Pixmix

Alexandra Taylor
Christina Charles
Edvard Eriksson
Nathan Burgess

Table of contents

1. Introduction

PixMix is designed to be a programming language that facilitates image manipulation. It is intended to make actions such overlaying, compounding and changing brightness easier using a simple c-like syntax, that can be written, like python, in a few lines with no main method.

Ideally the language should act as a scripting language which can be used for quick and easy editing while also supporting more advanced functionality by allowing users to specify custom struct-like objects and libraries.

2. Language Tutorial

PixMix relies on a few tools in order to successfully build and run it's programs. Before you can use PixMix, you must run the following commands (or your system's equivalent) in a Linux environment, Windows is not currently supported.

- edit the file `/etc/apt/sources.list.d/llvm.list` and put the following text in it:
  `deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-5.0 main`
- `wget -O - http://apt.llvm.org/llvm-snapshot.gpg.key | sudo apt-key add -`
- `sudo apt update`
- `sudo apt install opam ocaml cmake llvm-5.0-dev llvm m4 pkg-config`
- `opam init -y`
- ``eval `opam config env` ``
- `opam install llvm`

Once all the tools have been successfully installed, navigate to the project root and type `make` to compile PixMix. To ensure that everything has installed and compiled properly, use the command `make tests` to run the PixMix test suite.

For individual testing, there is a `make test` command which will run a single test on a file named `test.pm` in the project root. To view more information, there are several other commands that can be used to run `test.pm`:

```
● make testast       Print the AST
● make testsast      Print the SAST
● make testlli       Print the LLVM IR code
● make testasl       Run the three above tests in tandem
● make debugtest     Run the test and display the state transitions
● make debug         Dump the entire state transition table to parser.output
```

3. Language Manual

Data Types

Primitive

num

The num data type is a signed, double precision, 64-bit two’s complement floating point number.

char

A char is an 8 bit single character enclosed in single quotation marks.

bool

A 1 byte binary indicator storing either 1 or 0, or true or false.

Reference

Arrays

Arrays store a series of user specified value with the same data type.

Objects

Objects are user defined data types containing an optional list of local variable declarations and/or optional function declarations.

Lexical Elements

Comments

`#:`	Beginning of a multi line comment
`:#`	End of a multi line comment
`#`	Single line comment

```
#: I am a multi
line comment! :#

# I am a single line comment!
```

Identifiers

An identifier is a case-sensitive sequence of ASCII characters, digits, and underscores. Identifiers must begin with either a character or a underscore immediately followed by a character.

Keywords

The following keywords are reserved and cannot be used as identifiers in Pixmix:

and	or	not	if	else
void	int	while	break	for
continue	null	return	isnt	node
bool	string	float	Object	Array

Literals

LIterals are fixed value, syntactic representations of bool, char, num, and string data types.

Num Literals

A num literal is expressed as an integer, a decimal point, and a fraction part. A negative num is preceded by a '-' symbol to denote negativity.

Example: `1.567`

Char Literals

A char literal is a single ASCII character enclosed in single quotes.

Example: `‘a’`

Bool Literals

Boolean literals have two possible values: true and false. They represent logical true and logical false.

Example: `true`

String Literals

String literals are a sequence of characters enclosed in double quotes.

Example: “`Hello, World!`”

Separators

A separator is a single character token which serves to separate other tokens. Separators in Pixmix are:

(	{ LPAREN }	)	{ RPAREN }
[	{ LSQUARE }	]	{ RSQUARE }
{	{ LCURL }	}	{ RCURL }
,	{ COMMA }	.	{ DOT }
;	{ SEMI }		

White Space

White space refers to several characters: the space character, the tab character, the newline character, and the carriage-return character. Whitespace characters are used to separate tokens, however they are not tokens themselves. Outside of token separation and string constraints, whitespace is ignored.

Expressions and Operators

Expressions

An expressions in Pixmix can be:

- A single literal
- An arithmetic operation
- An identifier
- An expression assigned to an identifier

Arithmetic Operators

PixMix utilizes the following arithmetic operators, which all have left to right associativity:

( )	Parenthetical grouping
*	Multiplication
/	Division
%	Modulo
+	Addition
-	Subtraction

Comparison Operators

Pixmix utilizes the following comparison operators, which all have left to right associativity and return true if the evaluation is successful, false otherwise:

==	Equal to
is	Equal to
<	Less than

>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to
not	Not equal to

Logical Operators

Logical and operators compare two expressions and return true if both expressions evaluate to true, and false if either expression evaluates to false. The logical and operator is represented as either ‘&&’ or simply ‘and’.

```
if (x > y and y < z) {
	statement;
}
```

Logical or operators compare two expressions and return true if both expressions evaluate to true, and false if either expression evaluates to false. The logical and operator is represented as either ‘||’ or simply ‘or’.

```
if (x > y or y < z) {
	statement;
}
```

Unary Operators

A unary operator operates on a single operand and is associated right to left. Pixmix utilizes the following unary operators:

-	Unary minus
!	Logical not
not	Logical not

The following example prints “true”:

```
bool f = false;

If (not f) {
    print ("true");
}
else {
    print("false");
}
```

Statements

Expression Statement

An expression statement is an expression followed by a semicolon. When the code is executed, the expression will be evaluated.

Variable Declaration Statement

In PIxMix lvalues are variables and rvalues are expressions. In order to assign a variable the lvalue and the rvalue must be of the same type.

A variable declaration consists of the primitive type, an identifier, an optional initial value, and a semicolon. If no initial value is declared, default values will be applied as follows: a num will be initialized to 0.0, a bool will be initialized to 'false' and a string will be initialized to null.

Examples:

```
num x;
num y = 3;
bool a;
string = "Hello World!"
```

Control Flow Statements

If

The 'if' statement evaluates a conditional boolean expression and executes a block of code if the expression evaluates as true. If the expression evaluates as false, then the subsequent block of code is skipped.

```
if (conditional expression) {
	Code to be executed if expression is true;
}
```

If Else

Similar to the if statement, an if else statement evaluates the conditional expression and uses the result of that evaluation to determine where to pass control. If the conditional expression evaluates to true, the block of code immediately following the conditional statement will be executed. If the conditional expression evaluates to false, control will be passed to the block of code immediately following the 'else' keyword.

```
if (conditional expression {
	Code to be executed if expression is true;
}
else {
	Code to be executed if expression is false;
}
```

For loop

The for loop allows for repeated execution of a block of code a specified number of times. The for loop consists of three statements: an initialized variable, a conditional expression, and an arithmetic operation on the initialized variable. The second statement is evaluated at the beginning of every loop, if it is true, the block of code will run, if it is false the loop will end and control will be passed on. The third statement is executed at the end of every loop.

```
for (i = 0; i < 10; i = i + 1) {
	Code to be executed if middle expression is true;
}
```

While loop

The while loop allows for repeated execution of a block of code as long as the conditional statement continues to evaluate to true. The block of code is executed first, then the conditional statement is evaluated. All subsequent executions happen if the conditional evaluates to true, once it evaluates to false, the loop stops and control is passed on.

```
while (conditional expression) {
	Code to be executed if expression is true;
}
```

Functions

Function Declaration

Functions are declared with a data type, identifier, optional parameters, a statement list, and an optional return statement. The return keyword must be followed by an expression of the same type as the function. If there is no return statement inside the function, the ‘void’ data type may be used. The parameters and their types are in parentheses, and the function body is enclosed in curly brackets.

With return statement:

```
num a = 5;
num b = 2;

num add(num x, num y) {
	return x + y;
}
```

Without return statement:

```
void add(num x, num y){
      print(x + y);
}
```

Calling Functions

Functions are called with the function identifier and and a list of arguments to be pass into the function in parentheses. The arguments must match the parameters of the function in number and type.

```
num add(num x, num y) {
      return x + y;
}

print(add(1, 2));
```

Calls to functions which have a return value can be used as expressions and assigned to variables of the same type.

```
num a = add(3, 5);
```

Objects

Object Declaration

Objects are declared with the keyword ‘Object’ followed by an identifier, assignment operator, a body which is enclosed in curly braces, and a semicolon. WIthin the body of an object the user may define variables as well as functions.

```
Object math = {
      num someVal = 54;

      num double(num val) {
            return val * val;
      }
};
```

Calling functions within Objects

Calls to functions within objects are done by specifying the object id, followed by the '.' accessor, the name of the function and the function's optional parameters.

The following statement will return 9:

```
math.double(3);
```

Arrays

Array Declaration

Arrays are declared using the keyword 'Array' followed by an identifier, data type, assignment operator, the length of the array inside square brackets, and a semicolon. The following example initializes an array of 5 nums:

```
Array num a = [5];
```

Array Operations

Array elements are accessed by specifying the array identifier and the index of the desired element in brackets. The return value of the array access operation can be assigned to a variable of the same type as the array.

```
num x;
x = a[2];
```

Values can be assigned to specific array elements as long as the value is of the same type as the array.

a[1] = 3;

4. Project Plan

Processes for planning, specification, development and testing

The team met most Sundays for 5-6 hours to work on, plan or discuss issues with the language. We met with TA Heather to check in weekly or biweekly on our progress. Planning and specification generally went through a cycle of specifying deal functionalities and look, then attempt to implement it, see what was possible to implement and then revising our design. We learned quickly that the best approach was to simplify syntax first, get functionalities working and then adapt syntax as possible in incremental steps.

We started by designing a test suite for bare bones functionality such as printing a string, adding two integers or storing a variable. The idea was to stick with test driven development as much as possible, given, initially, our limited ability to know exactly what and how to test. As our specifications changed, we updated the tests to stay up-to-date with current specifications. Most syntax stayed close to our original design, but throughout the process we removed the need for a main method, ints and updated the syntax for objects and arrays. Although, we had a wiki for specification that we updated continuously, our goal was to be able to refer to the test suite for specification, making it easier to test and seeing how our code worked in relation to the specified syntax.

As a general rule, when we disagreed on a design or implementation issue, we would vote to decide on changes or approaches. If the decision was vote was split, the Language Guru's vote would determine the vote in the case of syntax issues; the System Architect's vote would determine the vote for implementation or other backend issues; and the Tester would determine the votes regarding testing related issues. The manager's main role in this process was to act as a counterweight to ambition and consider the constraints of time and functionality.

The early development phase - setting up scanner, parser, ast - consisted of the whole group programming together on one screen and thinking out loud as we went over the microC code and made changes. The goal of this was to make sure that everyone was introduced to the initially intimidating codebase and that we all agreed on the specifications and kept a consistent variable naming. This meant we spent a lot of time on the front end of the compiler, but also that everyone had a basic understanding of the code base and how we would implement pixmix before we split the coding up.

After we were finished with the scanner parser and ast we generally split the code up by functionality and began working individually. We still met on Sundays to help each other, talk through the code and discuss and vote on any changes. Dividing the code up based on functionality was helpful in that everyone had to understand every file and what one person learned about for example codegen, they could show or explain to another. It also meant that the implementation logic for each functionality was kept consistent, avoiding someone making changes that broke the code for someone else.

We used git for version, slack for discussing and Trello for scheduling and setting deadlines. Slack was used both to discuss the project and to keep track of and comment on specific commits. Online communication was meant to act as a complement to in person communication, not a substitute. We stressed meeting and discussing things in person in order to have more control over and ability to include everyone in the decision and development process and to resolve conflicts correctly.

Style guide

While writing code we did our best to adhere to the following style guidelines:

- Indentation is done with a series of 4 spaces

A common standard practice. Makes code more legible and logic easier to follow.

Yes:
```
let someFunc a b =
    let c = a + b in
        c + 2
```
No:
```
let someFunc a b =
  let c = a + b in
         c + 2
```

- Variables and functions are given descriptive, meaningful names

This allows us to use less comments and make the code more legible and less prone to errors.

Yes:
```
ArrayCreate
```

No:
```
Create
```
- Variable names are written in camelCase

Exceptions of this rule are variables used with the Llvm library since Llvm for Ocaml uses underscore as per the Llvm core API documentation: https://llvm.moe/ocaml/Llvm.html

Yes:
```
varType
```

No:
```
var_type
```

- Variable and function names are named consistently across files

Make sure that functions that do similar things have similar logic behind their names to maintain consistency. A variable that is used in two files should have the same name unless it has somehow been manipulated.

- Keep lines of code under 80 characters long

There are some exceptions that are ok, when the logic becomes less clear if we keep it on the same line.

- Every pattern matching begins with |

This makes the code easier to read, more consistent and less prone to errors

Yes:
```
and varType =
    | NullType
    | VoidType
    | IntType
    | NumType
    | StringType
    | BoolType
    | ImageType
    | ObjectType
    | ArrayType
```

No:
```
and varType =
      NullType
    | VoidType
    | IntType
    | NumType
    | StringType
    | BoolType
    | ImageType
    | ObjectType
    | ArrayType
```

Project timeline

Date	**Milestone**
October	LRM, grammar, specification
November 1st	Setup develop environment and git
November 4th	Development begins
November 5th	Scanner ready
November 7th	Hello pixmix!
November 30th	Frontend largely complete, work split into tasks/functionalities
December 2nd	necessity for main function removed, declaration and assignment same line done, code compiles and runs for basic arithmetic functionality, strings, printing and methods
December 4th	new set of tests added for arrays and objects, some updates in syntax
December 7th	Update parts of syntax to make implementation of objects and arrays easier
December 10th	begin work on small standard Math.lib and skeleton Image and Pixel libraries
December 15th	object function calls working
December 16th	array declarations working

Roles and responsibilities

Language Guru - Christina
Manager - Edvard
System Architect - Nathan
Tester - Allie

Language Guru - Christina

Responsibilities:
Defining, analyzing and updating language syntax across files and in manual
Coordinate tests and specifications with tester and system architect

Worked on:

- Scanner, parser, ast,sast
- objects
- declaration/assignments
- LRM

Veto vote on:
Language consistency or syntactical issues

Manager - Edvard

Responsibilities:
Managing meetings, deadlines, deliverables
Acting as a counterpoint to the ambition of the team, and place goals in relation to timeline

Worked on:

- Scanner, parser, ast, sast
- adapting semant.ml to grammar changes in front end
- arrays
- prioritizing, planning and organizing workflow and meetings

Veto vote on:
Anything relating to prioritization of work and deadlines

System Architect - Nathan

Responsibilities:

Set up, organize and maintain development environment, file structure and tools
Coordinate and evaluate the implementation of functionalities in the language

Worked on:
- scanner, parser, ast, sast, codegen
- objects
- organizing project structure
- setting up and updating Makefile
- updating codegen for grammar changes

Veto vote on:
How to implement functionalities, llvm specific decisions

Tester - Allie

Responsibilities:
Building, updating and maintaining and automated test suite
Coordinate with other members to determine what tests, the require or want to update

Worked on:
- Scanner, parser, ast
- shell script for test suite that shows clearly each pass and fail
- building tests
- LRM

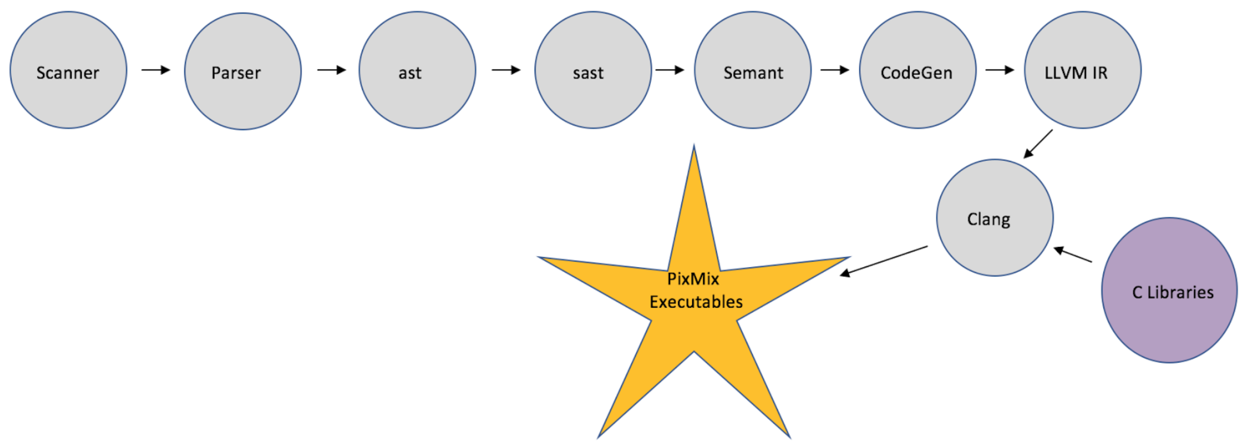
Veto on:
Testing related issues

Development environment

Each team member ran an Ubuntu 17.04 virtual machine with clang 4.0 and LLVM 4.0 or 5.0 for development.

Code was version controlled via GitHub.

5. Architectural Design



Major components of code

- PixMix reads in a file provided over stdin and the relevant channel over to Lexing which then generates a lexbuf.
- Lexbuf is tokenized by the Scanner and then passed into the Parser which generates an AST based on our grammar.
- Once the AST has been generated, it's fed into our SAST module which does some rearranging of the original code to get features to work.
- Once SAST has finished, it passes the result into the Semant which does some final checks to ensure the provided source can be compiled into a working executable.
- Once Semand has finished, the SAST is handed off to Codegen which then translates the SAST into LLVM IR.
- Once the IR has been generated, it's piped through clang and combined with the compiled bytecode of our C library and a final executable is produced.

Interfaces

- Describe the interfaces between the components

Who implemented each component:

Scanner:	Allie, Nathan, Christina, Edvard
Parser:	Allie, Nathan, Christina, Edvard
Abstract Syntax Tree:	Allie, Nathan, Christina, Edvard
Tests:	Allie
Test script:	Allie
Assignment/declarations:	Nathan, Christina
Unary/Binary operators:	Christina
Arrays:	Edvard
Objects:	Nathan and Christina
Adapting semant to parser, ast changes:	Edvard
Adapting codegen to parser ast changes:	Nathan
Sast:	Nathan, Christina, Edvard
Math Library:	Nathan
Makefile:	Nathan
Example code:	Allie, Nathan

6. Test Plan

All of our tests involved black box testing. Although, the parser.mly file was heavily referenced in the development of tests, the testing remained black box (versus white box) oriented as the parser.mly was referenced more as a language specification manual than a line by line internal analysis. In all tests, the tests were written from a pseudo-user perspective. Unit Tests and Integration tests were utilized. For every new release regression was tested for by running the test suite to check that old functionality was not broken when new functionality was added. If regression had occurred the group mates was notified so that issues could be resolved before further work was done on PixMix.

Unit Testing

Throughout work on the PixMix language as each new aspect of the language was developed, unit tests were written to verify that it worked as expected. A majority of new language additions worked as expected but some did not and real time unit testing was an excellent tool in catching and resolving bugs. For each new language addition a test, or tests, were added to the test suite.

Integration Testing

Integration tests were added over the course of project development. Integration test design was intended to verify integrated components of the language. Integration testing was more common in test implementation as many of PixMix's tests function as snippets of longer code that a user may utilize. Integration tests were also developed in close communication with the PixMix group. As the goals, and realities, of the project changed so did the scope and type of integration test. Integration testing was also useful for debugging code by checking either for test failure or for a test output different than was expected. As an example, via multiple unexpected test failures integration testing revealed that our while loop was functioning as a do while loop. When an integration test failed more tests were written to see if the source code bug could be clarified. Debugging then occurred via communication with the PixMix group member responsible for the language feature. For example, when Array Access continued to fail debugging was attempted with the group member assigned to Arrays. Integration tests also closely followed the language reference manual and the language guru was an invaluable asset to me in individual test development.

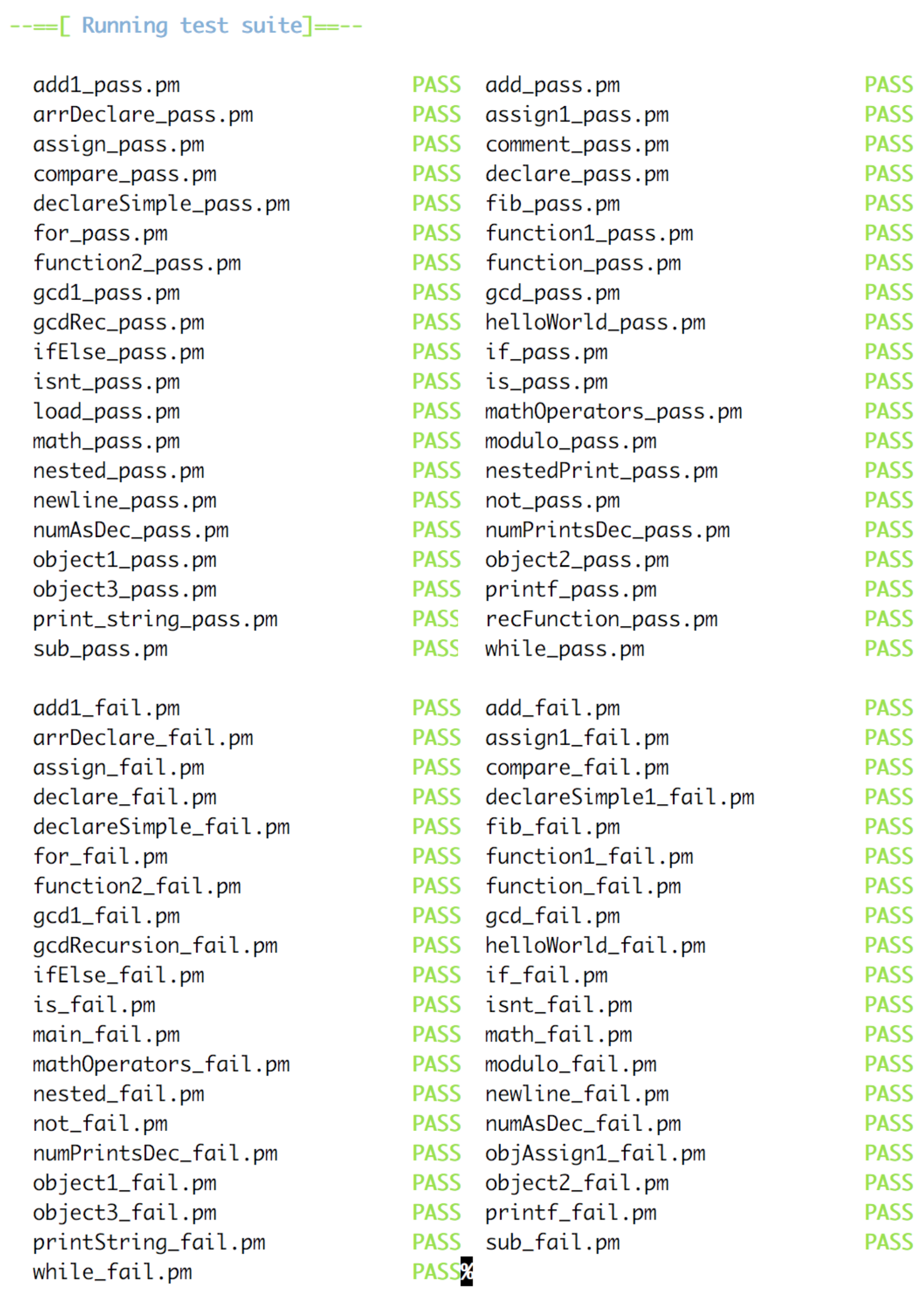
Test Suites and Automation

Tests were divided into two folders: `tests_pass` which contains test cases that are expected to pass and `tests_fail` which contains test cases that are expected to fail. The tests contained in both folders are run by typing `make tests`. This generates a list of all tests and a PASS or

FAIL . At their terminal line, all tests include a print function which prints `finished`. The shell script checks tests in `tests_pass` and `tests_fail` for the presence of `finished` and prints `PASS` if it is found and `FAIL` if it is not found in and vice versa for `tests_fail`.

The `testall.sh` test script (`/pixmix/testall.sh`) runs all the files within the `tests_pass` and `tests_fail` directories. The files must have the extension `.pm`.

Output of test script:



```
--==[ Running test suite]==--

  add1_pass.pm                   PASS  add_pass.pm                    PASS
  arrDeclare_pass.pm             PASS  assign1_pass.pm                PASS
  assign_pass.pm                 PASS  comment_pass.pm                PASS
  compare_pass.pm                PASS  declare_pass.pm                PASS
  declareSimple_pass.pm          PASS  fib_pass.pm                    PASS
  for_pass.pm                    PASS  function1_pass.pm              PASS
  function2_pass.pm              PASS  function_pass.pm               PASS
  gcd1_pass.pm                   PASS  gcd_pass.pm                    PASS
  gcdRec_pass.pm                 PASS  helloWorld_pass.pm             PASS
  ifElse_pass.pm                 PASS  if_pass.pm                     PASS
  isnt_pass.pm                   PASS  is_pass.pm                     PASS
  load_pass.pm                   PASS  mathOperators_pass.pm          PASS
  math_pass.pm                   PASS  modulo_pass.pm                 PASS
  nested_pass.pm                 PASS  nestedPrint_pass.pm            PASS
  newline_pass.pm                PASS  not_pass.pm                    PASS
  numAsDec_pass.pm               PASS  numPrintsDec_pass.pm           PASS
  object1_pass.pm                PASS  object2_pass.pm                PASS
  object3_pass.pm                PASS  printf_pass.pm                 PASS
  print_string_pass.pm           PASS  recFunction_pass.pm            PASS
  sub_pass.pm                    PASS  while_pass.pm                  PASS

  add1_fail.pm                   PASS  add_fail.pm                    PASS
  arrDeclare_fail.pm             PASS  assign1_fail.pm                PASS
  assign_fail.pm                 PASS  compare_fail.pm                PASS
  declare_fail.pm                PASS  declareSimple1_fail.pm         PASS
  declareSimple_fail.pm          PASS  fib_fail.pm                    PASS
  for_fail.pm                    PASS  function1_fail.pm              PASS
  function2_fail.pm              PASS  function_fail.pm               PASS
  gcd1_fail.pm                   PASS  gcd_fail.pm                    PASS
  gcdRecursion_fail.pm           PASS  helloWorld_fail.pm             PASS
  ifElse_fail.pm                 PASS  if_fail.pm                     PASS
  is_fail.pm                     PASS  isnt_fail.pm                   PASS
  main_fail.pm                   PASS  math_fail.pm                   PASS
  mathOperators_fail.pm          PASS  modulo_fail.pm                 PASS
  nested_fail.pm                 PASS  newline_fail.pm                PASS
  not_fail.pm                    PASS  numAsDec_fail.pm               PASS
  numPrintsDec_fail.pm           PASS  objAssign1_fail.pm             PASS
  object1_fail.pm                PASS  object2_fail.pm                PASS
  object3_fail.pm                PASS  printf_fail.pm                 PASS
  printString_fail.pm            PASS  sub_fail.pm                    PASS
  while_fail.pm                  PASS%
```

If there was an unexpected, or undesirable, result an individual could attempt to debug by running a test on an individual file via copying the file into `/pixmix/test.pm` and typing `make testasl`. This prints the output of the AST, the SAST, the LLVM IR and the test output. Project PixMix used this functionality for debugging code. All tests which generated a `PASS` in the test suite test, individually had their actual output checked against their expected output by copying the test file into test.pm and typing `make test`. The comparison of expected vs. actual output was not fully automated and was done with the power of human reading comprehension.

`math_pass.pm`

```
Object Math = {
    num add(num a, num b) {
        return a + b;
    }

    num mod(num a, num b) {
        return a % b;
    }

    num pow(num a, num b) {
        num base = a;
        while(b > 1) {
            a = a * base;
            b = b - 1;
        }
        return a;
    }
};

printf("4 + 1 = %f\n", Math.add(4, 1));
printf("10 % 6 = %f\n", Math.mod(10, 6));
printf("2^5 = %f\n", Math.pow(2, 5));
```

Output of `make testasl` for `math_pass.pm`:

```
--==[ Printing the AST... ]==--
Object Math = {
num add(num a, num b)
{
return a + b;
}
num mod(num a, num b)
{
return a % b;
}
num pow(num a, num b)
{
num base = a;
while (b > 1.) a = a * base;
b = b - 1.;
return a;
}
};
printf("4 + 1 = %f\n", Math.add(4., 1.));
printf("10 % 6 = %f\n", Math.mod(10., 6.));
printf("2^5 = %f\n", Math.pow(2., 5.));
```

```
--==[ Printing the SAST... ]==--


num main.Math.pow(num a;
, num b;
)
{
num base;
base = a;
while (b > 1.) a = a * base;
b = b - 1.;
return a;
}
num main.Math.mod(num a;
, num b;
)
{
return a % b;
}
num main.Math.add(num a;
, num b;
)
{
return a + b;
}
num main.Math()
{
}
int main()
{
printf("4 + 1 = %f\n", main.Math.main.Math.add(4., 1.));
printf("10 % 6 = %f\n", main.Math.main.Math.mod(10., 6.));
printf("2^5 = %f\n", main.Math.main.Math.pow(2., 5.));
}
```

```
--==[ Printing the LLVM IR... ]==--
; ModuleID = 'PixMix'
source_filename = "PixMix"

@str_tmp = private unnamed_addr constant [12 x i8] c"4 + 1 = %f\0A\00"
@str_tmp.1 = private unnamed_addr constant [13 x i8] c"10 % 6 = %f\0A\00"
@str_tmp.2 = private unnamed_addr constant [10 x i8] c"2^5 = %f\0A\00"
@main.Math.add.a = global double 0.000000e+00
@main.Math.add.b = global double 0.000000e+00
@main.Math.mod.a = global double 0.000000e+00
@main.Math.mod.b = global double 0.000000e+00
@main.Math.pow.a = global double 0.000000e+00
@main.Math.pow.b = global double 0.000000e+00
@main.Math.pow.base = global double 0.000000e+00

declare i32 @VoidtoInt(i8*)

declare double @VoidtoFloat(i8*)

declare i1 @VoidtoBool(i8*)

declare i8* @VoidtoString(i8*)

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

declare i32 @printBool(i1)

define double @main.Math.pow(double, double, ...) {
entry:
  store double %0, double* @main.Math.pow.a
  store double %1, double* @main.Math.pow.b
  %a = load double, double* @main.Math.pow.a
  store double %a, double* @main.Math.pow.base
  br label %while

while:                                            ; preds = %while_body, %entry
  %b2 = load double, double* @main.Math.pow.b
  %flt_sgttmp = fcmp ogt double %b2, 1.000000e+00
  br i1 %flt_sgttmp, label %while_body, label %merge

while_body:                                       ; preds = %while
  %a1 = load double, double* @main.Math.pow.a
  %base = load double, double* @main.Math.pow.base
  %flt_multmp = fmul double %a1, %base
  store double %flt_multmp, double* @main.Math.pow.a
  %b = load double, double* @main.Math.pow.b
  %flt_subtmp = fsub double %b, 1.000000e+00
  store double %flt_subtmp, double* @main.Math.pow.b
  br label %while
merge:                                            ; preds = %while
  %a3 = load double, double* @main.Math.pow.a
  ret double %a3
}

define double @main.Math.mod(double, double, ...) {
entry:
  store double %0, double* @main.Math.mod.a
  store double %1, double* @main.Math.mod.b
  %a = load double, double* @main.Math.mod.a
  %b = load double, double* @main.Math.mod.b
  %flt_fremtmp = frem double %a, %b
  ret double %flt_fremtmp
}

define double @main.Math.add(double, double, ...) {
entry:
  store double %0, double* @main.Math.add.a
  store double %1, double* @main.Math.add.b
  %a = load double, double* @main.Math.add.a
  %b = load double, double* @main.Math.add.b
  %flt_addtmp = fadd double %a, %b
  ret double %flt_addtmp
}

define double @main.Math(...) {
entry:
  ret double 0.000000e+00
}

define i32 @main(...) {
entry:
  %main.Math.add_result = call double (double, double, ...) @main.Math.add(double 4.000000e+00, double 1.000000e+00)
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([12 x i8], [12 x i8]* @str_tmp, i32 0, i32 0), double %main.Math.add_result)
  %main.Math.mod_result = call double (double, double, ...) @main.Math.mod(double 1.000000e+01, double 6.000000e+00)
  %printf1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([13 x i8], [13 x i8]* @str_tmp.1, i32 0, i32 0), double %main.Math.mod_result)
  %main.Math.pow_result = call double (double, double, ...) @main.Math.pow(double 2.000000e+00, double 5.000000e+00)
  %printf2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @str_tmp.2, i32 0, i32 0), double %main.Math.pow_result)
  ret i32 0
}
```

```
--==[ Printing output from running the program... ]==--
make[1]: Entering directory '/home/vagrant/code/pixmix'
4 + 1 = 5.000000
10 % 6 = 4.000000
2^5 = 32.000000
```

Lessons Learned

Allie

- In test script composition, comparing output of test files has some major advantages. I ended up comparing anticipated versus expected output of tests, test by test which was tedious.
- Tester is perhaps the role which requires the most communication. You must know when new features are added or you may not be able to test as efficiently or debug from test results as efficiently as possible.
- Don't be afraid to ask your teammates questions if you are stuck. Your teammate may understand something more easily or better than you. It is okay to not understand. If you have good teammates they answer your questions and most people prefer it to time wasted, especially on such a time constrained project.
- I really enjoy testing. Although we were all involved in multiple parts of the project, I took on the role of the tester to learn more about the process. This is the most extensive testing I have worked on in any of my coursework at Columbia and I like it more than development.

Christina

- Frequent and open communication is very important for a group project of this scope
- Meeting in person and making sure everyone has a chance to speak and ask questions will help reduce the chance that someone gets left behind
- New features should be added incrementally and tested often
- Set many internal deadlines to stay on track, not just the ones set by Prof. Edwards

Eddie

- Meeting in person is crucial to be able to communicate, resolve conflict, solve problems and bond as a group. Make sure everyone gets equal speaking time and is included in the decision process.
- Efficient meetings require preparing, working and studying beforehand and focusing on the work during the meeting
- The most efficient approach is to delegate by component/functionality and not by file
- Progress in this project is not measured in lines of code and number of changes, but in the quality and precision of the code; 5 good lines of code can often do more than 50 lines of bad code
- Given the limited time, it is better to adapt the aesthetic aspects of the language to what is easy to implement first. Then when you have a forking compiler, cross off things on our wish list one by one.

Nathan

- Experiment more and write small programs to help cement understanding on anything that might be shaky.
- Prototype certain features outside of the main project before implementing so that the path to implementation is more clear.

Advice the team has for future teams

- Start early
- Split the project into functionalities/components (e.g. arrays, objects) rather than by file (e.g. codegen.ml, parser.mly)
- After hello world is due, spend an entire day (on your own), at least, going over and experimenting with the microc code and make sure to get a basic understanding of it as soon as possible.
- Weekly meetings in the beginning of the semester, biweekly meetings : one short, one at least 6 hours starting around the midpoint of the semester.

Appendix

ast.ml

```
(*
    Authors:
    Nathan Burgess
    Christina Charles
    Edvard Eriksson
    Alexandra Taylor
 *)

type binop =
    | Add
    | Sub
    | Mult
    | Div
    | Equal
    | Neq
    | Leq
    | LThan
    | GThan
    | Geq
    | And
    | Or
    | Mod

and unop =
    | Neg
    | Not

and varType =
    | NullType
    | VoidType
    | IntType
    | NumType
    | StringType
    | BoolType
    | ArrayType of varType

and formal = Formal        of varType * string

and local  = Local         of varType * string * expr

and expr =
    | Null
    | Noexpr
    | NumLit                of float
    | IntLit                of int
    | StringLit             of string
    | BoolLit                of bool
    | Binop                  of expr * binop * expr
    | Unop                   of unop * expr
    | Id                     of string
    | Assign                 of string * expr
    | Call                   of string * expr list
    | CallObject             of string * string * expr list
    | ObjectAccess           of string * string
    | ArrayCreate            of varType * string * expr
    | ArrayAccess            of expr * expr
    | ArrayAssign            of expr * expr * expr

and stmt =
    | Expr                   of expr
    | Return                 of expr
    | For                    of expr * expr * expr * stmt list
    | If                     of expr * stmt list * stmt list
    | While                  of expr * stmt list
    | Variable               of local
    | Function               of funcDecl
    | Object                 of objBody

and objBody = {
    objName     :            string;
    objStmts    :            stmt list;
}

and funcDecl = {
    returnType  :            varType;
    name        :            string;
    args        :            formal list;
    body        :            stmt list;
}

and program = stmt list

let rec stringOfBinop = function
    | Add            -> "+"
    | Sub            -> "-"
    | Mult           -> "*"
    | Div            -> "/"
    | Equal          -> "=="
    | Neq            -> "!="
    | LThan          -> "<"
    | Leq            -> "<="
    | GThan          -> ">"
    | Geq            -> ">="
    | And            -> "&&"
    | Or             -> "||"
    | Mod            -> "%"
```

```
and stringOfUnop e = function
    | Neg           -> "-" ^ stringOfExpr e
    | Not           -> "!" ^ stringOfExpr e

and stringOfVarType = function
    | NullType      -> "null"
    | VoidType      -> "void"
    | IntType       -> "int"
    | NumType       -> "num"
    | StringType    -> "string"
    | BoolType      -> "bool"
    | ArrayType(t)  -> "[" ^ stringOfVarType t ^ "]"

and stringOfLocal = function
    | Local(t, s, e) -> stringOfVarType t ^ " " ^ s ^ " = " ^ stringOfExpr e ^ ";\n"

and string_of_formal = function
    | Formal(t, s) -> stringOfVarType t ^ " " ^ s

and stringOfExpr = function
    | Null -> "null"
    | Noexpr -> "undefined"
    | IntLit i -> string_of_int i
    | NumLit i -> string_of_float i
    | StringLit s -> "\"" ^ String.escaped s ^ "\""
    | BoolLit b -> if b then "true" else "false"
    | Binop(e1, op, e2) -> stringOfExpr e1 ^ " " ^ stringOfBinop op ^ " " ^ stringOfExpr e2
    | Unop(op, e) -> stringOfUnop e op
    | Id s -> s
    | Assign(s, e) -> s ^ " = " ^ stringOfExpr e
    | ArrayCreate(t, n, e) -> "Array " ^ stringOfVarType t ^ " " ^ n ^ " = " ^ " [" ^ stringOfExpr e ^
"]"
    | ArrayAccess(arrCreate, index) -> stringOfExpr arrCreate ^ "[" ^ stringOfExpr index ^ "]"
    | ArrayAssign(a,b,c) -> stringOfExpr a ^ " [" ^ stringOfExpr b ^ "] = " ^ stringOfExpr c
    | Call(f, e) -> f ^ "(" ^ String.concat ", " (List.map stringOfExpr e) ^ ")"
    | CallObject(o, f, e) -> o ^ "." ^ f ^ "(" ^ String.concat ", " (List.map stringOfExpr e) ^ ")"
    | ObjectAccess(o, v) -> o ^ "." ^ v

and stringOfObject o =
    "Object " ^ o.objName ^ " = {\n"
        ^ String.concat "" (List.map stringOfStatement o.objStmts) ^ "};\n"

and stringOfFunction f =
    stringOfVarType f.returnType ^ " " ^ f.name ^ "(" ^
    String.concat ", " (List.map string_of_formal f.args) ^ ")\n{\n" ^
    String.concat "" (List.map stringOfStatement f.body) ^ "}\n"

and stringOfStatement = function
    | Expr(expr) -> stringOfExpr expr ^ ";\n";
    | Return(expr) -> "return " ^ stringOfExpr expr ^ ";\n"
    | For(e1, e2, e3, s) -> "for (" ^ stringOfExpr e1  ^ " ; " ^ stringOfExpr e2 ^ " ; " ^ stringOfExpr
e3  ^ ") "
        ^ String.concat "" (List.map stringOfStatement s)
    | If(e, s1, s2) ->  "if (" ^ stringOfExpr e ^ ")\n"
        ^ String.concat "" (List.map stringOfStatement s1)
        ^ "else\n" ^ String.concat "" (List.map stringOfStatement s2)
    | While(e, s) -> "while (" ^ stringOfExpr e ^ ") " ^ String.concat "" (List.map stringOfStatement s)
    | Variable(v) -> stringOfLocal v
    | Function(f) -> stringOfFunction f
    | Object o -> stringOfObject o

and stringOfProgram stmnts =
    String.concat "" (List.map stringOfStatement stmnts) ^ "\n"
```

sast.ml

```
(*
    Authors:
    Nathan Burgess
    Christina Charles
 *)

module A = Ast
module StringMap = Map.Make(String)

type binop =
    | Add
    | Sub
    | Mult
    | Div
    | Equal
    | Neq
    | Leq
    | LThan
    | GThan
    | Geq
    | And
    | Or
    | Mod

and unop =
    | Neg
    | Not

and varType =
    | NullType
    | VoidType
    | IntType
    | NumType
    | StringType
    | BoolType
    | ObjectType
    | ArrayType             of varType

and formal = Formal         of varType * string

and local  = Local          of varType * string * expr

and expr =
    | Null
    | Noexpr
    | NumLit                of float
    | IntLit                of int
    | StringLit             of string
    | BoolLit               of bool
    | Binop                 of expr * binop * expr
    | Unop                  of unop * expr
    | Id                    of string
    | Assign                of string * expr
    | ArrayCreate           of varType * string * expr
    | ArrayAccess           of expr * expr
    | ArrayAssign           of expr * expr * expr
    | Call                  of string * expr list
    | CallObject            of string * string * expr list
    | ObjectAccess          of string * string

and stmt =
    | Expr                  of expr
    | Return                of expr
    | For                   of expr * expr * expr * stmt list
    | If                    of expr * stmt list * stmt list
    | While                 of expr * stmt list
    | Object                of objBody

and objBody = {
    objName     :           string;
    objStmts    :           stmt list;
    objLocals   :           formal list;
}

and funcDecl = {
    returnType  :           varType;
    name        :           string;
    args        :           formal list;
    body        :           stmt list;
    locals      :           formal list;
    parent      :           string;
}

and program = funcDecl list

(* SAST Printing Functions *)
let rec stringOfBinop = function
    | Add           -> "+"
    | Sub           -> "-"
    | Mult          -> "*"
    | Div           -> "/"
    | Equal         -> "=="
    | Neq           -> "!="
    | LThan         -> "<"
    | Leq           -> "<="
    | GThan         -> ">"
    | Geq           -> ">="
    | And           -> "&&"
    | Or            -> "||"
    | Mod           -> "%"

and stringOfUnop = function
    | Neg           -> "-"
    | Not           -> "!"

and stringOfVarType = function
    | NumType -> "num"
    | IntType -> "int"
    | StringType -> "string"
    | BoolType -> "bool"
    | NullType -> "null"
    | ArrayType(t) -> "Array [" ^ stringOfVarType t ^ "]"

and stringOfFormal (Formal(t,s)) = stringOfVarType t ^ " " ^ s ^ ";\n"

and stringOfLocal (Local(t,s,e)) = stringOfVarType t ^ " " ^ s ^ " = " ^ stringOfExpr e ^ ";\n"

and stringOfExpr = function
    | Null -> "null"
    | Noexpr -> "noexpr"
    | NumLit f -> string_of_float f
    | StringLit s -> "\"" ^ String.escaped s ^ "\""
    | BoolLit b -> if b then "true" else "false"
    | Binop(e1, op, e2) -> stringOfExpr e1 ^ " " ^ stringOfBinop op ^ " " ^ stringOfExpr e2
    | Unop(op, e) -> stringOfUnop op ^ stringOfExpr e
    | Id s -> s
    | Assign(s, e) -> s ^ " = " ^ stringOfExpr e
    | ArrayCreate(t, n, e) -> "Array " ^ stringOfVarType t ^ " " ^ n ^ " = " ^ " [" ^ stringOfExpr e ^
"]"
    | ArrayAccess(arrCreate, index) -> stringOfExpr arrCreate ^ " [" ^ stringOfExpr index ^ "]"
    | ArrayAssign(a,b,c) -> stringOfExpr a ^ " [" ^ stringOfExpr b ^ "] = " ^ stringOfExpr c
    | Call(f, e) -> f ^ "(" ^ String.concat ", " (List.map stringOfExpr e) ^ ")"
    | CallObject(o, f, e) -> o ^ "." ^ f ^ "(" ^ String.concat ", " (List.map stringOfExpr e) ^ ")"
    | ObjectAccess(o, v) -> o ^ "." ^ v

and stringOfStatement = function
    | Expr(expr) -> stringOfExpr expr ^ ";\n";
    | Return(expr) -> "return " ^ stringOfExpr expr ^ ";\n"
    | For(e1, e2, e3, s) -> "for (" ^ stringOfExpr e1  ^ " ; " ^ stringOfExpr e2 ^ " ; " ^ stringOfExpr
e3  ^ ") "
        ^ String.concat "" (List.map stringOfStatement s)
    | If(e, s1, s2) ->  "if (" ^ stringOfExpr e ^ ")\n"
        ^ String.concat "" (List.map stringOfStatement s1)
        ^ "else\n" ^ String.concat "" (List.map stringOfStatement s2)
    | While(e, s) -> "while (" ^ stringOfExpr e ^ ") " ^ String.concat "" (List.map stringOfStatement s)
    | Object o -> "Object " ^ o.objName ^ " = {\n"
        ^ String.concat "" (List.map stringOfFormal o.objLocals) ^ "};\n"

and stringOfFunction f =
    stringOfVarType f.returnType ^ " " ^ f.name ^ "(" ^
    String.concat ", " (List.map stringOfFormal f.args) ^ ")\n{\n" ^
    String.concat "" (List.map stringOfFormal f.locals) ^
    String.concat "" (List.map stringOfStatement f.body) ^ "}\n"

and stringOfProgram funcs =
    "\n\n" ^ String.concat "" (List.map stringOfFunction funcs) ^ "\n"

let convertBinOp = function
    | A.Add                 -> Add
    | A.Sub                 -> Sub
    | A.Mult                -> Mult
    | A.Div                 -> Div
    | A.Mod                 -> Mod
    | A.Equal               -> Equal
    | A.Neq                 -> Neq
    | A.LThan               -> LThan
    | A.Leq                 -> Leq
    | A.GThan               -> GThan
    | A.Geq                 -> Geq
    | A.And                 -> And
    | A.Or                  -> Or

let convertUnOp = function
    | A.Neg -> Neg
    | A.Not -> Not

let convertVarType = function
    | A.NullType            -> NullType
    | A.VoidType            -> VoidType
    | A.IntType             -> IntType
    | A.NumType             -> NumType
    | A.StringType          -> StringType
    | A.BoolType            -> BoolType

let rec getName map aux curName =
    if StringMap.mem curName map
    then (let aux = (StringMap.find curName map) ^ "." ^ aux in
        getName map aux (StringMap.find curName map))
    else aux

let rec convertExpr map = function
    | A.Null                    -> Null
    | A.Noexpr                  -> Noexpr
    | A.IntLit a                -> IntLit a
    | A.NumLit a                -> NumLit a
    | A.StringLit a             -> StringLit a
    | A.BoolLit a               -> BoolLit a
    | A.Binop (a, b, c)         -> Binop ((convertExpr map a), (convertBinOp b), (convertExpr map c))
    | A.Unop (a, b)             -> Unop ((convertUnOp a), (convertExpr map b))
    | A.Id a                    -> Id a
    | A.Assign (a, b)           -> Assign (a, (convertExpr map b))
    | A.ArrayCreate(a, b, c)    -> ArrayCreate((convertVarType a), (getName map b b), (convertExpr map
c))
    | A.ArrayAccess(a, b)       -> ArrayAccess((convertExpr map a), (convertExpr map b))
    | A.ArrayAssign(a, b, c)    -> ArrayAssign ((convertExpr map a), (convertExpr map b), (convertExpr
map c))
    | A.Call (a, b)             -> Call ((getName map a a), (convertExprs map b))
    | A.CallObject (a, b, c)    -> CallObject ((getName map a a), (getName map b b), (convertExprs map
c))
    | A.ObjectAccess(a, b)      -> ObjectAccess ((getName map a a), (getName map b b))

and convertExprs map = function
    | [] -> []
    | [ x ] -> [ convertExpr map x ]
    | (_ as l) -> List.map (convertExpr map) l

let convertFormal = function
    | A.Formal (v, s) -> Formal ((convertVarType v), s)

let buildFormals = function
    | [] -> []
    | [ x ] -> [ convertFormal x ]
    | (_ as l) -> List.map convertFormal l

let rec getFunctionBodyA = function
    | [] -> []
    | A.Function _ :: tl -> getFunctionBodyA tl
    | ((_ as x)) :: tl -> x :: (getFunctionBodyA tl)

let rec convertStatement map = function
    | A.Expr a -> Expr (convertExpr map a)
    | A.Return a -> Return (convertExpr map a)
    | A.For (e1, e2, e3, stls) ->
        For ((convertExpr map e1), (convertExpr map e2), (convertExpr map e3), (List.map
(convertStatement map) stls))
    | A.If (e, stls1, stls2) ->
        If ((convertExpr map e), (List.map (convertStatement map) stls1), (List.map (convertStatement
map) stls2))
    | A.While (e, stls) -> While ((convertExpr map e), (List.map (convertStatement map) stls))
    | _ -> Expr Noexpr

let rec getFunctionLocals = function
    | [] -> []
    | A.Variable (A.Local (t, n, _)) :: tl -> (Formal ((convertVarType t), n)) :: (getFunctionLocals tl)
    | ((_ as x)):: tl ->
        let findExpr = function
            | A.Expr e ->
                let findArrExprs = function
                    | A.ArrayCreate (t, n, _) ->
```

```
                        (Formal (ArrayType(convertVarType t), n)) :: (getFunctionLocals tl)
                    | _ -> getFunctionLocals tl
                in
                findArrExprs e
            | _ ->  getFunctionLocals tl
        in
        findExpr x

let rec getFunctionBodyS map = function
    | [] -> []
    | A.Variable (A.Local (_, name, v)) :: tl when v <> A.Noexpr ->
        (Expr (Assign (name, (convertExpr map v)))) :: (getFunctionBodyS map tl)
    | A.Variable (A.Local (_, _, v)) :: tl when v = A.Noexpr -> getFunctionBodyS map tl
    | ((_ as x)) :: tl ->
        let findExpr = function
            | A.Expr e ->
                let findArrExprs = function
                    | A.ArrayCreate (_, n, e) ->
                        (Expr(Assign(n, convertExpr map e))) :: (getFunctionBodyS map tl)
                    | _ -> (convertStatement map x) :: (getFunctionBodyS map tl)
                in
                let _ = findArrExprs e in
                (convertStatement map x) :: (getFunctionBodyS map tl)
            | _ -> (convertStatement map x) :: (getFunctionBodyS map tl)
        in
        findExpr x

let rec getFunctionsA = function
    | [] -> []
    | ((A.Function _ as x)) :: tl -> x :: (getFunctionsA tl)
    | _ :: tl -> getFunctionsA tl

let rec mapper parent map = function
    | [] -> map
    | A.Function { A.name = n; _ } :: tl ->
        mapper parent (StringMap.add n parent map) tl
    | _ -> map

let buildFunctionBody map = function
    | A.Function { A.name = n; A.body = b; _ } ->
        let curr = getFunctionsA b in
        let map = mapper n map curr in (curr, map)
    | _ -> ([], map)

let buildMethodBody map parent = function
    | A.Function { A.name = _; A.body = b; _ } ->
        let curr = getFunctionsA b in
        let map = mapper parent map curr in (curr, map)
    | _ -> ([], map)
```

```
let rec convertFunctionList map = function
    | [] -> []
    | A.Function { A.returnType = r; A.name = n; A.args = a; A.body = b } :: tl ->
        {
            returnType = convertVarType r;
            name = getName map n n;
            args = buildFormals a;
            body = getFunctionBodyS map b;
            locals = getFunctionLocals b;
            parent = if n = "main" then "main" else getName map (StringMap.find n map) (StringMap.find n
map);
        } :: (convertFunctionList map tl)
    | _ :: tl -> convertFunctionList map tl

let rec buildFunction map result = function
    | [] -> (List.rev result, map)
    | (A.Function { A.returnType = r; A.name = n; A.args = args; A.body = b } as a) :: tl ->
        let result1 = buildFunctionBody map a in
        let latterlist = tl @ (fst result1) in
        let map = snd result1 in
        let addedFunc = A.Function {
            A.returnType = r;
            A.name = n;
            A.args = args;
            A.body = getFunctionBodyA b;
        } in
        let result = result @ [ addedFunc ] in buildFunction map result latterlist
    | _ -> ([], map)

let rec convertObjects = function
    | [] -> []
    | A.Object o :: tl ->
        let func = A.Function {
            A.returnType = A.NumType;
            A.name = o.objName;
            A.args = [];
            A.body = convertObjects o.objStmts;
        } in
        func :: (convertObjects tl)
    | ((_ as x)) :: tl -> x :: (convertObjects tl)

let createMain stmts = A.Function
    {
        A.returnType = A.IntType;
        A.name = "main";
        A.args = [];
        A.body = convertObjects stmts;
    }

let convert stmts =
    let main = createMain stmts
    and funcMap = StringMap.empty in
        let finalList = buildFunction funcMap [] [ main ] in
            convertFunctionList (snd finalList) (fst finalList)
```

scanner.mll

```
(*
    Authors:
    Nathan Burgess
    Christina Charles
    Edvard Eriksson
    Alexandra Taylor
 *)

{
  open Parser
  let unescape s = Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)
}

let digit       = ['0'-'9']
let letter      = ['a'-'z' 'A'-'Z']
let variable    = (letter | ('_' letter)) (letter | digit | '_') *
let escape      = '\\' ['\\' ''' '"' 'n' 'r' 't']
let ascii       = ([' '-'!' '#'-'[' ']'-'~'])

rule token = parse
    | [' ' '\t' '\r' '\n']                 { token lexbuf }
    | "#:"                                 { commentMl lexbuf }
    | "#"                                  { comment lexbuf }
    | "+"                                  { PLUS }
    | "-"                                  { MINUS }
    | "*"                                  { TIMES }
    | "/"                                  { DIVIDE }
    | "%"                                  { MOD }
    | ";"                                  { SEMI }
    | ","                                  { COMMA }
    | "="                                  { ASSIGN }
    | ":"                                  { COLON }
    | "."                                  { DOT }
    | "and"                                { AND }
    | "&&"                                 { AND }
    | "or"                                 { OR }
    | "||"                                 { OR }
    | "not"                                { NOT }
    | "!"                                  { NOT }
    | "if"                                 { IF }
    | "else"                               { ELSE }
    | "for"                                { FOR }
    | "while"                              { WHILE}
    | "break"                              { BREAK }
    | "continue"                           { CONTINUE }
    | "in"                                 { IN }
    | "return"                             { RETURN }
    | ">"                                  { GT }
    | ">="                                 { GEQ }
    | "<"                               { LT }
    | "<="                              { LEQ }
    | "=="                              { EQUAL}
    | "is"                              { EQUAL}
    | "!="                              { NEQ }
    | "isnt"                            { NEQ }
    | "void"                            { VOID }
    | "num"                             { NUM }
    | "string"                          { STRING }
    | "bool"                            { BOOL }
    | "null"                            { NULL }
    | "Array"                           { ARRAY }
    | "Object"                          { OBJECT }
    | "Image"                           { IMAGE }
    | "Pixel"                           { PIXEL }
    | "Color"                           { COLOR }
    | "Console"                         { CONSOLE }

    | '"' ((ascii | escape)* as lit) '"'    { STRING_LITERAL(unescape lit) }
    | digit+'.'?digit* as lit           { NUM_LITERAL(float_of_string lit) }
    | "true" | "false" as boolLit       { BOOL_LITERAL(bool_of_string boolLit)}

    | "["                               { LSQUARE }
    | "]"                               { RSQUARE }
    | "{"                               { LCURL }
    | "}"                               { RCURL }
    | "("                               { LPAREN }
    | ")"                               { RPAREN }
    | variable as id                    { ID(id) }
    | eof                               { EOF }

and commentMl = parse
    | ":#"                              {token lexbuf}
    | _                                 {commentMl lexbuf}

and comment = parse
    | '\n'                              {token lexbuf}
    | _                                 {comment lexbuf}
```

parser.mly

```
/*
    Authors:
    Nathan Burgess
    Christina Charles
    Edvard Eriksson
    Alexandra Taylor

 */

%{ open Ast %}

%token PLUS MINUS TIMES DIVIDE MOD SEMI COMMA ASSIGN COLON DOT
%token GT GEQ LT LEQ EQUAL NEQ AND OR NOT IF ELSE FOR WHILE BREAK
%token CONTINUE IN RETURN LSQUARE RSQUARE LCURL RCURL LPAREN
%token RPAREN VOID NULL INT NUM STRING BOOL ARRAY OBJECT IMAGE
%token PIXEL COLOR CONSOLE EOF

/* Identifiers */
%token <string> ID

/* Literals */
%token <int> INT_LITERAL
%token <float> NUM_LITERAL
%token <bool> BOOL_LITERAL
%token <string> STRING_LITERAL

/* Order */
%right ASSIGN
%left AND OR
%left EQUAL NEQ
%left GT LT GEQ LEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT
%right LINK RIGHTLINK LEFTLINK AMPERSAND
%right LPAREN
%left  RPAREN
%right COLON
%right DOT

%start program
%type <Ast.program> program

%%

program: stmtList EOF                           { List.rev $1 }

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

stmt:
    | expr SEMI                             { Expr($1) }
    | varDecl SEMI                          { Variable($1) }
    | OBJECT objDecl SEMI                   { Object($2) }
    | funcDecl                              { Function($1) }
    | RETURN SEMI                           { Return(Noexpr) }
    | RETURN expr SEMI                      { Return($2) }
    | FOR LPAREN forExpr SEMI expr SEMI forExpr RPAREN LCURL stmtList RCURL
        { For($3, $5, $7, List.rev $10) }
    | IF LPAREN expr RPAREN LCURL stmtList RCURL ELSE LCURL stmtList RCURL
        { If($3,List.rev $6,List.rev $10) }
    | IF LPAREN expr RPAREN LCURL stmtList RCURL
        { If($3,List.rev $6,[]) }
    | WHILE LPAREN expr RPAREN LCURL stmtList RCURL
        { While($3, List.rev $6) }

objDecl:
    | ID stmtList                           { {
        objName         = $1;
        objStmts        = [] } }
    | ID ASSIGN LCURL stmtList RCURL        { {
        objName         = $1;
        objStmts        = List.rev $4 } }

varDecl:
    | varType ID                            { Local($1, $2, Noexpr) }
    | varType ID ASSIGN expr                { Local($1, $2, $4) }

varType:
    | NULL                                  { NullType }
    | VOID                                  { VoidType }
    | INT                                   { IntType }
    | NUM                                   { NumType }
    | STRING                                { StringType }
    | BOOL                                  { BoolType }

formalexprList:
    | /* nothing */                         { [] }
    | formal                                { [$1] }
    | formalexprList COMMA formal           { $3 :: $1 }

formal:
    | varType ID                            { Formal($1, $2) }

funcDecl:
    | varType ID LPAREN formalexprList RPAREN LCURL stmtList RCURL { {
        returnType = $1;
        name = $2;
        args = List.rev $4;
        body = List.rev $7 } }

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

expr:
    | literals                              { $1 }
    | NULL                                  { Null }
    | 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 EQUAL        expr                { Binop($1, Equal, $3) }
    | expr NEQ          expr                { Binop($1, Neq,   $3) }
    | expr LEQ          expr                { Binop($1, Leq,   $3) }
    | expr LT           expr                { Binop($1, LThan, $3) }
    | expr GT           expr                { Binop($1, GThan, $3) }
    | expr GEQ          expr                { Binop($1, Geq,   $3) }
    | expr AND          expr                { Binop($1, And,   $3) }
    | expr MOD          expr                { Binop($1, Mod,   $3) }
    | expr OR           expr                { Binop($1, Or,    $3) }
    | NOT               expr                { Unop(Not, $2) }
    | MINUS             expr                { Unop(Neg, $2) }
    | ID                                    { Id($1) }
    | ID ASSIGN expr                        { Assign($1, $3) }
    | LPAREN expr RPAREN                        { $2 }
    | ID LPAREN exprList RPAREN             { Call($1, List.rev $3) }
    | ID DOT ID LPAREN exprList RPAREN      { CallObject($1, $3, List.rev $5) }
    | ID DOT ID                             { ObjectAccess($1, $3) }
    | expr LSQUARE expr RSQUARE ASSIGN expr { ArrayAssign($1, $3, $6) }
    | ARRAY varType ID arrCreate            { ArrayCreate($2, $3, $4) }
    | expr arrAccess                        { ArrayAccess($1, $2) }

arrCreate:

    | ASSIGN LSQUARE expr RSQUARE           { $3 }

arrAccess:
    | LSQUARE expr RSQUARE                  { $2 }

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

literals:
        | NUM_LITERAL                           { NumLit($1) }
        | INT_LITERAL                           { IntLit($1) }
        | STRING_LITERAL                            { StringLit($1) }
        | BOOL_LITERAL                              { BoolLit($1) }
```

semant.ml

```
(*
   Authors:
   Nathan Burgess
   Edvard Eriksson
 *)

open Sast
open Printf

module StringMap = Map.Make(String)

exception SemanticError of string

(* error message functions *)
let undeclaredFunctionError name =
   let msg = sprintf "undeclared function %s" name
   in raise (SemanticError msg)

let duplicateFormalDeclError func name =
   let msg = sprintf "duplicate formal %s in %s" name func.name
   in raise (SemanticError msg)

let duplicateLocalDeclError func name =
   let msg = sprintf "duplicate local %s in %s" name func.name
   in raise (SemanticError msg)

let undeclaredIdentifierError name =
   let msg = sprintf "undeclared identifier %s" name
   in raise (SemanticError msg)

let illegalAssignmentError lvaluet rvaluet ex =
   let msg = sprintf "illegal assignment %s = %s in %s" lvaluet rvaluet ex
   in raise (SemanticError msg)

let illegalBinaryOperationError typ1 typ2 op ex =
   let msg = sprintf "illegal binary operator %s %s %s in %s" typ1 op typ2 ex
   in raise (SemanticError msg)

let illegalArrayAccessError typ1 typ2 ex =
   let msg = sprintf "illegal array access %s %s in %s" typ1 typ2 ex
   in raise (SemanticError msg)

let illegalUnaryOperationError typ op ex =
   let msg = sprintf "illegal unary operator %s %s in %s" op typ ex
   in raise (SemanticError msg)
```

```
let unmatchedFuncArgLenError name =
   let msg = sprintf "args length not match in function call: %s" name
   in raise (SemanticError msg)

let incompatibleFuncArgTypeError typ1 typ2 =
   let msg =
      sprintf "incompatible argument type %s, but %s is expected" typ1 typ2
   in raise (SemanticError msg)

let invalidExprAfterReturnError _ =
   let msg = sprintf "nothing may follow a return"
   in raise (SemanticError msg)

let redifinePrintFuncError _ =
   let msg = sprintf "function print may not be defined"
   in raise (SemanticError msg)

let duplicateFuncError name =
   let msg = sprintf "duplicate function declaration: %s" name
   in raise (SemanticError msg)

let unsupportedOperationError typ name =
   let msg = sprintf "unsupport operation on type %s: %s" typ name
   in raise (SemanticError msg)

let returnTypeMisMatchError typ1 typ2 =
   let msg = sprintf "wrong function return type: %s, expect %s" typ1 typ2
   in raise (SemanticError msg)

let rec isArrayType t = match t with | ArrayType _ -> true | _ -> false
and getArrayNesting t = match t with | ArrayType inner -> 1 + (getArrayNesting inner) | _ -> 1

let checkReturnType func typ =
   let lvaluet = func.returnType
   and rvaluet = typ in
      match lvaluet with
         | NumType when rvaluet = NumType -> ()
         | StringType when rvaluet = NullType -> ()
         | _ -> if lvaluet == rvaluet then ()
            else returnTypeMisMatchError (stringOfVarType rvaluet) (stringOfVarType lvaluet)

let getFunctionObject name func_map =
   try StringMap.find name func_map
   with | Not_found -> undeclaredFunctionError name

let reportDuplicateVar exceptf list =
   let rec helper =
      function
         | n1 :: n2 :: _ when n1 = n2 -> exceptf n1
      | _ :: t -> helper t
      | [] -> ()
  in helper (List.sort compare list)

let checkFunction func_map func =
  let args = List.map (function | Formal (_, n) -> n) func.args in
    (reportDuplicateVar (duplicateFormalDeclError func) args;

    let locals = List.map (function | Formal (_, n) -> n) func.locals in
      (reportDuplicateVar (duplicateLocalDeclError func) locals;

      let rec typeOfIdentifier func s =
        let symbols = List.fold_left (fun m -> function | Formal (t, n) -> StringMap.add n t m)
          StringMap.empty (func.args @ func.locals) in
          try StringMap.find s symbols with
            | Not_found ->
              if func.name = "main"
              then undeclaredIdentifierError s
              else typeOfIdentifier (StringMap.find func.parent func_map) s
      in

      let checkAssign lvaluet rvaluet ex = match lvaluet with
        | NumType when rvaluet = NumType -> lvaluet
        | StringType when rvaluet = NullType -> lvaluet
        | _ -> if lvaluet == rvaluet
          then lvaluet
          else illegalAssignmentError (stringOfVarType lvaluet) (stringOfVarType rvaluet) (stringOfExpr ex) in

      let rec expr = function
        | IntLit _     -> IntType
        | NumLit _     -> NumType
        | Null         -> NullType
        | StringLit _   -> StringType
        | BoolLit _    -> BoolType
        | (Binop (e1, op, e2) as e) -> let t1 = expr e1 and t2 = expr e2 in (match op with
          | Add | Sub | Mult | Div when (t1 = IntType) && (t2 = IntType) -> IntType
          | Add | Sub | Mult | Div when (t1 = NumType) && (t2 = NumType) -> NumType
          | Add | Sub | Mult | Div when (t1 = IntType) && (t2 = NumType) -> NumType
          | Add | Sub | Mult | Div when (t1 = NumType) && (t2 = IntType) -> NumType
          | Equal | Neq when t1 = t2 -> BoolType
          | LThan | Leq | GThan | Geq  when ((t1 = IntType) || (t1 = NumType)) && ((t2 = IntType) || (t2 = NumType)) ->
BoolType
          | And | Or when (t1 = BoolType) && (t2 = BoolType) -> BoolType
          | Mod when (t1 = IntType) && (t2 = IntType) -> IntType
          | _ -> illegalBinaryOperationError (stringOfVarType t1) (stringOfVarType t2) (stringOfBinop op)
(stringOfExpr e))
        | (Unop (op, e) as ex) -> let t = expr e in (match op with
          | Neg when t = IntType -> IntType
          | Neg when t = NumType -> NumType
          | Not when t = BoolType -> BoolType
      | _ -> illegalUnaryOperationError (stringOfVarType t) (stringOfUnop op) (stringOfExpr ex))
  | Id s -> typeOfIdentifier func s
  | (Assign (var, e) as ex) -> let lt = typeOfIdentifier func var and rt = expr e in checkAssign lt rt ex
  | ObjectAccess(_, _) -> NullType
  | CallObject(_, _, _) -> NullType
  | ArrayAssign(_, _, _) -> NullType
  | ArrayCreate(_, _, expr1) -> let e_type = expr expr1 in
      if e_type != IntType
      then (raise(Failure("Array dimensions must be type int")))
      else IntType
  | (ArrayAccess(expr1, expr2) as ex) -> let e_type = expr expr1 and e_num = expr expr2 in
      if (e_num != IntType)
        then illegalArrayAccessError (stringOfVarType e_type) (stringOfVarType e_num) (stringOfExpr ex)
      else
        (match e_type with (* add object to this when ready *)
          | IntType
          | NumType
          | StringType
          | BoolType
          | _ -> illegalArrayAccessError (stringOfVarType e_type) (stringOfVarType e_num) (stringOfExpr ex))
  | Noexpr -> VoidType
  | Call (n, args) -> let func_obj = getFunctionObject n func_map in
      let checkFunctionCall func args =
        let check_args_length l_arg r_arg =
          if (List.length l_arg) = (List.length r_arg)
          then ()
          else unmatchedFuncArgLenError func.name in
        (if List.mem func.name [ "printb"; "print"; "printf"; "string"; "float"; "int"; "bool" ] then ()
        else check_args_length func.args args;
        let check_args_type l_arg r_arg =
          List.iter2 (function | Formal (t, _) -> (fun r -> let r_typ = expr r in
            if t = r_typ
            then ()
            else incompatibleFuncArgTypeError (stringOfVarType r_typ) (stringOfVarType t)))
          l_arg r_arg in
            if List.mem func.name [ "printb"; "print"; "printf"; "string"; "float"; "int"; "bool" ]
            then ()
            else check_args_type func.args args)
        in
          (ignore (checkFunctionCall func_obj args);
          func_obj.returnType)
in

let rec stmt = function
    | Expr e    -> ignore (expr e)
    | Return e  -> ignore (checkReturnType func (expr e))
    | For (e1, e2, e3, stls) ->
        (
          ignore (expr e1);
          ignore (expr e2);
```

```
                    ignore (expr e3);
                    ignore (stmt_list stls)
                  )
                | If (e, stls1, stls2) -> (ignore e; ignore (stmt_list stls1); ignore (stmt_list stls2))
                | While (e, stls) -> (ignore e; ignore (stmt_list stls))
                | Object o -> (ignore o.objName; ignore (stmt_list o.objStmts))

              and stmt_list = function
                | Return _ :: ss when ss <> [] -> invalidExprAfterReturnError ss
                | s :: ss -> (stmt s; stmt_list ss)
                | [] -> ()
              in stmt_list func.body))

let check program =
  let m = StringMap.empty in
    (ignore (List.map (fun f ->
      if StringMap.mem f.name m
      then duplicateFuncError f.name
      else StringMap.add f.name true m) program));
  let builtInFuncs =
    let funcs = [ ("print", {
      returnType = VoidType;
      name = "print";
      args = [ Formal (StringType, "x") ];
      locals = [];
      body = [];
      parent = "main";
    });
    ("printf", {
       returnType = VoidType;
       name = "printf";
       args = [ Formal (StringType, "x") ];
       locals = [];
       body = [];
       parent = "main";
    })] in
    let add_func funcs m =
      List.fold_left (fun m (n, func) -> StringMap.add n func m) m funcs in
      add_func funcs StringMap.empty in
    let func_map = List.fold_left (fun m f -> StringMap.add f.name f m) builtInFuncs program in
    let checkFunction_wrapper func m = func m in
      List.iter (checkFunction_wrapper checkFunction func_map) program
```

codegen.ml

```
(*
    Authors:
    Nathan Burgess
    Edvard Eriksson
    Christina Charles
 *)

module L = Llvm
module S = Sast

module StringMap = Map.Make(String)

let context     = L.global_context ()
let llctx       = L.global_context ()

let utilsBuffer = L.MemoryBuffer.of_file "lib/utils.bc"
let llm         = Llvm_bitreader.parse_bitcode llctx utilsBuffer
let the_module  = L.create_module   context "PixMix"

let i32_t       = L.i32_type        context
and f_t         = L.double_type     context
and i8_t        = L.i8_type         context
and i1_t        = L.i1_type         context
and void_t      = L.void_type       context
and void_ptr_t  = L.pointer_type    (L.i8_type context)
and str_t       = L.pointer_type    (L.i8_type context)
and obj_t       = L.pointer_type    (L.i8_type context)

let rec getLTypeOfType = function
    | S.VoidType -> void_t
    | S.IntType -> i32_t
    | S.NumType -> f_t
    | S.BoolType -> i1_t
    | S.StringType -> str_t
    | S.ArrayType(typ) -> L.pointer_type (getLTypeOfType typ)
    | S.ObjectType -> obj_t
    | _ as t -> raise (Failure ("[Error] Could not find type \"" ^ S.stringOfVarType t ^ "\"."))

and getLConstOfType = function
    | S.IntType -> L.const_int i32_t 0
    | S.NumType -> L.const_int i32_t 1
    | S.BoolType -> L.const_int i32_t 2
    | S.StringType -> L.const_int i32_t 3
    | S.ArrayType(_) -> L.const_int i32_t 4
    | S.ObjectType -> L.const_int i32_t 5
    | _ as t -> raise (Failure ("[Error] Could not find type \"" ^ S.stringOfVarType t ^ "\"."))

let int_zero = L.const_int i32_t 0
and num_zero = L.const_float f_t 0.
and bool_false = L.const_int i1_t 0
and bool_true = L.const_int i1_t 1
and const_null = L.const_int i32_t 0
and str_null = L.const_null str_t
and object_null = L.const_null obj_t

let getNullForType = function
    | S.StringType -> str_null
    | S.NumType -> num_zero
    | S.ObjectType -> object_null
    | _ as x -> raise (Failure ("[Error] Could not get null type for " ^ S.stringOfVarType x ^ "."))

let getDefaultValue = function
    | (S.IntType as t) -> L.const_int (getLTypeOfType t) 0
    | (S.BoolType as t) -> L.const_int (getLTypeOfType t) 0
    | (S.NumType as t) -> L.const_float (getLTypeOfType t) 0.
    | t -> L.const_null (getLTypeOfType t)

(*
    Type casting
 *)
let intToFloat llbuilder v = L.build_sitofp v f_t "tmp" llbuilder
let floatToInt llbuilder v = L.build_fptosi v i32_t "tmp" llbuilder

let void_to_int_t = L.function_type i32_t [| L.pointer_type i8_t |]
let void_to_int_f = L.declare_function "VoidtoInt" void_to_int_t the_module
let void_to_int void_ptr llbuilder =
    let actuals = [| void_ptr |]
    in L.build_call void_to_int_f actuals "VoidtoInt" llbuilder

let void_to_float_t = L.function_type f_t [| L.pointer_type i8_t |]
let void_to_float_f = L.declare_function "VoidtoFloat" void_to_float_t the_module
let void_to_float void_ptr llbuilder =
    let actuals = [| void_ptr |]
    in L.build_call void_to_float_f actuals "VoidtoFloat" llbuilder

let void_to_bool_t = L.function_type i1_t [| L.pointer_type i8_t |]
let void_to_bool_f = L.declare_function "VoidtoBool" void_to_bool_t the_module
let void_to_bool void_ptr llbuilder =
    let actuals = [| void_ptr |]
    in L.build_call void_to_bool_f actuals "VoidtoBool" llbuilder

let void_to_string_t = L.function_type str_t [| L.pointer_type i8_t |]
let void_to_string_f = L.declare_function "VoidtoString" void_to_string_t the_module
let void_to_string void_ptr llbuilder =
    let actuals = [| void_ptr |]
    in L.build_call void_to_string_f actuals "VoidtoString" llbuilder

(*
    Built-in function declarations
 *)
let printf_t = L.var_arg_function_type i32_t [| str_t |]
let printf_func = L.declare_function "printf" printf_t the_module
let codegen_print llbuilder el = L.build_call printf_func (Array.of_list el) "printf" llbuilder

let print_bool_t = L.function_type i32_t [| i1_t |]
let print_bool_f = L.declare_function "printBool" print_bool_t the_module
let print_bool e llbuilder = L.build_call print_bool_f [| e |] "print_bool" llbuilder

let print_array_t = L.function_type i32_t [| L.pointer_type i8_t |]
let print_array_f = L.declare_function "printArray" print_array_t the_module
let print_array llbuilder el = L.build_call print_array_f (Array.of_list el) "print_array" llbuilder

let codegen_string_lit s llbuilder = L.build_global_stringptr s "str_tmp" llbuilder

let context_funcs_vars = Hashtbl.create 50

let print_hashtbl tb = print_endline (Hashtbl.fold (fun k _ m -> k ^ (", " ^ m)) tb "")

(*
    "Main" function of codegen, translates the program into it's LLVM IR equivalent
 *)
let translate program =
    let functionDecls =
        let funDecl m fdecl =
            let name = fdecl.S.name
            and formal_types =
                Array.of_list (List.map (function | S.Formal (t, _) -> getLTypeOfType t) fdecl.S.args)
in
            let ftype =
                L.var_arg_function_type (getLTypeOfType fdecl.S.returnType) formal_types
            in
            StringMap.add name ((L.define_function name ftype the_module), fdecl) m
        in
        List.fold_left funDecl StringMap.empty program
    in

    let buildFunctionBody fdecl =
        let getVariableName fName vName = fName^"."^vName in
        let (func, _) = StringMap.find fdecl.S.name functionDecls in
        let builder = L.builder_at_end context (L.entry_block func) in

        let _ =
        let addToContext locals = (ignore (Hashtbl.add context_funcs_vars fdecl.S.name locals); locals)
in
        let addFormal m = function
            | S.Formal (t, vName) -> (fun p ->
                let vName' = getVariableName fdecl.S.name vName in
                let local = L.define_global vName' (getDefaultValue t) the_module in
                (if L.is_null p
                    then ()
                    else ignore (L.build_store p local builder);
                    StringMap.add vName' (local, t) m
                )
            )
        in
        let addLocal m = function
            | S.Formal (t, vName) ->
                let vName' = getVariableName fdecl.S.name vName in
                let local_var = L.define_global vName' (getDefaultValue t) the_module in
                    StringMap.add vName' (local_var, t) m in
                let formals = List.fold_left2 addFormal StringMap.empty fdecl.S.args
                    (Array.to_list (L.params func))
        in
        addToContext (List.fold_left addLocal formals fdecl.S.locals)
    in

    let lookup vName =
        let getFunctionParentName fName =
            let (_, fdecl) = StringMap.find fName functionDecls
            in fdecl.S.parent
        in
        let rec findParent vName fName =
            try
                StringMap.find (getVariableName fName vName) (Hashtbl.find context_funcs_vars fName)
            with
                | Not_found ->
                    if fName = "main"
                    then raise (Failure ("[Error] Local variable " ^ (getVariableName fName vName) ^ "
not found. ("^fName^")"))
                    else findParent vName (getFunctionParentName fName)
        in findParent vName fdecl.S.name
    in

    (*
        Create the code for a binary operation
        e1 : the left-hand expression
        op : the operation
        e2 : the right-hand expression
        t  : the type that the operation should return
     *)
    let binop e1 op e2 t llbuilder =

        let floatBinops op e1 e2 = match op with
            | S.Add -> L.build_fadd e1 e2 "flt_addtmp" llbuilder
            | S.Sub -> L.build_fsub e1 e2 "flt_subtmp" llbuilder
            | S.Mult -> L.build_fmul e1 e2 "flt_multmp" llbuilder
            | S.Div -> L.build_fdiv e1 e2 "flt_divtmp" llbuilder
            | S.Mod -> L.build_frem e1 e2 "flt_fremtmp" llbuilder
            | S.Equal -> L.build_fcmp L.Fcmp.Oeq e1 e2 "flt_eqtmp" llbuilder
            | S.Neq -> L.build_fcmp L.Fcmp.One e1 e2 "flt_neqtmp" llbuilder
            | S.Leq -> L.build_fcmp L.Fcmp.Ole e1 e2 "flt_leqtmp" llbuilder
            | S.LThan -> L.build_fcmp L.Fcmp.Ult e1 e2 "flt_lesstmp" llbuilder
            | S.GThan -> L.build_fcmp L.Fcmp.Ogt e1 e2 "flt_sgttmp" llbuilder
            | S.Geq -> L.build_fcmp L.Fcmp.Oge e1 e2 "flt_sgetmp" llbuilder
            | _ -> raise (Failure "[Error] Unrecognized float binop opreation.")
        in
        let intBinops op e1 e2 = match op with
            | S.Add -> L.build_add e1 e2 "addtmp" llbuilder
            | S.Sub -> L.build_sub e1 e2 "subtmp" llbuilder
            | S.Mult -> L.build_mul e1 e2 "multmp" llbuilder
            | S.Div -> L.build_sdiv e1 e2 "divtmp" llbuilder
            | S.Mod -> L.build_srem e1 e2 "sremtmp" llbuilder
            | S.Equal -> L.build_icmp L.Icmp.Eq e1 e2 "eqtmp" llbuilder
            | S.Neq -> L.build_icmp L.Icmp.Ne e1 e2 "neqtmp" llbuilder
            | S.Leq -> L.build_icmp L.Icmp.Sle e1 e2 "leqtmp" llbuilder
            | S.LThan -> L.build_icmp L.Icmp.Slt e1 e2 "lesstmp" llbuilder
            | S.GThan -> L.build_icmp L.Icmp.Sgt e1 e2 "sgttmp" llbuilder
            | S.Geq -> L.build_icmp L.Icmp.Sge e1 e2 "sgetmp" llbuilder
            | S.And -> L.build_and e1 e2 "andtmp" llbuilder
            | S.Or -> L.build_or e1 e2 "ortmp" llbuilder
        in
        let types t = match t with
            | S.NumType -> floatBinops op e1 e2
            | S.BoolType | S.IntType -> intBinops op e1 e2
            | _ -> raise (Failure "[Error] Unrecognized binop data type.")
        in ((types t), (match op with
            | S.Add | S.Sub | S.Mult | S.Div | S.Mod -> t
            | _ -> S.BoolType))
    in
    let rec getExpr builder = function
        | S.IntLit i -> ((L.const_int i32_t i), S.IntType)
        | S.NumLit f -> ((L.const_float f_t f), S.NumType)
        | S.BoolLit b -> ((L.const_int i1_t (if b then 1 else 0)), S.BoolType)
        | S.StringLit s -> ((codegen_string_lit s builder), S.StringType)
        | S.Noexpr -> ((L.const_int i32_t 0), S.VoidType)
        | S.Null -> (const_null, S.NullType)
        | S.Id s -> let (var, typ) = lookup s in ((L.build_load var s builder), typ)
        | S.Binop (e1, op, e2) ->
            let (e1', t1) = getExpr builder e1
            and (e2', t2) = getExpr builder e2
            in (match (t1, t2) with
                | (_, S.NullType) -> (match op with
                    | S.Equal -> ((L.build_is_null e1' "isNull" builder), S.BoolType)
                    | S.Neq -> ((L.build_is_not_null e1' "isNull" builder), S.BoolType)
                    | _ -> raise (Failure "[Error] Unsupported Null Type Operation."))
                | (S.NullType, _) -> (match op with
                    | S.Equal -> ((L.build_is_null e2' "isNotNull" builder), S.BoolType)
                    | S.Neq -> ((L.build_is_not_null e2' "isNotNull" builder), S.BoolType)
                    | _ -> raise (Failure "[Error] Unsupported Null Type Operation."))
                | (t1, t2) when t1 = t2 -> binop e1' op e2' t1 builder
                | (S.IntType, S.NumType) ->
                    binop (intToFloat builder e1') op e2' S.NumType builder
                | (S.NumType, S.IntType) ->
                    binop e1' op (intToFloat builder e2') S.NumType builder
                | _ -> raise (Failure "[Error] Unsuported Binop Type."))
        | S.Unop (op, e) ->
            let (e', typ) = getExpr builder e
            in (((match op with
                | S.Neg -> if typ = S.IntType then L.build_neg else L.build_fneg
                | S.Not -> L.build_not) e' "tmp" builder
            ), typ)
        | S.Assign (s, e) ->
            let (e', etyp) = getExpr builder e in
            let (var, typ) = lookup s
            in ((match (etyp, typ) with
                | (t1, t2) when t1 = t2 -> (ignore (L.build_store e' var builder); e')
                | (S.NullType, _) -> (ignore (L.build_store (getNullForType typ) var builder);
                    getNullForType typ)
                | (S.IntType, S.NumType) ->
                    let e' = intToFloat builder e'
                    in (ignore (L.build_store e' var builder); e')
                | _ -> raise (Failure "[Error] Assign Type inconsist.")
            ), typ)
        | S.ArrayAccess(name, _) ->
            let (arr, _) = getExpr builder name in
            let index = L.const_int i32_t 1 in
            let ind = L.build_add index (L.const_int i32_t 1) "array_index" builder in
            let _val = L.build_gep arr [| ind |] "array_access" builder in
                (L.build_load _val "array_access_val" builder, S.IntType)

        | S.ArrayAssign(name, _, e) ->
            let (e', _) = getExpr builder e in
            let (var, typ) = lookup (S.stringOfExpr name) in
            (L.build_store e' var builder, typ)

        | S.ArrayCreate(typ, _, e) ->
            let t = getLTypeOfType typ in
            let (e', _) = getExpr builder e in
            let size = floatToInt builder e' in

            let size_t = L.build_intcast (L.size_of t) i32_t "tmp" builder in
            let size = L.build_mul size_t size "tmp" builder in
            let size_real = L.build_add size (L.const_int i32_t 1) "tmp" builder in

            let arr = L.build_array_malloc t size_real "tmp" builder in
            let arr = L.build_pointercast arr (L.pointer_type t) "tmp" builder in
            let arr_len_ptr = L.build_pointercast arr (L.pointer_type i32_t) "tmp" builder in
                ignore(L.build_store size_real arr_len_ptr builder);
            (arr, typ)
```

```
        | S.Call ("print", el) -> let print_expr e =
            let (eval, etyp) = getExpr builder e in (match etyp with
                | S.IntType -> ignore (codegen_print builder [ codegen_string_lit "%d\n" builder; eval
])
                | S.NullType -> ignore (codegen_print builder [ codegen_string_lit "null\n" builder ])
                | S.BoolType -> ignore (print_bool eval builder)
                | S.NumType -> ignore (codegen_print builder [ codegen_string_lit "%f\n" builder; eval
])
                | S.StringType -> ignore (codegen_print builder [ codegen_string_lit "%s\n" builder;
eval ])
                | S.ArrayType(t) -> (match t with
                    | S.IntType -> ignore (codegen_print builder [ codegen_string_lit "%d\n" builder;
eval ])
                    | S.NullType -> ignore (codegen_print builder [ codegen_string_lit "null\n" builder
])
                    | S.BoolType -> ignore (print_bool eval builder)
                    | S.NumType -> ignore (codegen_print builder [ codegen_string_lit "%f\n" builder;
eval ])
                    | S.StringType -> ignore (codegen_print builder [ codegen_string_lit "%s\n" builder;
eval ])
                    | _ -> raise (Failure "[Error] Unsupported type for print."))
                | _ -> raise (Failure "[Error] Unsupported type for print."))
            in (List.iter print_expr el; ((L.const_int i32_t 0), S.VoidType))
        | S.Call ("printf", el) ->
            ((codegen_print builder (List.map (fun e -> let (eval, _) = getExpr builder e in eval) el)),
S.VoidType)
        | S.Call (f, act) ->
            let (fdef, fdecl) = StringMap.find f functionDecls in
            let actuals = List.rev (List.map (fun e -> let (eval, _) = getExpr builder e in eval)
(List.rev act)) in
            let result = (match fdecl.S.returnType with
                | S.VoidType -> ""
                | _ -> f ^ "_result")
            in
            ((L.build_call fdef (Array.of_list actuals) result builder), (fdecl.S.returnType))
        | S.CallObject (_, f, act) ->
            let (fdef, fdecl) = StringMap.find f functionDecls in
            let actuals = List.rev (List.map (fun e -> let (eval, _) = getExpr builder e in eval)
(List.rev act)) in
            let result = (match fdecl.S.returnType with
                | S.VoidType -> ""
                | _ -> f ^ "_result")
            in
            ((L.build_call fdef (Array.of_list actuals) result builder), (fdecl.S.returnType))
        | S.ObjectAccess(_, _) ->
            (const_null, S.NullType)
        in

    let add_terminal builder f = match L.block_terminator (L.insertion_block builder) with
        | Some _ -> ()
        | None -> ignore (f builder) in

    let rec stmt builder = function
        | S.Expr e -> (ignore (getExpr builder e); builder)
        | S.Return e -> (ignore (let (ev, et) = getExpr builder e in
            match ((fdecl.S.returnType), et) with
                | (S.VoidType, _) -> L.build_ret_void builder
                | (t1, t2) when t1 = t2 -> L.build_ret ev builder
                | (S.NumType, S.IntType) ->
                    L.build_ret (intToFloat builder ev) builder
                | (t1, S.NullType) ->
                    L.build_ret (getDefaultValue t1) builder
                | _ -> raise (Failure "[Error] Return type doesn't match."));
            builder)
        | S.If (predicate, then_stmt, else_stmt) ->
            let (bool_val, _) = getExpr builder predicate in
            let merge_bb = L.append_block context "merge" func in
            let then_bb = L.append_block context "then" func in
                (add_terminal
                    (List.fold_left stmt (L.builder_at_end context then_bb) then_stmt)
                    (L.build_br merge_bb);
                    let else_bb = L.append_block context "else" func in
                        (add_terminal
                            (List.fold_left stmt (L.builder_at_end context else_bb) else_stmt)
                            (L.build_br merge_bb);
                            ignore (L.build_cond_br bool_val then_bb else_bb builder); L.builder_at_end
context merge_bb))
        | S.While (predicate, body) ->
            let pred_bb = L.append_block context "while" func in
                (ignore (L.build_br pred_bb builder);
                    let body_bb = L.append_block context "while_body" func in
                        (add_terminal (List.fold_left stmt (L.builder_at_end context body_bb) body)
(L.build_br pred_bb);
                            let pred_builder = L.builder_at_end context pred_bb in
                            let (bool_val, _) = getExpr pred_builder predicate in
                            let merge_bb = L.append_block context "merge" func in
                                (ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
L.builder_at_end context merge_bb)))
        | S.For (e1, e2, e3, body) -> List.fold_left stmt builder
            [ S.Expr e1; S.While (e2, (body @ [ S.Expr e3 ])) ]
        | S.Object _ -> builder
    in

    let builder = List.fold_left stmt builder fdecl.S.body in
    add_terminal builder (match fdecl.S.returnType with
        | S.VoidType -> L.build_ret_void
        | (S.IntType as t) -> L.build_ret (L.const_int (getLTypeOfType t) 0)
        | (S.BoolType as t) -> L.build_ret (L.const_int (getLTypeOfType t) 0)
        | (S.NumType as t) -> L.build_ret (L.const_float (getLTypeOfType t) 0.)
        | t -> L.build_ret (L.const_null (getLTypeOfType t)))
    in
    List.iter buildFunctionBody (List.rev program);

the_module
```

pixmix.ml

```
(*
   Authors:
   Nathan Burgess
 *)

module StringMap = Map.Make(String)

type action = Ast | Sast | LLVM_IR | Compile

let _ =
   let action = ref Compile in
   let set_action a () = action := a in
   let speclist = [
      ("-a", Arg.Unit (set_action Ast), "Print the AST");
      ("-s", Arg.Unit (set_action Sast), "Print the AST");
      ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
      ("-c", Arg.Unit (set_action Compile),
       "Check and print the generated LLVM IR (default)");
   ] in
   let usage_msg = "usage: ./pixmix.native [-a|-l|-c] [file.pm]" in
   let channel = ref stdin in
   Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;

   let lexbuf = Lexing.from_channel !channel in
   let ast = Parser.program Scanner.token lexbuf in
   let sast = Sast.convert ast in
      Semant.check sast;

   match !action with
      | Ast     -> print_string (Ast.stringOfProgram ast)
      | Sast    -> print_string (Sast.stringOfProgram sast)
      | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
      | Compile -> let m = Codegen.translate sast in
         Llvm_analysis.assert_valid_module m;
         print_string (Llvm.string_of_llmodule m)
```

utils.c

```
/*
 * Authors:
 * Nathan Burgess
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include <stdarg.h>
#include "utils.h"

int32_t printBool(bool a) {
    printf("%s (bool)\n", a ? "true" : "false");
    return 0;
}
```

utils.h

```
/*
 * Authors:
 * Nathan Burgess
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>

#ifndef __UTILS_H__
#define __UTILS_H__

#define INT 0
#define FLOAT 1
#define BOOL 2
#define STRING 3
#define IMAGE 4

int32_t printBool(bool a);

struct Image {
        int32_t a;
};

#endif
```

testall.sh

#!/bin/bash

/* Authors:

```
 Nathan Burgess
Alexandra Taylor */
clrClear='\033[0m'
clrBlue='\033[1;34m'
clrPurple='\033[1;35m'
clrGreen='\033[1;32m'
clrRed='\033[31;01m'
clrYellow='\033[33;01m'

PASS_FILES="tests_pass/*.pm"
FAIL_FILES="tests_fail/*.pm"
output_file=temp.out
exe_file=temp.exe
count=0

printf "\n${clrGreen}--==[ ${clrBlue}Running test suite... ${clrGreen}]==--${clrClear}\n\n"

runTest() {
  count=$((count + 1))
  name=$1
  if [ $2 == "pass" ]; then
    testName=${name/tests_pass\//}
  else
    testName=${name/tests_fail\//}
  fi

  ./pixmix.native $1 &> temp.ll

  clang -Wno-override-module lib/utils.bc temp.ll -o ${exe_file} -lm &>/dev/null

  if [ -e "$exe_file" ]; then
    ./${exe_file} > $output_file

    if [ "$(tail -1 $output_file)" == "finished" ] && [ $2 == "pass" ]; then
      printf "  %-30s ${clrGreen}PASS${clrClear}" ${testName} 1>&2
      if [ $((count % 2)) = 0 ]; then
        printf "\n"
      fi
    else
      printf "  %-30s ${clrRed}FAIL${clrClear}" ${testName} 1>&2
      if [ $((count % 2)) = 0 ]; then
        printf "\n"
      fi
    fi

    rm ${exe_file} &>/dev/null
    rm ${output_file} &>/dev/null
  else
    if [ $2 == "fail" ]; then
      printf "  %-30s ${clrGreen}PASS${clrClear}" ${testName} 1>&2
      if [ $((count % 2)) = 0 ]; then
        printf "\n"
      fi
    else
      printf "  %-30s ${clrRed}FAIL${clrClear}" ${testName} 1>&2
      if [ $((count % 2)) = 0 ]; then
        printf "\n"
      fi
    fi
  fi

  rm temp.ll >/dev/null
}

for input_file in $PASS_FILES; do
  runTest $input_file "pass"
done

if [ $((count % 2)) != 0 ]; then
  printf "\n"
fi

count=0
printf "\n"
for input_file in $FAIL_FILES; do
  runTest $input_file "fail"
done

exit 0
```

Makefile

```
=======__Filename: Makefile__========
# Authors:
# Nathan Burgess

OBJS = ast.cmx codegen.cmx parser.cmx pixmix.cmx sast.cmx scanner.cmx semant.cmx
UTILS_FILE = lib/utils.bc

# Color Definitions
clrClear = \033[0m
clrBlue  = \033[1;34m
clrPurple= \033[1;35m
clrGreen = \033[1;32m
clrRed   = \033[31;01m
clrYellow= \033[33;01m


default: pixmix.native

.PHONY : pixmix.native
pixmix.native :
	@rm ${UTILS_FILE} parser.ml parser.mli | true
	ocamlbuild -use-ocamlfind -pkgs
llvm,llvm.analysis,llvm.linker,llvm.bitreader,llvm.irreader -cflags -w,+a-4
pixmix.native;
	@make clibs

pixmix: $(OBJS)
	ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis $(OBJS) -o
pixmix

scanner.ml : scanner.mll
	ocamllex scanner.mll

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

.PHONY : clean
clean :
	ocamlbuild -clean | true
	rm -f pixmix.native
	rm -f ${UTILS_FILE}
	rm -f *.cmx *.cmi *.cmo *.cmx *.o* *.ll *.exe
	rm -f pixmix parser.ml parser.mli scanner.ml *.cmo *.cmi

# Generate a parser.output file with all states from the parser
.PHONY : debug
debug:
	@make
	OCAMLRUNPARAM='p' ocamlyacc -v parser.mly

# Run the testing suite
.PHONY : tests
tests :
	@make
	@./testall.sh

# Run just a single test on the file test.pm in the project root
.PHONY : test
test :
	@make
	@make runtest

.PHONY : runtest
runtest :
	@./pixmix.native test.pm > test.ll
	@clang -Wno-override-module ${UTILS_FILE} test.ll -o test.exe -lm
	@./test.exe
	@rm test.ll
	@rm test.exe

# Run the same test as the test rule, but print out the AST
.PHONY : testast
testast :
	@make
	@./pixmix.native test.pm -a

# Run the same test as the test rule, but print out the SAST
.PHONY : testsast
testsast :
	@make
	@./pixmix.native test.pm -s

# Run the same test as the test rule, but print out the LLVM IR code
.PHONY : testlli
```

```
testlli :
	@make
	@./pixmix.native test.pm

# Run the same test as the test rule, but print out the AST, SAST, and LLVM IR code
.PHONY : testasl
testasl :
	@make
	@echo "\n$(clrGreen)--==[ $(clrBlue)Printing the $(clrPurple)AST$(clrBlue)...
$(clrGreen)]==--$(clrClear)"
	@./pixmix.native test.pm -a
	@echo "\n$(clrGreen)--==[ $(clrBlue)Printing the $(clrPurple)SAST$(clrBlue)...
$(clrGreen)]==--$(clrClear)"
	@./pixmix.native test.pm -s
	@echo "\n$(clrGreen)--==[ $(clrBlue)Printing the $(clrPurple)LLVM
IR$(clrBlue)... $(clrGreen)]==--$(clrClear)"
	@./pixmix.native test.pm
	@echo "\n$(clrGreen)--==[ $(clrBlue)Printing $(clrPurple)output$(clrBlue) from
running the program... $(clrGreen)]==--$(clrClear)"
	@make runtest

# Run the same test as the test rule and print out the state transition table along
with it
.PHONY : debugtest
debugtest :
	@make
	@OCAMLRUNPARAM='p' make test

.PHONY : clibs
clibs :
	@clang -emit-llvm -o ${UTILS_FILE} -c lib/utils.c -Wno-varargs

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

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

%.cmx : %.ml
	ocamlfind ocamlopt -c -package llvm $<

# Generated by "ocamldep *.ml *.mli" after building scanner.ml and parser.ml
ast.cmo :
```

```
ast.cmx :
codegen.cmo : sast.cmo
codegen.cmx : sast.cmx
pixmix.cmo : semant.cmo sast.cmo codegen.cmo ast.cmo
pixmix.cmx : semant.cmx sast.cmx codegen.cmx ast.cmx
sast.cmo : ast.cmo
sast.cmx : ast.cmx
semant.cmo : sast.cmo
semant.cmx : sast.cmx
```

Tests

```
#author: Alexandra Taylor
=======__Filename: tests_fail/compare_fail.pm__========
num a = 0;
string d = "ten";
if ( a < 5) {
      print("less");
}
if ( d > 7) {
      print("more");
}
  if (a==0) {
      print("equal");
}
  if (a!=1)  {
      print("not equal");
}
if ( a <= 5) {
      print("lte");
}
if (d >= 10) {
      print("gte");
}
print("finished");

=======__Filename: tests_fail/assign_fail.pm__========
num a;
num b;
a = true;
b = false;
print(a);
print(b);
print("finished");

=======__Filename: tests_fail/assign1_fail.pm__========
bool a;
bool b;
a = true;
b = 3;
print("finished");

=======__Filename: tests_fail/arrDeclare_fail.pm__========
Array int a = [5];
print("finished");

=======__Filename: tests_fail/add1_fail.pm__========
float a;
float b;
num sum;
a = "hello";
b = "world";
sum = a + b;
print(sum);
print("finished");

=======__Filename: tests_fail/assign_fail.pm__========
num a;
num b;
a = true;
b = false;
print(a);
print(b);
print("finished");

=======__Filename: tests_fail/declare_fail.pm__========
num _1;
num _2;
_1 = 1;
_2 = 2;
print(_1);
print(_2);
print("finished");

=======__Filename: tests_fail/declareSimple1_fail.pm__========
float a;
print("finished");

=======__Filename: tests_fail/declareSimple_fail.pm__========
int a;
print("finished");

=======__Filename: tests_fail/fib_fail.pm__========
num fib(num n)
{
	num a = true;
	num b = 0;
	num temp;

	while (n >= 0) {
		temp = a;
		a = a + b;
		b = temp;
		n = n - 1;
	}

	return b;
}

print(fib(7));
print("finished");

=======__Filename: tests_fail/for_fail.pm__========
string i;
for (i = 0;  i < 7;  i = i + 1) {
	print(i);
}

print("finished");

=======__Filename: tests_fail/function1_fail.pm__========
num add(num x, num y)
{
	return x + y;
	num a;
	a = 5;
}
print(add(a+1, 3));
print("finished");

=======__Filename: tests_fail/function2_fail.pm__========
num a()
{
	num b;
	return a;
}
print(a();
print("finished");
```

```
=======__Filename: tests_fail/function_fail.pm__========
num add(num x, num y)
{
       return x + y;
       num: a;
       a = 5;
       print(add(a, 3));
}
print("finished");

=======__Filename: tests_fail/gcd1_fail.pm__========
num gcd (string x, num y)
{
    if(x == 0) {
        return 0;
    }

    while (y != 0) {
        if ( x > y) {
            x = x - y;
        }
        else {
            y = y - x;
        }
    }

    return x;
}
print("GCD Result: ", gcd(48, 18));
print("finished");

=======__Filename: tests_fail/gcd_fail.pm__========
num gcd (num x, num y)
{
       while ( x != y) {
              if ( x > y) {
                     x = x - y;
              } else {
                     y = y - x;
              }
       }
```

```
	return x;
}
print(gcd(2, 3));
print("finished");

=======__Filename: tests_fail/gcdRecursion_fail.pm__========
num gcd (num x, num y)
{
	if (y == 0) {
		return x;
	}
	return gcd(y, x%y);
	print("gcd");
}
print("GCD OUTPUT:", gcd(60, 36));

=======__Filename: tests_fail/helloWorld_fail.pm__========
string hw;
hw = "� ";
print(hw);
print("finished");

=======__Filename: tests_fail/ifElse_fail.pm__========
num a;
num b;
num c;
num x;
x = 0;
if (x < 1)
a = 2;
b = 3;
c = a + b;
else
{
	a = 4;
	b = 5;
	c = a + b;
}
print(c);
print("finished");

=======__Filename: tests_fail/if_fail.pm__========
num a;
num b;
num c;
num x;
x = 0;
if (x < 1) {
	a = 2;
	b = 3;
	c = a + b;
	if (x > 1)
	a = 4;
	b = 5;
	c = a + b;
}
print(c);
print("finished");

=======__Filename: tests_fail/is_fail.pm__========
bool x;
while (x is 5) {
	x = x + 1;
}
print("finished");

=======__Filename: tests_fail/isnt_fail.pm__========
bool x;
while (x isnt 5) {
	x = x + 1;
}
print("finished");

=======__Filename: tests_fail/main_fail.pm__========
int main {
	num a;
	num b;
}
return 0;

=======__Filename: tests_fail/math_fail.pm__========
Object Math = {
	num add(num a, num b) {
		return a + b;
	}
```

```
        string mod(num a, num b) {
                return a % b;
        }

        num pow(num a, num b) {
                num base = a;
                while(b > 1) {
                        a = a * base;
                        b = b - 1;
                }
                return 0;
        }
};

printf("4 + 1 = %f\n", Math.add(4, 1));
printf("10 % 6 = %f\n", Math.mod(10, 6));
printf("2^5 = %f\n", Math.pow(2, 5));
print("finished");

=======__Filename: tests_fail/mathOperators_fail.pm__========
num a = 1 + 1;
a = 2 - 1;
a = 2 * 3;
a = 9 / 4;
a = 8 / 4;
a = 9 % 4;
a = 8 % 4;
num b = true +1;
b = 1.2 - 1;
b = 1 - 1.2;
b = 2 * 0.4;
b = 9. / 4;
b = -8;
b = -2.1;
b = -1;
b = -2.1;
print("finished");

=======__Filename: tests_fail/modulo_fail.pm__========
string a;
a = 5 % 3;
print(a);
print("finished");
```

```
=======__Filename: tests_fail/nested_fail.pm__========
num nested (num x, num y)
{
	while ( x != y) {
		if ( x > y) {
			x = x + y;
			if ( y < x) {
				x = x + y;
			} else {
				x = y - x;
			}
		}
		return x;
	}
}
print(nested(4, 3));
print("finished");

=======__Filename: tests_fail/newline_fail.pm__========
string a;
a = 5;
printf("%f\n", a);
print("finished");

=======__Filename: tests_fail/not_fail.pm__========
bool t = 1;
if (not t){
    print(false);
}
else {
    print(true);
}
print("finished");

=======__Filename: tests_fail/numAsDec_fail.pm__========
num b = "";
b = 1.2 - 1;
b = 1 - 1.2;
b = 2 * 0.4;
b = 9. / 4;
b = -8;
```

```
b = -2.1;
b = -1;
b = -2.1;
print("finished");

=======__Filename: tests_fail/numPrintsDec_fail.pm__========
float a;
a = 5.2;
printf("%f\n", a);
print("finished");

=======__Filename: tests_fail/objAssign1_fail.pm__========
Object myObj = 5;
print("finished");

=======__Filename: tests_fail/object1_fail.pm__========
object myObj;
print("finished");

=======__Filename: tests_fail/object2_fail.pm__========
Object myObj = {
    int a;
    num b = 0;
};
print("finished");

=======__Filename: tests_fail/object3_fail.pm__========
Object myObj = {
    int a = 5;

    num foo(num b, num c) {
        return b * c;
    }
};
print("finished");

=======__Filename: tests_fail/printf_fail.pm__========
num a;
a = 5;
printf(a);
print("finished");

=======__Filename: tests_fail/printString_fail.pm__========
string a;
 = "� ";
print(a);
print("finished");

=======__Filename: tests_fail/sub_fail.pm__========
string a;
string b;
num sum;
a = "hello";
b = "world";
sum = a - b;
print(sum);
print("finished");

=======__Filename: tests_fail/while_fail.pm__========
string hw;
hw = "Hello World!";
{
    num x = 1;
    while (x < 7)
        print(hw);
        x = x + 1;
}
print("finished");

=======__Filename: tests_pass/add1_pass.pm__========
num a;
num b;
num sum;
a = 2;
b = 3;
sum = a + b;
print(sum);
print("finished");

=======__Filename: tests_pass/add_pass.pm__========
num a;
num b;
num sum;
a = 2;
b = 3;
sum = a + b;
```

```
print(sum);
print("finished");

=======__Filename: tests_pass/arrDeclare_pass.pm__========
Array num a = [5];
print("finished");

=======__Filename: tests_pass/assign1_pass.pm__========
num a;
num b;
a = 1;
b = 2;
print(a);
print(b);
print("finished");

=======__Filename: tests_pass/assign_pass.pm__========
string a;
string b;
a = "one";
b = "two";
print(a);
print(b);
print("finished");

=======__Filename: tests_pass/comment_pass.pm__========
num a; # Single line comment
#:
    Multi-line comment
    num a;
:#
print("finished");

=======__Filename: tests_pass/compare_pass.pm__========
num a = 0;
num d = 10;

if (a < 5) {
      print("less");
}
if (d > 7) {
      print("more");
}
if (a == 0) {
	print("equal");
}
if (a != 1) {
	print("not equal");
}
if (a <= 5) {
	print("lte");
}
if (d >= 10) {
	print("gte");
}
print("finished");

=======__Filename: tests_pass/declare_pass.pm__========
num _a;
num _b;
_a = 1;
_b = 2;
print(_a);
print(_b);
print("finished");

=======__Filename: tests_pass/declareSimple_pass.pm__========
num a;
print("finished");

=======__Filename: tests_pass/fib_pass.pm__========
num fib(num n) {
    num a = 1;
    num b = 0;
    num temp;

    while (n >= 0) {
        temp = a;
        a = a + b;
        b = temp;
        n = n - 1;
    }
    return b;
}
```

```
print(fib(7));
print("finished");

=======__Filename: tests_pass/for_pass.pm__========
num i;
for (i = 0;  i < 7;  i = i + 1) {
    print(i);
}

print("finished");

=======__Filename: tests_pass/function1_pass.pm__========
num add(num x, num y)
{
      return x + y;
}

num a;
a = 5;
print(add(a+1, 3));
print("finished");

=======__Filename: tests_pass/function2_pass.pm__========
num a()
{
      num b;
      return b;
}

print(a());
print("finished");

=======__Filename: tests_pass/function_pass.pm__========
num add(num x, num y)
{
      return x + y;
}

num a;
a = 5;
print(add(a, 3));
print("finished");
```

```
=======__Filename: tests_pass/gcd1_pass.pm__========
num gcd (num x, num y)
{
    if(y == 0) {
        return x;
    }

    return gcd(y, (x % y));
}

print(gcd(240, 150));
print ("finished");

=======__Filename: tests_pass/gcd_pass.pm__========
num gcd (num x, num y)
{
    if(x == 0) {
        return y;
    }

    while (y != 0) {
        if (x > y) {
            x = x - y;
        }
        else {
            y = y - x;
        }
    }

    return x;
}

print("GCD Result: ", gcd(48, 18));
print("finished");

=======__Filename: tests_pass/gcdRec_pass.pm__========
num gcd (num x, num y) {
      if(y == 0) {
             return x;
      }
      return gcd(y, (x % y));
}
```

```
print("GCD Result: ", gcd(240, 150));
print("finished");

=======__Filename: tests_pass/helloWorld_pass.pm__========
string hw;
hw = "Hello World!";
print(hw);
print("finished");

=======__Filename: tests_pass/ifElse_pass.pm__========
num a;
num b;
num c;
num x;
x = 0;

if (x < 1) {
	a = 2;
	b = 3;
	c = a + b;
}
else {
	a = 4;
	b = 5;
	c = a + b;
}

print(c);
print("finished");

=======__Filename: tests_pass/if_pass.pm__========
num a;
num b;
num c;
num x;
x = 0;

if (x < 1) {
	a = 2;
	b = 3;
	c = a + b;
}
```

```
if (x > 1){
        a = 4;
        b = 5;
        c = a + b;
}

print(c);
print("finished");

=======__Filename: tests_pass/isnt_pass.pm__========
num x;
while (x isnt 5) {
    x = x + 1;
}
print("finished");

=======__Filename: tests_pass/is_pass.pm__========
num x;
while (x is 5) {
    x = x + 1;
}
print("finished");

=======__Filename: tests_pass/load_pass.pm__========
num a = 3227;
num b = 555;
num c = a / b;
print(c);
print("finished");

=======__Filename: tests_pass/mathOperators_pass.pm__========
num a = 1 + 1;
a = 2 - 1;
a = 2 * 3;
a = 9 / 4;
a = 8 / 4;
a = 9 % 4;
a = 8 % 4;
print("finished");

=======__Filename: tests_pass/math_pass.pm__========
Object Math = {
        num add(num a, num b) {
                return a + b;
        }

        num mod(num a, num b) {
                return a % b;
        }

        num pow(num a, num b) {
                num base = a;
                while(b > 1) {
                        a = a * base;
                        b = b - 1;
                }
                return a;
        }
};

printf("4 + 1 = %f\n", Math.add(4, 1));
printf("10 % 6 = %f\n", Math.mod(10, 6));
printf("2^5 = %f\n", Math.pow(2, 5));
print("finished");

=======__Filename: tests_pass/modulo_pass.pm__========
num a;
a = 12%10;
print(a);
print("finished");

=======__Filename: tests_pass/nested_pass.pm__========
num nested (num x, num y){
        while ( x != y) {
                if ( x > y) { x = x + y;}
                if ( y < x) { x = x + y;}
                else {
                        x = y - x;
                }
        return x;
        }
}

print(nested(4, 3));
print("finished");
```

```
=======__Filename: tests_pass/nestedPrint_pass.pm__========
num nested (num x, num y) {
      while ( x != y) { print("past while");
            if ( x > y) { print("past outer if"); x = x + y;}
            if ( y < x) { print("past inner if"); x = x - y;}
            else {
                  print("past else");
                  x = y - x;
            }
      Return x;
      }
}

print(nested(4, 3));
print("finished");

=======__Filename: tests_pass/newline_pass.pm__========
num a;
a = 5;
printf("%f\n", a);
print("finished");

=======__Filename: tests_pass/not_pass.pm__========
bool t = true;
if (not t){
    print("false");
}
else {
    print("true");
}
print("finished");

=======__Filename: tests_pass/numAsDec_pass.pm__========
num b = 1.2 +1;
b = 1.2 - 1;
b = 1 - 1.2;
b = 2 * 0.4;
b = 9. / 4;
b = -8;
b = -2.1;
b = -1;
b = -2.1;
print("finished");
```

```
=======__Filename: tests_pass/numPrintsDec_pass.pm__========
num a;
a = 5.2;
printf("%f\n", a);
print("finished");

=======__Filename: tests_pass/object1_pass.pm__========
Object myObj;
print("finished");

=======__Filename: tests_pass/object2_pass.pm__========
Object myObj = {
    num a;
    num b = 0;
};

print("finished");

=======__Filename: tests_pass/object3_pass.pm__========
Object myObj = {
    num a = 5;

    num foo(num b, num c) {
        return b * c;
    }
};

print("finished");

=======__Filename: tests_pass/printf_pass.pm__========
num a;
a = 5;
printf("%f\n", a);
print("finished");

=======__Filename: tests_pass/print_string_pass.pm__========
string a;
a = "hi";
print( a );
print("finished");

=======__Filename: tests_pass/recFunction_pass.pm__========
num recursion(num r) {
	if (r == 0) {
		return 1;
	}
	else {
		return r * recursion(r-1);
	}
}
print("finished");

=======__Filename: tests_pass/sub_pass.pm__========
num a;
num b;
num sum;
a = 2;
b = 3;
sum = a - b;
print(sum);
print("finished");

=======__Filename: tests_pass/while_pass.pm__========
string hw;
hw = "Hello World!";
num x = 1;
while (x < 7) {
	print(hw);
	 x = x + 1;
}
print("finished");
```