early
- One should communicate one's concerns/sorrows/aspirations to one's teammates clearly and regularly

7.2 Lessons : Fatima

- Communication is extremely important! Let people know if they are expanding the project too much and it doesn't seem doable in a semester. Or if you feel like you are taking on too much responsibility and someone else isn't, share that and hold the other person accountable, rather than being passive aggressive.
- You won't really be able to tell what your AST should actually look like when you create it, because at that point, you really have no idea how codegen or semantic analysis actually works. So I would say be

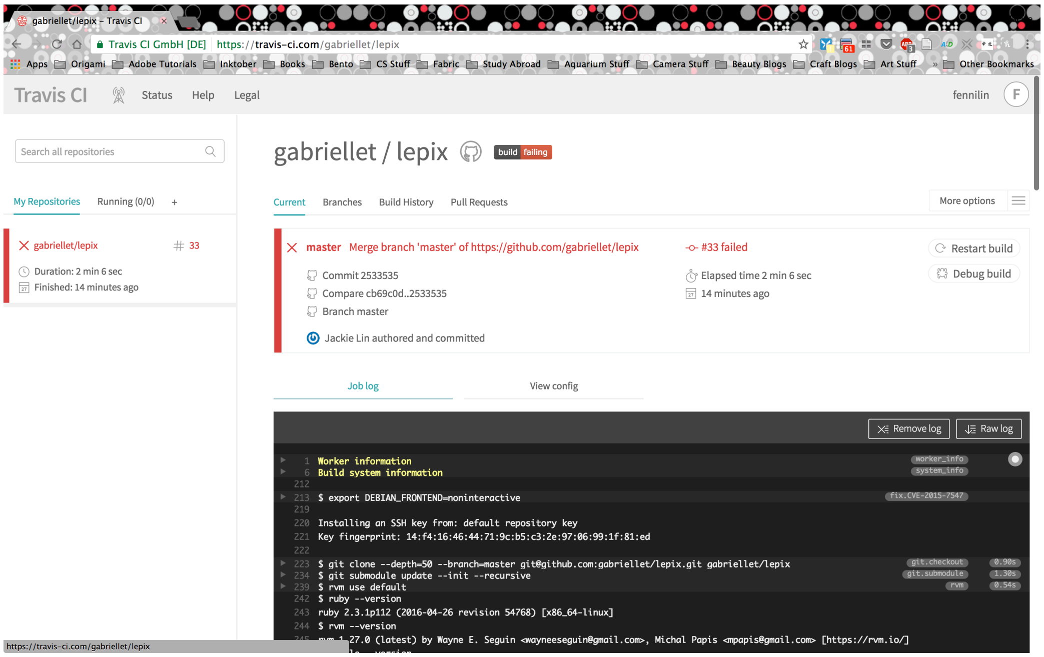


Figure 3: A broken build

flexible and willing to go back and change it completely if it makes your life easier. But figure this out sooner rather than later, so you don't end up with ugly hacks that work around the limitations of your AST.

7.3 Lessons : Gabrielle

- Start early. It will make you happier. If you look back at your Github repository and it looks like this, you've done it wrong.

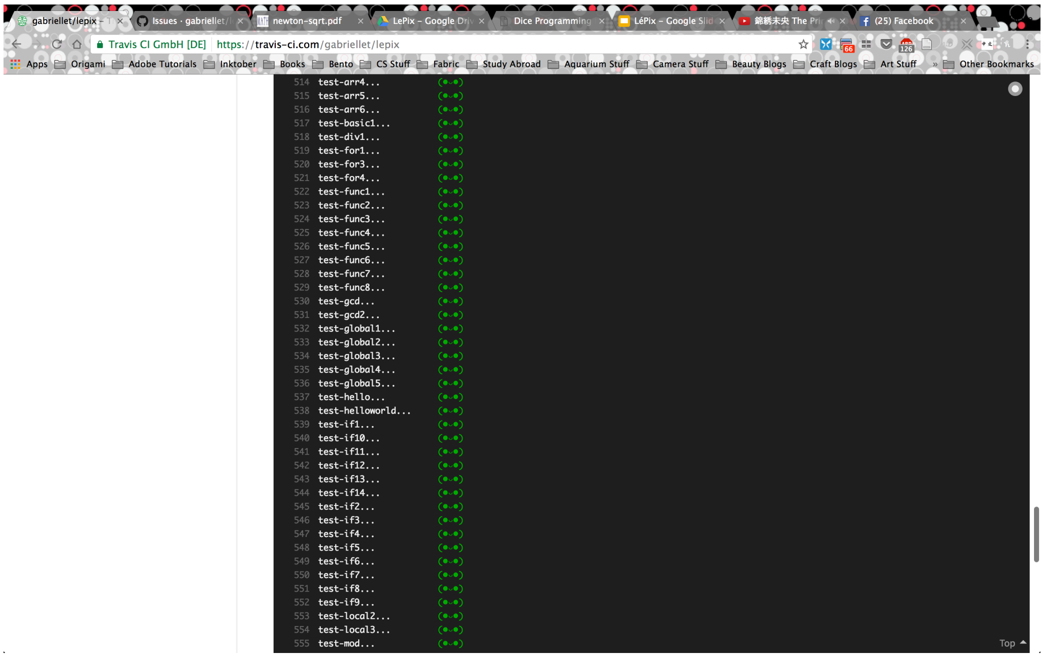


Figure 4: Happy Tests

- Choose teammates carefully . Before you decide to join someone's

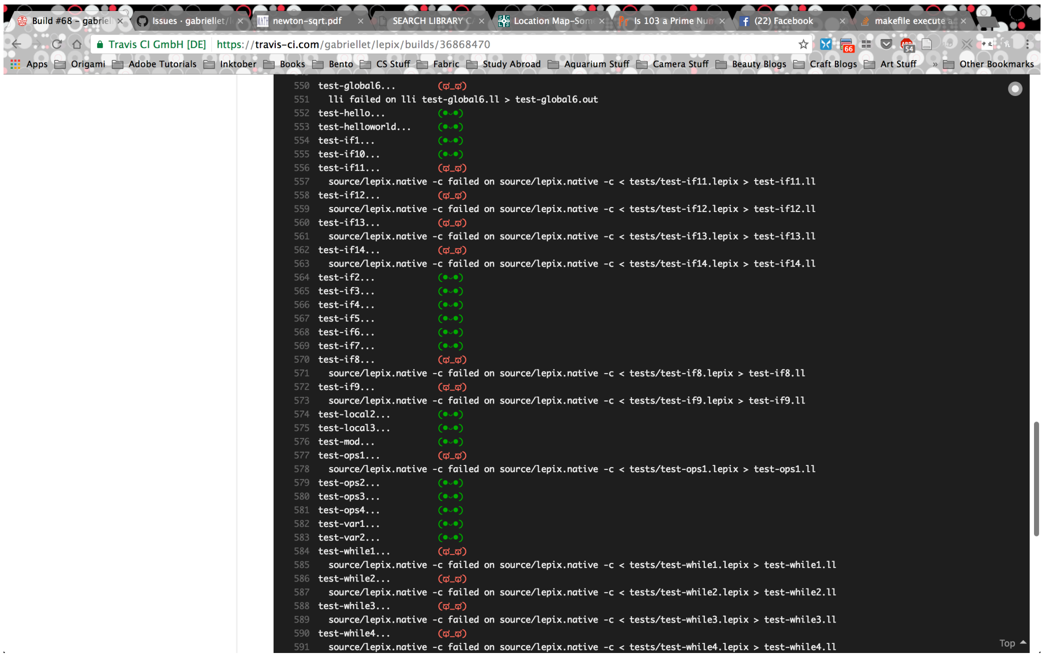


Figure 5: Not-So-Happy Tests

group, make it clear what you expect from the project.

- Take setbacks in stride. When things happen that seem like major setbacks, complaining about them won't make a difference; all you can do is move forward.

- Don't be evil. The point of a group project is group work. The point of group work is learning how to function in an actual work environment. Taking out issues you have with participating in group projects on the members of the project is irritating to everyone concerned.

- Keep it light. Even in the darkest moments, it's possible to make light of your situation. At one point we thought this language would be an "image preservation language" because we couldn't edit actual images. By the end of the project we had created this masterpiece.

7.4 Lessons : Jackie

- Start Early! You will be very unhappy otherwise.
- Test Often! Test as often as possible to catch the source of mistakes as early as possible. Integrating a continuous integration tool with your version control system will let you know which commit breaks the build so you can pinpoint the source of errors faster.

- Communicate! Make it clear to everyone what your expectations for the scope of the project are. Whether your goal is to produce something simple and that builds cleanly or to go all out and produce something new and exciting, inform your potential teammates when forming your team and if anything changes over the course of the project. Not everyone will have the same priorities, interests, or time availability; don't be afraid to be vocal about yours.
- Communicate, Part 2! Speak up if you have any issues or grievances with anyone else on your team. Politeness won't fix these problems anytime soon, and the sooner they are resolved, the happier everyone will be. Maybe. (See point below)
- Compromise! Strong personalities and conflicting goals lead to conflicts (see first point) and require compromise. The point of compromise is not to reach the solution that satisfies everyone the most, but the one that dissatisfies everyone the least. Anticipate some mild dissatisfaction in some of your team's decisions and make sure to participate in discussions if you feel uncommonly dissatisfied with anything. (And don't just rewrite the codebase without informing anyone if you are unhappy. Please.)

8 Appendix

The complete code listing for the Lepix programming language is given below:

8.1 Scanner.mll

```

{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| "/*"     { mcomment 0 lexbuf }
| "//"       { scomment lexbuf }
| '('      { LPAREN }
| ')'      { RPAREN }
```

```
| '{'        { LBRACE }
| '}'        { RBRACE }
| '['        { LSQUARE }
| ']'        { RSQUARE }
| ';'        { SEMI }
| ':'        { COLON }
| ','        { COMMA }
| '+'        { PLUS }
| '-'        { MINUS }
| '*'        { TIMES }
| '/'        { DIVIDE }
| '='        { ASSIGN }
| "=="       { EQ }
| "!="       { NEQ }
| '<'        { LT }
| "<="       { LEQ }
| ">"        { GT }
| ">="       { GEQ }
| "&&"       { AND }
| '.'        { DOT }
| "||"       { OR }
| "!"        { NOT }
| "if"       { IF }
| "else"     { ELSE }
| "for"      { FOR }
| "while"    { WHILE }
| "by"       { BY }
| "to"       { TO }
| "return"   { RETURN }
| "int"      { INT }
| "float"    { FLOAT }
| "bool"     { BOOL }
| "void"     { VOID }
| "true"     { TRUE }
| "false"    { FALSE }
| "var"      { VAR }
| "fun"      { FUN }
| "break" { BREAK }
| "continue" { CONTINUE }
| ['0'-'9']+ as lxm { INTLITERAL(int_of_string lxm) }
| '.' ['0'-'9']+ ('e' ('+'|'-')? ['0'-'9']+)? as lxm { FLOATLITERAL(float_of_string lxm) }
| ['0'-'9']+ ( '.' ['0'-'9']* ('e' ('+'|'-')? ['0'-'9']+)? | ('e' ('+'|'-')? ['0'-'9']+)) as lxm { FLOATLITERAL(float_of_string lxm) }
```

```
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }

| eof { EOF }


and mcomment level = parse
  "*/" { if level = 0 then token lexbuf else mcomment (level-1) lexbuf }
|  "/*" { mcomment (level+1) lexbuf }
| _    { mcomment level  lexbuf }

and scomment = parse
  "\n" { token lexbuf }
| _    { scomment lexbuf }
```

8.2 Parser.mly

```
%{
open Ast

let reverse_list l =
  let rec builder acc = function
    | [] -> acc
    | hd::tl -> builder (hd::acc) tl
  in
  builder [] l

%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA LSQUARE RSQUARE COLON FUN CONTINUE BREAK TO BY STRING
%token DOT QUOTE
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR VAR
%token RETURN IF ELSE FOR WHILE INT BOOL VOID FLOAT
%token <int> INTLITERAL
%token <float> FLOATLITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
```

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

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

args_list: { [] }
| expr { [$1] }
|  args_list COMMA expr {  $3::$1 }

int_list :
| INTLITERAL { [$1] }
| int_list COMMA INTLITERAL { $3::$1 }

type_name:
| INT { Int }
| FLOAT { Float }
| BOOL { Bool }
| VOID { Void }
| type_name LSQUARE int_list RSQUARE{ Array($1, $3 , 1) }
| type_name LSQUARE LSQUARE int_list RSQUARE RSQUARE { Array($1, $4, 2) }
| type_name LSQUARE LSQUARE LSQUARE int_list RSQUARE RSQUARE RSQUARE { Array($1, $5, 3) }

expr:
| INTLITERAL { IntLit($1) }
| FLOATLITERAL { FloatLit($1) }
| TRUE { BoolLit(true) }
| FALSE { BoolLit(false) }
| ID { Id($1) }
| LSQUARE args_list RSQUARE { ArrayLit(List.rev $2) }
| ID LSQUARE args_list RSQUARE { Access($1,List.rev $3)  }
| ID LPAREN args_list RPAREN  { Call($1,List.rev $3)  }
| MINUS expr %prec NEG { Unop( Neg, $2) }
| NOT expr { Unop( Not, $2) }
| expr TIMES expr { Binop( $1, Mult, $3) }
| expr DIVIDE expr { Binop( $1, Div, $3)  }
| expr PLUS expr { Binop( $1, Add, $3) }
```

```
| expr MINUS expr { Binop( $1, Sub, $3) }
| expr LT expr { Binop( $1, Less, $3) }
| expr GT expr { Binop( $1, Greater, $3) }
| expr LEQ expr { Binop( $1, Leq, $3) }
| expr GEQ expr { Binop( $1, Geq, $3) }
| expr NEQ expr { Binop( $1, Neq, $3) }
| expr EQ expr { Binop( $1, Equal, $3) }
| expr AND expr { Binop( $1, And, $3) }
| expr OR expr { Binop( $1, Or, $3) }
| ID ASSIGN expr { Assign($1,$3) }
| ID LSQUARE args_list RSQUARE ASSIGN expr { ArrayAssign($1,List
    .rev $3,$6) }

params_list: { [] }
| ID COLON type_name { [($1,$3)] }
| ID COLON type_name COMMA params_list { ($1,$3)::$5 }

var_decl:
  VAR ID COLON type_name ASSIGN expr SEMI { VarDecl(($2,$4),$6)
    }
| VAR ID COLON type_name SEMI { VarDecl(($2,$4),Noexpr) }

fun_decl:
FUN ID LPAREN params_list RPAREN COLON type_name LBRACE
    statement_list RBRACE { { func_name=$2; func_parameters= $4;
    func_return_type=$7; func_body=$9} }

statement_list_builder: { [] }
| statement_list_builder statement { $2::$1 }

statement_list :
| statement_list_builder { reverse_list $1 }

statement:
| expr SEMI { Expr($1) }
| IF LPAREN expr RPAREN LBRACE statement_list RBRACE %prec
    NOELSE { If($3,Block($6),Block([])) }
| IF LPAREN expr RPAREN LBRACE statement_list RBRACE ELSE LBRACE
    statement_list RBRACE { If($3,Block($6),Block($10))  }
| WHILE LPAREN expr RPAREN LBRACE statement_list RBRACE { While(
    $3,Block($6)) }
|  FOR LPAREN expr TO expr BY expr RPAREN LBRACE statement_list
    RBRACE { For($3,$5,$7,Block($10)) }
|  FOR LPAREN expr SEMI expr SEMI expr RPAREN LBRACE
    statement_list RBRACE { For($3,$5,$7,Block($10)) }
```

```
| RETURN expr SEMI  { Return($2) }
| BREAK SEMI { Break }
| CONTINUE SEMI { Continue }
| var_decl { VarDecStmt($1) }


decls_list : { [] }
| decls_list fun_decl { Func($2)::$1 }
| decls_list var_decl { Var($2)::$1 }

program:
| decls_list EOF { reverse_list $1 }
```

8.3 Ast.ml

```

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

type uop = Neg | Not

type typ =
      | Int
      | Bool
      | Void
      | Float
      | Array of typ * int list * int

type bind = string * typ

type expr =
      | BoolLit of bool
      | IntLit of int
      | FloatLit of float
      | Id of string
      | Call of string * expr list
      | Access of string * expr list
      | Binop of expr * op * expr
      | Unop of uop * expr
      | Assign of string * expr
      | ArrayAssign of string * expr list * expr
      | InitArray of string * expr list
      | ArrayLit of expr list
```

```
    | Noexpr

type var_decl =
    | VarDecl of bind * expr

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

type func_decl = {
    func_name : string;
    func_parameters : bind list;
    func_return_type : typ;
    func_body : stmt list;
}

type decl =
    | Func of func_decl
    | Var of var_decl

type prog = decl list

let string_of_op = function
    | Add -> "+"
    | Sub -> "-"
    | Mult -> "*"
    | Div -> "/"
    | Equal -> "=="
    | Neq -> "!="
    | Less -> "<"
    | Leq -> "<="
    | Greater -> ">"
    | Geq -> ">="
    | And -> "&&"
    | Or -> "||"

let rec string_of_list = function
    | [] -> ""
```

```
    | s::l -> s ^ "," ^ string_of_list l

let string_of_uop = function
    | Neg -> "-"
    | Not -> "!"

let rec string_of_expr = function
     IntLit(l) -> string_of_int l
    | BoolLit(true) -> "true"
    | BoolLit(false) -> "false"
    | FloatLit(f) -> string_of_float f
    | Id(sl) -> sl
    | Binop(e1, o, e2) ->
        string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
    | Unop(o, e) -> string_of_uop o ^ string_of_expr e
    | Access(e, l) -> e ^ "[" ^ string_of_list (List.map string_of_expr l) ^ "]"
    | ArrayAssign (s, l, e) -> s ^"[" ^ string_of_expr_list l ^ "] = " ^ string_of_expr e
    | Assign(v, e) -> v ^ " = " ^ string_of_expr e
    | InitArray(s, el) -> s ^ " = [" ^ String.concat ", " (List.map string_of_expr el) ^ "]"
    | Call(e, el) ->
        e ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
    | Noexpr -> "{ Noop }"
    | ArrayLit(el) -> "[ " ^ String.concat ", " (List.map string_of_expr el) ^ " ]"

and string_of_expr_list = function
    | [] -> ""
    | s::l -> string_of_expr s ^ "," ^ string_of_expr_list l

let rec string_of_typ = function
    | Int -> "int"
    | Bool -> "bool"
    | Void -> "void"
    | Float -> "float"
    | Array(t, il, d) -> string_of_typ t ^ ( String.make d '[' ) ^ ( String.make d ']' )

let rec string_of_bind = function
    | (str, typ) -> str ^ " : " ^ string_of_typ typ
```

```
let rec string_of_bind_list = function
      | [] -> ""
      | hd::[] -> string_of_bind hd
      | hd::tl -> string_of_bind hd ^ string_of_bind_list tl

let rec string_of_var_decl = function
      | VarDecl(binding,expr) -> "var " ^ string_of_bind binding ^ " = " ^ string_of_expr expr ^ ";\n"

let rec string_of_stmt_list = function
      | [] -> ""
      | hd::[] -> string_of_stmt hd
      | hd::tl -> string_of_stmt hd ^ ";\n" ^ string_of_stmt_list tl ^ "\n"
      and string_of_stmt = function
            | Block(sl) -> string_of_stmt_list sl
            | Expr(expr) -> string_of_expr expr ^ ";\n";
            | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
            | If(e, s, s2) -> "if (" ^ string_of_expr e ^ ")\n" ^"{" ^  string_of_stmt s ^ "}\n" ^ "else\n{" ^ string_of_stmt s2 ^"\n}"
            | For(e1, e2, e3, s) -> "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^ string_of_expr e3  ^ ")\n{ " ^ string_of_stmt s ^ "}"
            | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
            | Break -> "break;\n"
            | Continue -> "continue;\n"
            | VarDecStmt(vdecl) -> string_of_var_decl vdecl
(*          | Parallel(el,sl) -> "parallel( invocations = " ^ string_of_expr_list el ^ " )\n{\n" ^ string_of_stmt_list sl ^ "\n}\n"
            | Atomic(sl) -> "atomic {\n" ^ string_of_stmt_list sl ^ "}\n"
*)
let string_of_func_decl fdecl =
      "fun " ^  fdecl.func_name
     ^ "(" ^ string_of_bind_list fdecl.func_parameters ^ ") :"
     ^ string_of_typ fdecl.func_return_type  ^ "{\n"
     ^ string_of_stmt_list fdecl.func_body
     ^ "}"

let string_of_decl = function
      | Func(fdecl) -> string_of_func_decl fdecl
```

```
    | Var(vdecl) -> string_of_var_decl vdecl

let rec string_of_decls_list = function
    | [] -> ""
    | hd::[] -> string_of_decl hd
    | hd::tl -> string_of_decl hd ^ string_of_decls_list tl

let string_of_program p =
    string_of_decls_list p
```

8.4 Semant.ml

```

open Ast
open Semast

exception SemanticException of string

let rec check_dup l = match l with [] -> false
                                 | hd::tl -> let x = (List.filter (fun x -> x = hd) tl ) in
                                 if (x == []) then
                                      check_dup tl
                                 else
                                      true
let rec list_if_uniq l = if (check_dup l) then raise(SemanticException("Duplicate arg names in func")) else l


let rec find_variable scope name =
    try
        List.find (fun (_,s) -> s = name) scope.vars
    with Not_found ->
    (
        match scope.parent_scope
    with Some(parent) ->
        find_variable parent name
    | _ -> raise (SemanticException ("Undefined ID " ^ name))
    )

let rec list_compare l1 l2 =
    match (l1,l2) with ([],[]) -> true
    | ((Array(_,_,_),_)::tl1 , (Array(_,_,_),_)::tl2) -> true
    | (hd1::tl1 , hd2::tl2) -> if hd1 = hd2 then list_compare
```

```
    tl1 tl2 else false
      | _ -> false

let rec list_compare_typ l1 l2 =
      match (l1,l2) with ([],[]) -> true
      | (hd1::tl1 , hd2::tl2) -> if hd1 = hd2 then
    list_compare_typ tl1 tl2 else false
          | _ -> false

let get_expr_type sexpr =
      match sexpr with S_IntLit(i) -> Int
      | S_BoolLit(b) -> Bool
      | S_FloatLit(f) -> Float
      | S_Id(s,typ) -> typ
      | S_Call(s,el,typ) -> typ
      | S_Access(s,el,typ,dims) -> typ
      | S_Binop(l,op,r,typ) -> typ
      | S_Unop(op,e,typ) -> typ
      | S_Assign(s,e,typ) -> typ
      | S_ArrayAssign(s,el,e,typ,atyp) -> typ
      | S_ArrayLit(el,typ) -> typ
      | S_InitArray(s,el,typ) -> typ
      | S_Noexpr -> Void

let rec check_expr e env =
          match e with
          IntLit(i) -> S_IntLit(i)
          | FloatLit(f) -> S_FloatLit(f)
          | BoolLit(b) -> S_BoolLit(b)
          | Id(x) -> (let (typ,var) = find_variable env.scope x in
     S_Id(var,typ))
          | Binop(l,op,r) -> check_binop l op r env
          | Unop(op,l) -> check_unop op l env
          | Call(s,el) -> check_call s el env
          | Access(s,el) -> check_access s el env
          | Assign(s,e) -> check_assign s e env
          | ArrayAssign(s,ind,exp) -> check_array_assign s ind exp
     env
          | InitArray(s,el) -> check_init_array s el env
          | ArrayLit(el) -> check_array_lit el env
          | Noexpr -> S_Noexpr


and check_binop l op r env =
      let sexpr_l = check_expr l env and
```

```
    sexpr_r = check_expr r env in
    let ltyp = get_expr_type sexpr_l and
    rtyp = get_expr_type sexpr_r in
    if ltyp = rtyp then
        match op with Add -> S_Binop(sexpr_l,op,sexpr_r,ltyp)
                    | Sub -> S_Binop(sexpr_l,op,sexpr_r,ltyp)
                    | Mult -> S_Binop(sexpr_l,op,sexpr_r,ltyp)
                    | Div -> S_Binop(sexpr_l,op,sexpr_r,ltyp)
                    | _ -> S_Binop(sexpr_l,op,sexpr_r,Bool)
    else  raise (SemanticException("Incompatible types"))
and check_unop op e env =
    let sexp = check_expr e env in
    let sexp_typ = get_expr_type sexp in
    match sexp_typ with
    Int -> (match op with Neg -> S_Unop(op,sexp,sexp_typ) | _ -> raise(SemanticException("Invalid operator")))
     | Float -> (match op with Neg -> S_Unop(op,sexp,sexp_typ) | _ -> raise(SemanticException("Invalid operator")))
     | Bool -> (match op with Not -> S_Unop(op,sexp,sexp_typ) | _ -> raise(SemanticException("Invalid operator")))
     | _ -> raise(SemanticException("Unary op on invalid type"))
and check_assign l r env =
    let (ltype,vname) = find_variable env.scope l
    and sexpr_r = check_expr r env in
    let rtype = get_expr_type sexpr_r in
    if ltype = rtype then S_Assign(vname,sexpr_r,ltype) else raise(SemanticException("Incompatible types in assignment"))
and check_expr_list el typ env =
    match el with  [] -> raise(SemanticException("Invalid array access"))
     | hd::[] -> let sexpr = check_expr hd env in if get_expr_type sexpr <> typ
         then raise(SemanticException("Invalid array access"))
         else sexpr::[]
     | hd::tl -> let sexpr = check_expr hd env in if get_expr_type sexpr <> typ
         then raise(SemanticException("Invalid array access"))
         else sexpr::check_expr_list tl typ env
and check_access s el env =
    let (typ,name) = find_variable env.scope s and
    sexpr_list = check_expr_list el Int env in
    match typ with Ast.Array(t,il,d) -> S_Access(s,sexpr_list,t,typ)
     | _ -> raise(SemanticException("Attempting array access in non-array"))
```

```
and create_sexpr_list el env =
    match el with  [] -> []
    | hd::tl -> (check_expr hd env)::(create_sexpr_list tl env)

and find_function env fname el =
        let sexpr_list_args = create_sexpr_list el env in
        let args_types_call = List.map get_expr_type sexpr_list_args in
        try
                let found = List.find ( fun f -> f.func_name = fname ) env.funcs in
                let formals_types = List.map fst found.func_parameters in
                if List.length args_types_call = List.length formals_types
                then (if list_compare_typ args_types_call formals_types
                then found
                else raise(SemanticException("Incompatible args to func")))
                else raise(SemanticException("Wrong num of args to func"))
        with   Not_found -> raise(SemanticException("Undefined func called"))
and check_call s el env =
    let sfunc = find_function env s el in
    S_Call(s,create_sexpr_list el env,sfunc.func_return_type)
and check_array_assign s el e env =
    let (atype,var) = find_variable env.scope s in
    let sexpr_index = check_expr_list el Int env and
    sexpr_assign = check_expr e env    in
    let assgn_type = get_expr_type sexpr_assign in
    let arr_prim_type = match atype with Array(t,il,d) -> t | _ -> raise(SemanticException("Accessing non array")) in
    if assgn_type = arr_prim_type then S_ArrayAssign(s,sexpr_index,sexpr_assign,assgn_type,atype)
    else raise(SemanticException("Invalid type in array assign"))
and check_init_array s el env =
    let (atype,name) = find_variable env.scope s in
    let sexpr_assgn_list = check_expr_list el atype env in
        S_InitArray(s,sexpr_assgn_list,atype)

and check_array_lit el env =
    let sexpr_list = create_sexpr_list el env  in
```

```
    let type_list = List.map get_expr_type sexpr_list in
    match type_list with [] -> raise(SemanticException("Empty array lit"))
    | hd::_ -> S_ArrayLit(check_expr_list el hd env, hd)

let rec check_stmt st env =
    match st with Expr(e) -> let sexpr = check_expr e env in let sexpr_typ = get_expr_type sexpr in S_Expr(sexpr,sexpr_typ)
    | Return(e) -> check_return e env
    | Block(sl) -> let new_scope = { parent_scope = Some(env.scope); vars = []; } in
                let new_env = { env with scope = new_scope} in
                let stmt_list = List.map (fun s -> check_stmt s new_env) sl in
                new_scope.vars <- List.rev new_scope.vars;
                S_Block(stmt_list)
    | If(e,sl1,sl2) -> check_if e sl1 sl2 env
    | For(e1,e2,e3,sl) -> check_for e1 e2 e3 sl env
    | While(e,sl) -> check_while e sl env
    | Break -> S_Break
    | Continue -> S_Continue
    | VarDecStmt(VarDecl((name,typ),e)) -> check_var_decl name typ e env


and check_return e env =
    if not env.in_function_body then raise(SemanticException("Return used outside function body"))
    else
    let sexpr = check_expr e env in
    let ret_typ = get_expr_type sexpr in
    if ret_typ = env.return_type then S_Return(sexpr,ret_typ)
    else raise(SemanticException("Incorrect return type"))

and check_if e sl1 sl2 env =
    let sexpr_cond = check_expr e env in
    let cond_typ = get_expr_type sexpr_cond
    and sstmt1 = check_stmt sl1 env
    and sstmt2 = check_stmt sl2 env in
    if cond_typ = Bool then S_If(sexpr_cond,sstmt1,sstmt2)
    else raise(SemanticException("If condition does not evaluate to bool"))

and check_for e1 e2 e3 sl env =
```

```
      let sexpr1 = check_expr e1 env
      in let t1 = get_expr_type sexpr1
      in let sexpr2 = check_expr e2 env
      in let t2 = get_expr_type sexpr2
      in let sexpr3 = check_expr e3 env
      in let t3 = get_expr_type sexpr3
      in if t1 <> Int && t1 <> Void then
           raise( SemanticException("For loop first expr of invalid type"))
      else (if t2 <> Bool then
           raise (SemanticException("For loop second expr not of type bool"))
      else (if t3 <> Int then
           raise (SemanticException("For loop third expr not of type int"))
      else (let s = check_stmt sl env in S_For(sexpr1,sexpr2,sexpr3,s))))

and check_while e sl env =
      let sexpr = check_expr e env
      in let sexpr_typ = get_expr_type sexpr
      in let s = check_stmt sl env in
      if sexpr_typ <> Bool then raise(SemanticException("While condition has invalid type"))
      else S_While(sexpr,s)

and check_array_var_decl name t il d e etype env =
      if etype = t then
           if d = List.length then
                S_VarDecStmt(S_VarDecl((name,t),e))
           else raise(SemanticException("Array literal size is incorrect"))
      else raise(SemanticException("Array literal has wrong type in assignment"))

and check_var_decl name typ e env =
        let sexpr = check_expr e env in
        let sexpr_typ = get_expr_type sexpr in
        if List.exists (fun (_,vname) -> vname = name) env.scope.vars
        then raise(SemanticException("Variable has already been declared"))
        else
          match typ with Array(t,il,d) -> env.scope.vars <- (typ,name)::env.scope.vars;
```

```
                                        S_VarDecStmt(S_VarDecl((name,typ),sexpr))
                            | _ ->
                                if sexpr_typ <> typ && sexpr_typ <> Void
                                        then raise(SemanticException("Invalid type assigned in declaration"))
                                else
                                        if typ = Void
                                        then raise(SemanticException("Cannot have var of type void"))
                                        else env.scope.vars <- (typ,name)::env.scope.vars;
                                                            S_VarDecStmt(S_VarDecl((name,typ),sexpr))

let check_func_decl (fdecl : Ast.func_decl) env =
    if env.in_function_body then
        raise (SemanticException("Nested function declaration"))
    else
        let f_env = { env with scope = {parent_scope = Some(env.scope);
                                        vars = List.map (fun (name,typ) -> (typ,name)) fdecl.func_parameters;};
                                        return_type = fdecl.func_return_type; in_function_body = true}
        in
        if (fdecl.func_return_type = Void ||
            List.exists (fun x -> match x with Return(e) -> true | _ -> false) fdecl.func_body)
        then let sfbody = List.map (fun s -> check_stmt s f_env) fdecl.func_body in
        let sfdecl = {Semast.func_name = fdecl.func_name;
                                        Semast.func_return_type = fdecl.func_return_type;
                                        Semast.func_parameters = List.map (fun (a,b) -> match b with
                                                            Void -> raise(SemanticException("Void type for func arg"))
                                                            |_ -> (b,a)) (list_if_uniq fdecl.func_parameters);
                                        Semast.func_body = sfbody;
                                        Semast.func_locals = List.map (fun x ->
```

```
                                                            match x with
                                                            S_VarDecStmt(S_VarDecl((name,typ),sexpr)) ->
                                                                (typ,name,sexpr)
                                                            | _ -> raise(SemanticException("Sacré bleu! You're in trouble because this shouldn't happen"))
                                                            )
                                                                (List.filter (fun decl ->
                                                                match decl with
                                                            S_VarDecStmt(S_VarDecl(t,sexpr)) ->
                                                             true
                                                            | _ -> false
                                                            ) sfbody);}
                in (
                if List.exists (fun f -> sfdecl.func_name = f.func_name
                                && list_compare sfdecl.func_parameters f.func_parameters) env.funcs
                then raise(SemanticException("Redefining function " ^ fdecl.func_name))
                else env.funcs <- sfdecl::env.funcs; sfdecl
                )
            else raise(SemanticException("No return stmt in func def" ^ fdecl.func_name))

let create_environment =
        let new_funcs = [{ Semast.func_return_type = Void;
                    Semast.func_name = "print";
                    Semast.func_parameters = [(Int,"a")];
                    Semast.func_body = [];
                    Semast.func_locals = [];
                };
                 { Semast.func_return_type = Void;
                    Semast.func_name = "printb";
                    Semast.func_parameters = [(Bool,"a")];
                    Semast.func_body = [];
                    Semast.func_locals = [];
                };
                 { Semast.func_return_type = Void;
                    Semast.func_name = "printf";
```

```
                    Semast.func_parameters = [(Float,"a")];
                    Semast.func_body = [];
                    Semast.func_locals = [];
                  };
                { Semast.func_return_type = Void;
                  Semast.func_name = "printppm";
                  Semast.func_parameters = [(Int,"a")];
                  Semast.func_body = [];
                  Semast.func_locals = [];
                };
                ]

      in
        let new_scope = { parent_scope = None; vars = [];} in
        {
                Semast.funcs = new_funcs;
                scope = new_scope;
                return_type = Void;
                in_function_body = false;
        }

let check_decl env prog =
                let vars = List.filter (fun decl ->  match decl
    with Var(vdecl) -> true | _ -> false) prog
          and funs = List.filter (fun decl -> match decl with
    Func(decl) -> true | _ -> false) prog
          in
          let globs = List.map (fun x -> match x with Var(vdecl)
     -> check_stmt (VarDecStmt(vdecl)) env
                                                 | _ -> raise(
    SemanticException("Func in vardecls list")) ) vars
          and fdcls = List.map (fun x -> match x with Func(fdecl
    ) -> check_func_decl fdecl env
                                              | _ -> raise(
    SemanticException("Var in funcdecls list")) ) funs
          in
          { Semast.globals = List.map (fun x -> match x with
    S_VarDecStmt(S_VarDecl((s,t),e)) -> (t,s,e)
                                              | _ -> raise(
    SemanticException("Var in funcdecls list")) ) globs;
            Semast.functions = fdcls
          }

let check_prog prog =
        let env = create_environment in
```

```
    let sprog = check_decl env prog
    in
    if List.exists (fun f -> f.func_name = "main" && f.
  func_return_type = Int) env.funcs
    then sprog
    else raise(SemanticException("Main function not defined"))
```

8.5 Semast.ml

```
open Ast

type s_expr =
|     S_IntLit of int
|     S_BoolLit of bool
|     S_FloatLit of float
|     S_Id of string * typ
|     S_Call of string * s_expr list * typ
|     S_Access of string * s_expr list * typ * typ
|     S_Binop of s_expr * op * s_expr * typ
|     S_Unop of uop * s_expr * typ
|     S_Assign of string * s_expr * typ
|     S_ArrayAssign of string * s_expr list * s_expr * typ * typ
|     S_ArrayLit of s_expr list * typ
|     S_InitArray of string * s_expr list * typ
|     S_Noexpr

type s_var_decl
      = S_VarDecl of bind * s_expr

type s_stmt =
|     S_Expr of s_expr * typ
|     S_Return of s_expr * typ
|     S_If of s_expr * s_stmt * s_stmt
|     S_For of s_expr * s_expr * s_expr * s_stmt
|     S_While of s_expr * s_stmt
|     S_Break
|     S_Continue
|     S_VarDecStmt of s_var_decl
|     S_Block of s_stmt list

type s_func_decl = {
      func_name : string;
      func_parameters : (typ * string) list;
      func_return_type : typ;
```

```
    func_body : s_stmt list;
    func_locals : (typ * string * s_expr)  list;
}

type s_decl =
|   S_Func of s_func_decl
|   S_Var of s_var_decl


type s_program = {
    globals : (Ast.typ*string*s_expr) list;
    functions : s_func_decl list;
}


type symbolTable = {
    parent_scope: symbolTable option;
    mutable vars: (typ * string) list;
}

type env = {
    mutable funcs: s_func_decl list;
    scope: symbolTable;
    return_type : typ;
    in_function_body : bool;
}
```

8.6 Codegen.ml

```

module L = Llvm
module A = Ast
module S = Semast
module StringMap = Map.Make(String)

exception CodegenError of string

let generate (sprog) =
  let context = L.global_context () in
  let _le_module = L.create_module context "Lepix"
  and f32_t   = L.float_type   context
  and f64_t   = L.double_type  context
  and i8_t    = L.i8_type      context
  and i32_t   = L.i32_type     context
```

```
and bool_t  = L.i1_type      context
and void_t  = L.void_type    context in

let compute_array_index d il = match d with 1 -> (List.nth il 0)
                                          | 2 -> (List.nth il 0) * (List.nth il 1)
                                          | 3-> (List.nth il 0) * (List.nth il 1) * (List.nth il 2)
                                          | _ -> raise(CodegenError("Too many dimensions"))
in
let rec ast_to_llvm_type = function
  | A.Bool -> bool_t
  | A.Int -> i32_t
  | A.Float -> f32_t
  | A.Void -> void_t
  | A.Array(t, il, d) -> L.array_type (ast_to_llvm_type t) (compute_array_index d il)
in
let global_vars =
  let global_var map (typ,name) =
    let init = L.const_int (ast_to_llvm_type typ) 0
    in StringMap.add name (L.define_global name init _le_module) map in
  let globals_list = List.map (fun (typ,s,e) -> (typ,s)) sprog.S.globals in
  List.fold_left global_var StringMap.empty globals_list
in
let print_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let print_func = L.declare_function "printf" print_t _le_module in

let function_decls =
  let function_decl map fdecl =
    let param_types = Array.of_list (List.map (fun (t,s) -> ast_to_llvm_type t) fdecl.S.func_parameters)
    in let ftype = L.function_type (ast_to_llvm_type fdecl.S.func_return_type) param_types
    in StringMap.add fdecl.S.func_name (L.define_function fdecl.S.func_name ftype _le_module,fdecl) map
  in List.fold_left function_decl StringMap.empty sprog.S.functions
in
```

```
let function_body fdecl =
  let (func,_) = StringMap.find fdecl.S.func_name function_decls
  in let builder = L.builder_at_end context (L.entry_block func) in

  let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder in
  let float_format_str = L.build_global_stringptr "%.2f\n" "floatfmt" builder in
  let char_format_str = L.build_global_stringptr "%s\n" "charfmt" builder in
  let header = L.build_global_stringptr "P3\n72 72\n255" "str1" builder in
  let local_vars =
    let add_formals map (name,typ) p = L.set_value_name name p;
      let local = L.build_alloca (ast_to_llvm_type typ) name builder in
      ignore (L.build_store p local builder);
      StringMap.add name local map in

    let rec add_local map (name,typ,e) = let local_var = L.build_alloca (ast_to_llvm_type typ) name builder in
      StringMap.add name local_var map
    in
    let params_list = List.map (fun (s,t) -> (t,s)) fdecl.S.func_parameters
    in
    let formals = List.fold_left2 add_formals StringMap.empty params_list (Array.to_list (L.params func))
    in
    let locals_list = List.map (fun (s,t,e) -> (t,s,e)) fdecl.S.func_locals in
    List.fold_left add_local formals locals_list


  in let lookup name = try StringMap.find name local_vars with
   Not_found -> StringMap.find name global_vars
  in let rec gen_expression sexpr builder =
       match sexpr with
         S.S_Id(s,typ) -> L.build_load (lookup s) s builder
       | S.S_BoolLit(value) -> L.const_int bool_t (if value then 1 else 0)
       | S.S_IntLit(value) -> L.const_int i32_t value
```

```
        | S.S_FloatLit(value) -> L.const_float f32_t value
        | S.S_Call("print", [e], typ) -> L.build_call
    print_func [| int_format_str ; (gen_expression e builder) |]
    "printf" builder
        | S.S_Call("printb",[e], typ) -> L.build_call
    print_func [| int_format_str ; (gen_expression e builder) |]
    "printf" builder
        | S.S_Call("printf",[e],typ) -> let gen= gen_expression
     e builder in
          let double = L.build_fpext gen f64_t "dou" builder in
          L.build_call print_func [| (float_format_str) ;
                                    double |] "printf" builder
        | S.S_Call("printppm", [e], typ) -> L.build_call
    print_func [| (char_format_str);

          (header) |] "printheader" builder;
        | S.S_Call(e, el,typ) -> let (fcode,fdecl) = StringMap.
    find e function_decls in
          let actuals = List.rev (List.map (fun s ->
    gen_expression s builder) (List.rev el) )in
          let result = (match fdecl.S.func_return_type with A.
    Void -> ""
                                                          | _
    -> e ^ "_result")
          in L.build_call fcode (Array.of_list actuals) result
    builder
        | S.S_ArrayLit(el,typ) -> L.const_array (
    ast_to_llvm_type typ) (Array.of_list (List.map (fun x ->
            gen_expression x builder) el))

        | S.S_Access(s, el,typ,A.Array(t,il,d)) -> (match d
    with 1 ->  let index = gen_expression (List.hd el) builder in
            let index = L.build_add index (L.const_int i32_t 0)
     "tmp" builder in
            let value = L.build_gep (lookup s)
                [| (L.const_int i32_t 0); index; |] "tmp"
    builder
            in L.build_load value "tmp" builder


    | 2 -> let indexlist = List.map (fun x -> gen_expression x
    builder) el in

      let index = L.build_add (L.const_int i32_t 0)

```

```
        (List.nth indexlist 1) "tmp" builder in
     let rows = L.build_mul (List.nth indexlist 0)  (L.const_int
    i32_t
                                                        (List.nth
    il 1)) "tmp2" builder
     in let index = L.build_add index rows "tmp" builder in
     let value = L.build_gep (lookup s)
        [| (L.const_int i32_t 0); index |] "tmp" builder
     in L.build_load value "tmp" builder

   | _ -> raise(CodegenError("Invalid dim number"))
          )
      | S.S_Binop(e1, op, e2,A.Float) ->
          let left = gen_expression e1 builder
          and right = gen_expression e2 builder in
          (
            match op with A.Add -> L.build_fadd
                          |             A.Sub -> L.build_fsub
                          |             A.Mult -> L.build_fmul
                          |             A.Div -> L.build_fdiv
                          |             A.Equal -> L.build_fcmp L
   .Fcmp.Ueq
                          |             A.Neq -> L.build_fcmp L.
   Fcmp.Une
                          |             A.Less -> L.build_fcmp L.
   Fcmp.Ult
                          |         A.Leq -> L.build_fcmp L.Fcmp.
   Ule
                          |         A.Greater -> L.build_fcmp L.
   Fcmp.Ugt
                          |             A.Geq -> L.build_fcmp L.
   Fcmp.Uge
                          | _ -> raise(CodegenError("Invalid
   operator for floats"))
          ) left right "tmp" builder
      | S.S_Binop(e1, op, e2,typ) ->
          let left = gen_expression e1 builder
          and right = gen_expression e2 builder in
```

```
          (
            match op with A.Add -> L.build_add
                        |         A.Sub -> L.build_sub
                        |         A.Mult -> L.build_mul
                        |         A.Div -> L.build_sdiv
                        |         A.And -> L.build_and
                        |         A.Or -> L.build_or
                        |         A.Equal -> L.build_icmp L.Icmp.Eq
                        |         A.Neq -> L.build_icmp L.Icmp.Ne
                        |         A.Less -> L.build_icmp L.Icmp.Slt
                        |      A.Leq -> L.build_icmp L.Icmp.Sle
                        |      A.Greater -> L.build_icmp L.Icmp.Sgt
                        |         A.Geq -> L.build_icmp L.Icmp.Sge
          ) left right "tmp" builder
        | S.S_Unop(op, e1, typ) ->
          let exp = gen_expression  e1 builder in
          (
            match op with A.Neg -> L.build_neg
                        | A.Not -> L.build_not
          ) exp "tmp" builder
        | S.S_Assign(s, e, typ) -> let e' = gen_expression e builder in ignore(L.build_store e' (lookup s) builder); e'

        | S.S_ArrayAssign(s, el, e2, typ,A.Array(t,il,d)) -> (match d with 1 ->  let index = gen_expression (List.hd el) builder in
            let index = L.build_add index (L.const_int i32_t 0) "tmp" builder in
            let value = L.build_gep (lookup s)
                [| (L.const_int i32_t 0); index; |] "tmp" builder
            in L.build_store (gen_expression e2 builder) value builder


            | 2 -> let indexlist = List.map (fun x -> gen_expression x builder) el in

              let index = L.build_add (L.const_int i32_t 0)
```

```
                    (List.nth indexlist 1) "tmp" builder in
                let rows = L.build_mul (List.nth indexlist 0)  (L.const_int i32_t

 (List.nth il 1)) "tmp2" builder
                in let index = L.build_add index rows "tmp" builder in
                let value = L.build_gep (lookup s)
                    [| (L.const_int i32_t 0); index |] "tmp" builder
                in L.build_store (gen_expression e2 builder) value builder


            | _ -> raise(CodegenError("Invalid dim number"))

          )

        | S.S_ArrayLit(el, typ) -> L.const_array (ast_to_llvm_type typ) (Array.of_list
                (List.map (fun x-> gen_expression x builder) el))

        | S.S_Noexpr ->
          L.const_int i32_t 0

        | _ -> L.const_int i32_t 0

 in
 let global=
   let globals (typ, s, e) =
     match typ with A.Array(t,il,d) -> if e = S.S_Noexpr then ()
       else let e' = gen_expression e builder
         in ignore(L.build_store e' ( StringMap.find s global_vars) builder );
         ignore(e');
                  | _ -> (match e with S.S_Noexpr -> ()
```

```
                                        | _ -> let e' =
gen_expression e builder in
                                          ignore(L.build_store e
' (StringMap.find s global_vars) builder);
                                          ignore (e');)

    in List.iter globals sprog.S.globals

  in
  let add_terminal builder e =
    match L.block_terminator (L.insertion_block builder ) with
      Some _ -> ()
    | None -> ignore (e builder)
  in
  let rec gen_statement builder s =
    match s with
      S.S_Expr(e, typ) -> ignore(gen_expression e builder);
builder
    | S.S_Return(e, typ) -> ignore (match fdecl.S.
func_return_type with A.Void -> L.build_ret_void builder

        | _ -> L.build_ret (gen_expression e builder) builder
); builder
    | S.S_Block(sl) -> gen_stmt_list sl builder
    | S.S_If(e, then_expr, else_expr) -> let cond =
gen_expression e builder in
      let start_bb = L.insertion_block builder in
      let func = L.block_parent start_bb in
      let then_bb = L.append_block context "then" func in
      L.position_at_end then_bb builder;

      let _ = gen_statement builder then_expr in
      let new_then_bb = L.insertion_block builder in
      let else_bb = L.append_block context "else" func in
      L.position_at_end else_bb builder;

      let _ = gen_statement builder else_expr in
      let new_else_bb = L.insertion_block builder in
      let merge_bb = L.append_block context "ifcont" func in
      L.position_at_end merge_bb builder;

      let else_bb_val = L.value_of_block new_else_bb in
      L.position_at_end start_bb builder;

      ignore (L.build_cond_br cond then_bb else_bb builder);
```

```
      L.position_at_end new_then_bb builder; ignore (L.build_br merge_bb builder);
      L.position_at_end new_else_bb builder; ignore (L.build_br merge_bb builder);

      L.position_at_end merge_bb builder;

      ignore(else_bb_val); builder

    | S.S_For(inite, compe, incre, sl) ->
      let the_function = L.block_parent (L.insertion_block builder) in
      let _ = gen_expression inite builder in
      let loop_bb = L.append_block context "loop" the_function in
      let inc_bb = L.append_block context "inc" the_function in
      let cond_bb = L.append_block context "cond" the_function in
      let after_bb = L.append_block context "afterloop" the_function in





      ignore(L.build_br cond_bb builder);
      L.position_at_end loop_bb builder;
      ignore(gen_statement builder sl);

      let bb = L.insertion_block builder in
      L.move_block_after bb inc_bb;
      L.move_block_after inc_bb cond_bb;
      L.move_block_after cond_bb after_bb;
      ignore(L.build_br inc_bb builder);

      L.position_at_end inc_bb builder;
      let _ = gen_expression incre builder in
      ignore(L.build_br cond_bb builder);

      L.position_at_end cond_bb builder;

      let cond_val = gen_expression compe builder in
      ignore(L.build_cond_br cond_val loop_bb after_bb builder
```

```
);

      L.position_at_end after_bb builder;

      builder;

    | S.S_While(expr, body) -> let null_expr = S.S_IntLit(0) in
      gen_statement builder (S.S_For(null_expr,expr,null_expr,body))

    | S.S_Break -> builder
    | S.S_Continue -> builder

    | S.S_VarDecStmt(S.S_VarDecl((name,typ),sexpr)) ->
      match typ with A.Array(t,il,d) -> if sexpr = S.S_Noexpr then builder
        else
          let e' = gen_expression sexpr builder
          in ignore(L.build_store e' (lookup name) builder );
          ignore(e'); builder
                | _ -> (match sexpr with S.S_Noexpr -> builder
                                | _ -> let e' = gen_expression sexpr builder in
                                  ignore(L.build_store e' (lookup name) builder);
                                  ignore(e'); builder)
and
  gen_stmt_list sl builder =
  match sl with [] -> builder
             |  hd::[] -> gen_statement builder hd
             |  hd::tl -> ignore(gen_statement builder hd); gen_stmt_list tl builder
in
let builder = gen_statement builder (S.S_Block fdecl.S.func_body) in

add_terminal builder (match fdecl.S.func_return_type with A.Void -> L.build_ret_void

                                                  | t -> L.build_ret (L.const_int (ast_to_llvm_type t) 0))
in
```

```
  List.iter function_body sprog.S.functions;
  _le_module
```

8.7 Testall.sh

```sh
#!/bin/sh

# LePiX Regression testing script
# based on testall.sh for MicroC by Stephen Edwards

# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="lli"

# Path to the lepix compiler.  Usually "./lepix.native"
# Try "_build/lepix.native" if ocamlbuild was unable to create a symbolic link.
LEPIX="source/lepix.native -c"
#LEPIX="source/_build/lepix.native"

# Set time limit for all operations
ulimit -t 30

# Colors!
RED="\033[0;31m"
GREEN="\033[0;32m"
NC="\033[0m" # No Color

# To align status messages
size=0

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.lepix files]"
```

```
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
    if [ $size -eq 2 ] ; then
        echo "\t${RED}(_)${NC}"
    else
        echo "\t\t${RED}(_)${NC}"
    fi
     error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
     SignalError "$1 differs"
    echo "=========="
    echo "EXPECTED OUTPUT:"
    cat $2
    echo "=========="
    echo "ACTUAL OUTPUT:"
    cat $1
    echo "=========="

     echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
     SignalError "$1 failed on $*"
     return 1
    }
```

```
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
     SignalError "failed: $* did not report an error"
     return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.lepix//'`
    reffile=`echo $1 | sed 's/.lepix$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."
    size=`echo $((${#basename} + 4))`
    size=`echo $(($size/8))`

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&
    Run "$LEPIX" "<" $1 ">" "${basename}.ll" &&
    Run "$LLI" "${basename}.ll" ">" "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
     if [ $keep -eq 0 ] ; then
         rm -f $generatedfiles
     fi
     if [ $size -eq 2 ] ; then
        echo "\t${GREEN• •}()${NC}"
    else
        echo "\t\t${GREEN• •}()${NC}"
```

```
    fi
     echo "###### SUCCESS" 1>&2
    else
     echo "###### FAILED" 1>&2
     globalerror=$error
    fi
}

CheckFail() {
    # echo "in checkfail"
    error=0
    basename=`echo $1 | sed 's/.*\\///
                              s/.lepix//'`
    reffile=`echo $1 | sed 's/.lepix$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."
    size=`echo $((${#basename} + 4))`
    size=`echo $(($size/8))`

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$LEPIX" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
     if [ $keep -eq 0 ] ; then
         rm -f $generatedfiles
     fi
    if [ $size -eq 2 ] ; then
        echo "\t${GREEN• •}()${NC}"
    else
        echo "\t\t${GREEN• •}()${NC}"
    fi
     echo "###### SUCCESS" 1>&2
    else
     echo "###### FAILED" 1>&2
```

```
        globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
     k) # Keep intermediate files
         keep=1
         ;;
     h) # Help
         Usage
         ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail


if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.lepix tests/fail-*.lepix"
fi

for file in $files
do
    case $file in
     *test-*)
         Check $file 2>> $globallog
         ;;
     *fail-*)
         CheckFail $file 2>> $globallog
         ;;
     *)
         echo "unknown file type $file"
```

```
            globalerror=1
            ;;
    esac
done

if [ $globalerror -eq 0 ] ; then
    echo "\n${GREEN }( _)${NC}"
else
    echo "\n${RED}(  _)  >-${NC}"
fi
exit $globalerror
```