MatchaScript

An Imperative Compiled Programming Language

Columbia University
COMS W4115: Programming Languages and Translators - Spring 2017

Kimberly Hou	kjh2146
Rebecca Mahany	rlm2175
Jorge Orbay	jao2154
Rachel Yang	ry2277

Contents

1 Introduction

MatchaScript was inspired by JavaScript and TypeScript. Unlike JavaScript, MatchaScript is statically typed . The syntax of MatchaScript can be described as "JavaScript, but with type specifications." The output of the MatchaScript compiler is LLVM IR, a cross-platform intermediate representation language. The GitHub repository for MatchaScript can be found at this link: https://github.com/RebeccaMahany/MatchaScript.

1.1 Background

Syntactically, MatchaScript is based on JavaScript. JavaScript is a high-level, dynamic, untyped interpreted programming language that can be used for imperative, object-oriented, and functional programming styles. It is primarily used as a client-side scripting language interpreted by web browsers, but is also used for server-side scripting (Node.js) and other applications.

Criticism of JavaScript includes:

- JavaScript allows use of undefined variables, which become automatic global variables.
- JavaScript's automatic type conversions can result in difficult-to-detect and counterintuitive bugs at runtime, such as:

```
// https://wiki.theory.org/YourLanguageSucks#JavaScript_sucks_because
var i = 1;
i = i + ""; // converts i to the string "1"
i + 1;      // evaluates to the string "11"
```

- JavaScript hoists variable and function declarations to the top of the scope in which it is defined.

Over the years, in order to make JavaScript development easier and improve the language, developers created the following:

- JavaScript subsets containing the language's "good parts":
    - "JavaScript: The Good Parts" - Douglas Crockford
        - Crockford describes JavaScript's "very good ideas" as including "functions, loose typing, dynamic objects, and an expressive object literal notation."
        - see Appendix D: Syntax Diagrams
    - Strict Mode
        - A restricted variant of JavaScript that changes some silent errors into throw errors
        - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode
- Tools for JavaScript developers:
    - JSLint
        - Static code analysis tool that detects error and potential problems in JS code
    - Flow
        - "Flow adds static typing to JavaScript to improve developer productivity and code quality. In particular, static typing offers benefits like early error checking, which helps you avoid certain kinds of runtime failures, and code intelligence, which aids code maintenance, navigation, transformation, and optimization." (https://code.facebook.com/posts/1505962329687926/flow-a-new-static-type-checker-for-javascript/)
- JavaScript-based Programming Languages:
    - JSJS

        - JSJS is a strongly typed functional language that transcompiles to JavaScript.
        - http://www.cs.columbia.edu/~sedwards/classes/2016/4115-spring/reports/JSJS.pdf
    - TypeScript:
        - Strict superset of JavaScript
        - Adds optional static typing, class-based object-oriented programming
        - Transcompiles to JavaScript
        - MatchaScript's features are similar to TypeScript, although we use a different static typing syntax

We created MatchaScript with the work of the JavaScript community in mind.

1.2 MatchaScript Features

- Static typing
- Imperative programming
- Ability to import any number of external MatchaScript programs into your program
- No global variables

2 Language Tutorial

2.1 Environment Setup

Ensure that OCaml 4.01.0 and LLVM.3.6 are installed.

2.2 Using the Compiler

Assuming use of the terminal, navigate into the MatchaScript directory. Inside the MatchaScript project directory, first type **make** to execute the Makefile. All human-readable MatchaScript program text files are .ms files. To compile a .ms file, type the following:

```
$  ./MatchaScript.native -c  < filename.ms > filename.ll
$  lli filename.ll
```

To run the MatchaScript tests, you can simply run the testall.sh shell script.

```
$  ./testall.sh
```

2.3 Simple Example

```
function void hello_world(int a,int b) {
    int c = a+b;
    if (c>1)
    {
        print(“Hello World!”);
    }
}
hello(1,1);
```

hello_world.ms

The above is a very simple first program that adds to arguments and prints “Hello World!” if the sum is greater than one.

3 Language Reference Manual

3.1 Program Structure

A MatchaScript program (file) consists of a list of #include's and a list of statements, where statements can include functions, variable declarations, and other blocks of statements. Statements may span one or multiple lines. Similar to other scripting languages, a MatchaScript program does not require a function declaration in order to run. The following three examples demonstrate this as well as give a basic template for MatchaScript syntax:

```
function void hello() {
    print("Hello World!");
}
hello();
```

hello.ms

```
print("Hello World!");
```

hello.ms (equivalent and alternate version)

```
function void hello() {
    print("Hello World!");
}
print("Hello there!");
```

hello-there.ms

```
$  ./MatchaScript.native -c  < hello.ms > hello.ll
$  lli hello.ll
Hello World!
$
$  ./MatchaScript.native -c  < hello-there.ms > hello-there.ll
$  lli hello-there.ll
Hello there!
$
```

The examples above demonstrate a few important components about MatchaScript programs:

- If functions are included in the file, then the function(s) must be called or else nothing will print when the program is compiled and executed.
- Whatever statements included in the outermost scope (that is, outside of any function) will be executed.
- The built-in function **print** will print a single argument and then a newline character. The single argument must be of type int, float, or string only.

3.2 Lexical Structure

Tokens in MatchaScript include identifiers, keywords, constants, literals, operators, and separators (e.g. braces). Tokens are separated by whitespace (blanks, tabs, and newlines) or comments.

MatchaScript supports single-line comments that begin with // and multi-line comments, which begin with /* and terminate with */.

Identifiers are sequences of letters and, optionally, one or more digits or underscores. Identifiers must begin with a letter. Camelcase is suggested.

The following are reserved keywords in MatchaScript:

- function, fun, print
- void, int, float, char, string, bool, void, true, false
- class, constructor, this, new
- if, else, for, while, return

Unlike in JavaScript, semicolons are mandatory in MatchaScript at the end of statements.

MatchaScript files may not name any function "main."

3.3 Types, Values, and Variables

Types

Below are MatchaScript's data types and their syntax usage in programs:

`integer`	`int thirty = 30;`
`32-bit floating point number`	`float five = 5.0;`
`string`	`string s = "Welcome!";`
`boolean`	`bool b = true;`

`void`	`function void main() {`
	`// Only used as a return type`
	`}`

`fun`	`fun Motto = function void () {`
	` // Used for anonymous functions and currying`
	`}`

Variable Declaration

Variables in MatchaScript may be declared and instantiated on the same line or on separate lines:

`int lifeTheUniverseAndEverything;`
`lifeTheUniverseAndEverything = 42;`
	`int lifeTheUniverseAndEverything = 42;`

Note that each variable must be declared with its type and cannot be strung together on the same line. For instance, **`int a,b,c;`** will result in a parsing error. Therefore, while every variable can be declared and then initialized on separate lines, they have to be individually declared with their corresponding type.

Variable Scope

Variables are valid within the function in which they are defined. If they are in the outermost scope (and not within an explicitly defined function), then they are valid within that outermost scope from the time they are declared.

3.4 Expressions and Operators

Primary Expressions

MatchaScript's compiler categorizes primary expressions as literals (int, float, bool, string), identifiers, function expressions, binary operators, unary operators, and function calls. In addition, the following set is used to further delineate expressions:

`function`	**The `function` keyword defines a function declaration or a function expression (anonymous function).**
`{ }`	**Braces define a block of statements such as the body of a function or the body of an if-else statement.**
`( )`	**Parentheses are used as a grouping operator.**

Function Invocation

After defining them, functions are called with zero or more arguments. The number and type of arguments passed in at call time must match the number and type of formal parameters specified during function definition. As an example, refer to hello.ms.

Operators

The following are supported operators in MatchaScript:

- Arithmetic operators: +, -, *, /, %
- Relational operators: >, <, >=, <=, ==, !=
- Logical operators: &&, ||, !
- Assignment operators: =

Arithmetic Expressions

MatchaScript supports addition, subtraction, multiplication, division, and the modulo operator. In addition, the standard library supports a number of other math-related functions as well (see section 3.10). The results of all these expressions evaluate to some number (a float or an int).

Relational Expressions

MatchaScript includes all standard relational operators. Unlike in JavaScript, MatchaScript does not have two kinds of equality/inequality. We have implemented only strict equality/inequality (=== and !== in JavaScript) but have chosen to represent such relations using the traditional == and !== for ease of use. The result of a relational expression evaluates to the boolean value True or False.

Note that relational operators may only be used with operands of exactly the same type. For instance, an int cannot be compared with a float.

Logical Expressions

There are three logical operators: &&, ||, and !. The result of a logical expression evaluates to the boolean value true or false.

```
function void testBoolean(bool x, bool y) {
  if (x == y) {
    print("X is equal to Y");
  } else {
    print("X is not equal to Y");
  }
  bool z = true;
  if (x && z) {
    print("X and Z are both true");
  } else if (x || z) {
    print("X or Z is true");
  } else {
    print("X and Z are both false");
  }
}
testBoolean(true, false);
```

Assignment Expressions

MatchaScript uses a single assignment operator. The type declared with the variable and the right operand that the variable is being assigned to must be of equal data types.

```
int myAge = 21;
bool t = true;
```

3.5 Statements

Statements in MatchaScript are defined as one of the following:

Compound Statements (Blocks)

Blocks are a list of statements between a left brace and a right brace. Note that blocks are not allowed to be empty. MatchaScript recursively parses every statement found in a block.

Expression Statements

Every expression as defined in section 3.4 (Expressions) above is a type of statement.

Declaration Statements

Variable Declaration

As explained above in section 3.3 (Types, Values, and Variables), variables are represented by variable declarations. The grammar for a variable declaration (also referred to as vdecl in MatchaScript's compiler) is a tuple of three elements:

typ string expr

where typ is one of int, float, bool, char, fun, or string; string is the name of the variable declaration; and expr is an expression (in most cases, a literal).

Function Declaration

Function declarations (referred to as fdecls in MatchaScript) are a type with the following fields:

return type : typ;
name : string;
formals : bind list;
body : stmt list;

where typ can be one of void, int, float, bool, char, fun, or string; the name is the function name; formals is a list of binds, a bind being a typ and a string tuple; and a body, which is itself a list of statements.

Conditionals

If-else statements themselves contain statements. For instance, an IF statement would consist of an expression, a statement, and then another statement to correspond with the predicate, the then body, and the else body.

```
int x = 3;
int y = 4;
if (x <= y) {
 print("3<=4");
} else {
 print("3>4");
}
```

Loops

MatchaScript supports for loops, while loops, and do-while loops. The latter functions exactly as while loops do, except that they always execute at least once, regardless of how the predicate evaluates. When using for loops, never declare a variable within the arguments. Always use a variable already declared.

An example of a do-while loop is below:

```
int i = 1;
do {
    print("I should be printed once!")
}
while ( i < 1 );
```

An example of a while loop is below:

```
function int whileLoop(int num) {
  int i = 0;
  if (num <= 0) {
    print("Next time run this program with a number higher than zero.");
    return -1;
  } else {
    while (i < num) {
      print(i);
      i = i + 1;
    }
  }
  return 0;
}
whileLoop(5);
```

```
function int forLoop_test(int num) {
  int i = 0;
  if (num <= 0) {
    print("Next time run this program with a number higher than zero.");
    return -1;
  } else {
    for (i=0; i < num; i++) {
      print(i);
    }
  }
  return 0;
}
forLoop_test(5);
```

3.6 Functions

Defining Functions

Function Declarations

Functions are declared in the following manner:

```
function return_type name ([param]...) {
  // function body here
}
```

where [param] consists of a type and an identifier of the formal parameter. There may be zero or more parameters defined for a function. Functions may not be called before they are defined. The specific grammar components of function declarations are explained in section 3.5.

Function Expressions

Function expressions, also known as anonymous functions, are explained in section 3.4.

Invoking Functions

MatchaScript supports method invocation rather than constructor invocation. This means that once functions are defined, they may be called using the following paradigm:

```
name([arg]...);
```

where the name is the name of the function, and the arguments are either literals or identifiers of previously defined variables. At call time, the name, type, and order of arguments must match those of the formal parameters.

3.7 Include Statements

MatchaScript allows you to include other MatchaScript programs using the following syntax:
`#include "<filename>"`
All of the statements in the included program are prepended to the statements of the toplevel program. The same thing will happen recursively for the include statements in the included program.

3.8 Standard Library Functions

MatchaScript's standard library provides basic math functions like rounding floats, getting the ceiling of a float, etc. The functions are listed below:

- floor:
- round
- ceil
- float_pow: with two float arguments a and b, get a ^b
- int_pow: with two int arguments
- int_min: get the minimum of two ints
- int_max: get the maximum of two ints
- floast_abs: get the absolute value of a float
- int_abs: get the absolute value of an int

These functions can be included into any MatchaScript file using `#include "include/stdlib.ms";`

4 Project Planning

4.1 Planning Process

In order to organize our project, Rachel created a shared Google Drive folder in which team members could share research, take notes on our meetings, and create TODO lists. Rebecca created a shared GitHub repository for MatchaScript and the team created guidelines for their workflow (working in branches, requesting reviewers before committing to master, etc).

The team scheduled a two-hour block each week to meet in order to discuss short-term goals and to delegate work. As the semester progressed, the team also began to meet regularly in order to work on coding together.

4.2 Specification Process

Rachel proposed the idea for "JavaScript but better" in January, having come across articles criticizing the language over winter break. To understand JavaScript, before the Project Proposal deadline, the group read the JavaScript Language Reference Manual in David Flanagan's *JavaScript: The Definitive Guide.* We also browsed blogs, JavaScript-related tools and languages (e.g. Flow, TypeScript), and Douglass Crockford's *JavaScript: The Good Parts* to see what were common complaints about JavaScript.

At the beginning of the semester, we thought we could create a just-in-time-compiled scripting language that could be integrated into an open-source browser that we would modify. After discussing with Professor Edwards, we decided that goal was too ambitious, and largely outside the scope of the class. Our project proposal shows that we decided to implement a compiled, statically typed, multi-paradigm language.

4.3 Development Process

Hello World was our first milestone.

After Hello World, we began to iteratively add features to MatchaScript. Our main priorities were adding nested functions and currying, and adding classes and objects. However, we also added smaller features like more types, additional operators, new kinds of control flow, and function expressions. Our team focused on having a strong and function semantic analyzer.

We also met with Daniel Echikson (our assigned TA), other class TAs, and Professor Edwards during our development process in order to best understand how to implement trickier features.

Finally, toward the end of the semester, we wrote our standard library and our demo program.

4.4 Testing Process

After Hello World, we began to add tests for each feature we implemented. Before Hello World, MicroC's testall.sh was adapted for MatchaScript. Two weeks later, Rachel also wrote a shell script for frontend testing, testfront.sh. The person implementing the feature wrote at least one test file per feature; each test file contains multiple test cases designed to test the full functionality of the feature or one aspect of the feature. As we developed the semantic analysis portion of our compiler, we also began to write fail tests.

4.5 Programming Style Guide

Our programming style is mostly based off of JavaScript's syntax, which in some cases resembles C-based languages as well. For the test cases, camelCase was used for variable names. Spaces were placed around operators to make the code as readable as possible. Although whitespace is mostly tossed out by the parser in terms of indentations, the informal style of MatchaScript uses indentations to clarify blocks and functions.

4.6 Software Development Environment

The team contributed to MatchaScript via GitHub. All team members used environments that were able to run MicroC: OCaml 4.01.0 and LLVM.3.6.

4.7 Team Responsibilities

Although we did not adhere to a strict division of labor based on group member titles, every member contributed to the codebase. Below are each team member's actual, rather than planned, contributions to the MatchaScript project.

Team Member	**Responsibilities**
Kimberly Hou	Semantic analyzer, SAST, nested functions (AST/scanner/parser/SAST/analyzer), LRM
Rebecca Mahany	Types, operators, control flow (do-while, frontend of switch), stdlib, LRM, final presentation slides
Jorge Orbay	LRM, final report, demo, testing
Rachel Yang	Hello World, codegen, created testing system, nested functions and function expressions (frontend), classes (frontend), SAST, Include statements

4.8 Project Timeline

The project timeline we aimed for was:

Date	**Milestone**
Feb 8	Complete Proposal

Feb 22	Complete Language Reference Manual
Mar 27	Hello World runs, integration test framework works
April 20	Add small features (MatchaScript types, operators, and statements not supported in MicroC) to frontend
Apr 25	Implement nested functions in frontend
Apr 28	Finish SAST (with nested functions but without classes and arrays), fill in LRM for final report
Apr 30	Finish codegen (with nested functions, without classes and arrays)
May 2	Add classes and arrays to SAST and codegen
May 3	Add try/catch/finally, typeof, string concatenation operator (+), more small features (switch, continue, break, +=, -=, *=, /=)
May 5	Add dictionary, link to C built-in functions in codegen
May 7	Finish Standard Library, write cool sample code
May 9	Clean up code, make powerpoint, finish final report
May 10	Project presentation

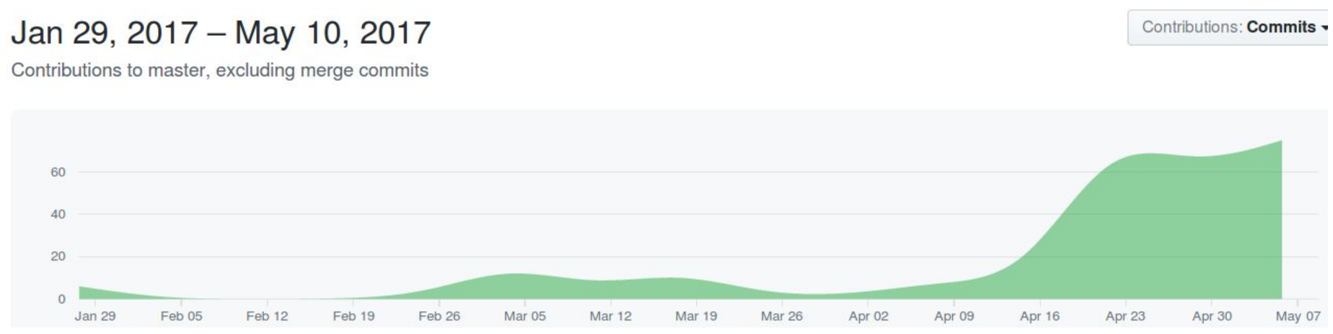
4.9 Project Log

Our actual development log was:

Date	**Milestone**
Feb 8	Completed Proposal
Feb 22	Completed Language Reference Manual
Mar 24	Final report begun
Mar 25	Hello World runs, integration test framework works
Apr 20	Added small features (types, variables, operators) to frontend and backend
Apr 25	Implemented nested functions and function expressions in frontend, added front-end test framework
Apr 29	Added classes to frontend
May 2	All control flow features added to frontend Final project presentation slides begun
May 8	Standard library completed Some control flow features added to backend Final presentation slides completed

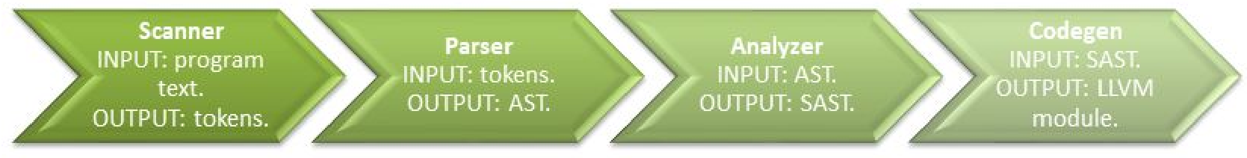
	Demo completed
May 9	Backend working again Project presentation!
May 10	Final report completed Do-while tested and completed Standard library linked during codegen Project submitted

Github Timeline



5 Architecture

5.1 Block Diagram



MatchaScript Compiler

Scanner

MatchaScript's scanner.mll reads in files and tokenizes them. It is able to recognize all operators, keywords, primitive types, variables, whitespace, single-line comments, and multi-line comments.

Parser

The MatchaScript parser takes in the tokens and parses the sequences of tokens as various structures: classes, functions, statements, expressions, etc.

AST

This step in compilation generates an abstract syntax tree representation of the original MatchaScript code. It also contains significant support for pretty-printing MatchaScript code after it is parsed.

Semantic Analyzer

MatchaScript's semantic analyzer is split into two separate files. The first is sast.ml, which specifies output types for the analyzer itself (in the same way that ast.ml specifies output types for the AST). The second is analyzer.ml, which performs type checking and all other requisite semantic analysis. The output is a semantically checked AST, which is similar to the AST produced before but with types attached.

Code Generator

The code generation step takes in the SAST and, from it, produces LLVM code that is an intermediate representation of the original MatchaScript code. In this step, codegen.ml also includes MatchaScript's short standard library in the IR code produced.

Utilities

Supplementary Code

MatchaScript's standard library contains basic functions for type conversion, returning information about objects, system/IO, and math. This standard library is included in all compiled MatchaScript files during the code generation step.

5.2 Interfaces

5.3 Division of Labor - Compiler

- Kimberly: Wrote semantic analyzer (analyzer.ml) and implemented features (nested functions, switch statement); significant contributions to sast.ml
- Jorge: added single-line comments to scanner
- Rebecca: implemented features (types, control flow, etc.) from end-to-end; wrote stdlib
- Rachel: Wrote codegen.ml, implemented features (nested functions, currying, function expressions) in frontend, significant contributions to sast.ml

6 Test Plan and Scripts

6.1 Automated Regression Testing

Front-end Tests

To test the front end (AST) structures built in MatchaScript, front end testing would be performed that "pretty-printed" out the tokens of the AST produced by each test program. The same front end tester would also generate the tokens from the original MatchaScript test files and compare the two token files. If they matched, the AST was considered to be generated properly.

An example is shown below with the test file test-assign-expr.ms.

This first line generates a list of tokens from the test file test-assign-expr.ms.

```
ocaml frontend_tests/scannerprint.ml < tests/test-assign-expr.ms >
      frontend_tests/test-assign-expr.tokens
```

This next line generates a pretty version of the test file test-assign-expr.ms, which is a version of the test file rederived from the AST and produced back into MatchaScript.

```
./MatchaScript.native -a < tests/test-assign-expr.ms >
      test-assign-expr.pretty
```

This next line generates tokens from the .pretty file of the previous line.

```
ocaml frontend_tests/scannerprint.ml < test-assign-expr.pretty >
      test-assign-expr.prettytokens
```

This last line compares the .prettytokens and .tokens files of the previous lines. If the files are equivalent, then the .diff file is empty and the front end test is a success!

```
Compare test-assign-expr.prettytokens
      frontend_tests/test-assign-expr.tokens test-assign-expr.diff
```

All frontend tests can be performed on all available tests by running ./testfront.sh.

Integration Success Testing

To test the successful compilation of MatchaScript files, integration testing would be performed that produced LLVM code. Then, the LLVM code would be run with lli, and the output would be compared with a pre-defined expected output file. An example is shown below with the test file test-assign-expr.ms.

This first line produces an llvm file from the test-assign-expr.ms test file.

```
./MatchaScript.native < tests/test-assign-expr.ms > test-assign-expr.ll
```

This next line produces a .out file by running the llvm file with lli.

```
lli test-assign-expr.ll > test-assign-expr.out
```

This last file compares the predefined output in tests with the output produced from llvm.

```
Compare test-assign-expr.out tests/test-assign-expr.out
       test-assign-expr.diff
```

Integrated Failure Testing

To test the correct failure of a MatchaScript file, integration testing would be performed that threw an exception. The thrown exception of a failed compilation would be compared with a pre-defined expected exception. An example is shown below with the fail test file fail-unknown-id.ms.

```
This first line creates an error file from an attempted compilation.
./MatchaScript.native fail_tests/fail-unknown-id.ms 2> fail-unknown-id.err

This line compares the expected error output with the error file.
Compare fail_tests/fail-unknown-id.err fail-unknown-id.err fail-unknown-id.diff
```

All successful and failure integrated testing can be run with **./testall.sh**.

6.2 Sample Test Programs

Test 1

tests/test-assign-expr.ms

```
int a = 3;
int b;
b=a;
print(b); //should be getting a 3 here
```

test-assign-expr.ll

```
; ModuleID = 'MatchaScript'

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"

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

define i32 @main() {
entry:
  %b = alloca i32
  %a = alloca i32
  store i32 3, i32* %a
  store i32 0, i32* %b
  %a1 = load i32* %a
  store i32 %a1, i32* %b
  %b2 = load i32* %b
  %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmt, i32 0, i32 0), i32 %b2)
  ret i32 0
}
```

Test 2
tests/test-print-hello-world.ms

```
print(“Hello world!”);
```

test-print-hello-world.ll

```
; ModuleID = 'MatchaScript'

@string = private unnamed_addr constant [13 x i8] c"Hello world!\00"
@fmt = private unnamed_addr constant [4 x i8] c"%s\0A\00"

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

define i32 @main() {
entry:
  %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmt, i32 0, i32 0), i8*
getelementptr inbounds ([13 x i8]* @string, i32 0, i32 0))
  ret i32 0
}
```

Test 3
tests/test-include-math-stdlib.ms

```
print(“Hello world!”);
```

test-include-math-stdlib.ll

```
; ModuleID = 'MatchaScript'

@string = private unnamed_addr constant [35 x i8] c"a = -4.3. absolute value of a is: \00"
@fmt = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@fmt1 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
@string2 = private unnamed_addr constant [34 x i8] c"b = 4.5. b to the power of 3 is: \00"
@fmt3 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@fmt4 = private unnamed_addr constant [4 x i8] c"%f\0A\00"

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

define i32 @int_abs(i32 %i) {
entry:
  %i1 = alloca i32
  store i32 %i, i32* %i1
  %i2 = load i32* %i1
  %sgttmp = icmp sgt i32 %i2, 0
  br i1 %sgttmp, label %then, label %else

merge:                                    ; No predecessors!
  ret i32 0

then:                                     ; preds = %entry
  %i3 = load i32* %i1
  ret i32 %i3

else:                                     ; preds = %entry
  %i4 = load i32* %i1
  %multtmp = mul i32 -1, %i4
  ret i32 %multtmp
}

define double @float_abs(double %i) {
entry:
  %i1 = alloca double
  store double %i, double* %i1
  %i2 = load double* %i1
  %flt_sgttmp = fcmp ogt double %i2, 0.000000e+00
  br i1 %flt_sgttmp, label %then, label %else

merge:                                    ; preds = %else
  %i4 = load double* %i1
  %flt_multmp = fmul double -1.000000e+00, %i4
  ret double %flt_multmp

then:                                     ; preds = %entry
  %i3 = load double* %i1
  ret double %i3

else:                                     ; preds = %entry
  br label %merge
}

define i32 @int_max(i32 %a, i32 %b) {
entry:
  %a1 = alloca i32
  store i32 %a, i32* %a1
  %b2 = alloca i32
  store i32 %b, i32* %b2
  %a3 = load i32* %a1
  %b4 = load i32* %b2
  %sgttmp = icmp sgt i32 %a3, %b4
  br i1 %sgttmp, label %then, label %else

merge:                                    ; No predecessors!
  ret i32 0

then:                                     ; preds = %entry
```

```
 %a5 = load i32* %a1
 ret i32 %a5

else:                                ; preds = %entry
 %b6 = load i32* %b2
 ret i32 %b6
}

define i32 @int_min(i32 %a, i32 %b) {
entry:
 %a1 = alloca i32
 store i32 %a, i32* %a1
 %b2 = alloca i32
 store i32 %b, i32* %b2
 %a3 = load i32* %a1
 %b4 = load i32* %b2
 %lesstmp = icmp slt i32 %a3, %b4
 br i1 %lesstmp, label %then, label %else

merge:                                ; No predecessors!
 ret i32 0

then:                                ; preds = %entry
 %a5 = load i32* %a1
 ret i32 %a5

else:                                ; preds = %entry
 %b6 = load i32* %b2
 ret i32 %b6
}

define i32 @int_pow(i32 %base, i32 %pow) {
entry:
 %base1 = alloca i32
 store i32 %base, i32* %base1
 %pow2 = alloca i32
 store i32 %pow, i32* %pow2
 %result = alloca i32
 %i = alloca i32
 store i32 1, i32* %i
 %base3 = load i32* %base1
 store i32 %base3, i32* %result
 br label %while

while:                                ; preds = %while_body, %entry
 %i7 = load i32* %i
 %pow8 = load i32* %pow2
 %lesstmp = icmp slt i32 %i7, %pow8
 br i1 %lesstmp, label %while_body, label %merge

while_body:                                ; preds = %while
 %result4 = load i32* %result
 %base5 = load i32* %base1
```

```
  %multtmp = mul i32 %result4, %base5
  store i32 %multtmp, i32* %result
  %i6 = load i32* %i
  %addtmp = add i32 %i6, 1
  store i32 %addtmp, i32* %i
  br label %while

merge:                                  ; preds = %while
  %result9 = load i32* %result
  ret i32 %result9
}

define double @float_pow(double %base, i32 %pow) {
entry:
  %base1 = alloca double
  store double %base, double* %base1
  %pow2 = alloca i32
  store i32 %pow, i32* %pow2
  %result = alloca double
  %i = alloca i32
  store i32 1, i32* %i
  %base3 = load double* %base1
  store double %base3, double* %result
  br label %while

while:                                  ; preds = %while_body, %entry
  %i7 = load i32* %i
  %pow8 = load i32* %pow2
  %lesstmp = icmp slt i32 %i7, %pow8
  br i1 %lesstmp, label %while_body, label %merge

while_body:                             ; preds = %while
  %result4 = load double* %result
  %base5 = load double* %base1
  %flt_multmp = fmul double %result4, %base5
  store double %flt_multmp, double* %result
  %i6 = load i32* %i
  %addtmp = add i32 %i6, 1
  store i32 %addtmp, i32* %i
  br label %while

merge:                                  ; preds = %while
  %result9 = load double* %result
  ret double %result9
}

define double @ceil(double %f) {
entry:
  %f1 = alloca double
  store double %f, double* %f1
  %new_f = alloca double
  %remainder = alloca double
  %f2 = load double* %f1
  %flt_sremtmp = frem double %f2, 1.000000e+00
  store double %flt_sremtmp, double* %remainder
  %f3 = load double* %f1
  %flt_addtmp = fadd double %f3, 1.000000e+00
  %remainder4 = load double* %remainder
  %flt_subtmp = fsub double %flt_addtmp, %remainder4
  store double %flt_subtmp, double* %new_f
  %new_f5 = load double* %new_f
  ret double %new_f5
}

define double @floor(double %f) {
entry:
  %f1 = alloca double
  store double %f, double* %f1
  %new_f = alloca double
  %remainder = alloca double
  %f2 = load double* %f1
  %flt_sremtmp = frem double %f2, 1.000000e+00
  store double %flt_sremtmp, double* %remainder
  %f3 = load double* %f1
  %remainder4 = load double* %remainder
  %flt_subtmp = fsub double %f3, %remainder4
  store double %flt_subtmp, double* %new_f
  %new_f5 = load double* %new_f
  ret double %new_f5
}

define double @round(double %f) {
entry:
  %f1 = alloca double
  store double %f, double* %f1
  %result = alloca double
  %floorOfF = alloca double
  %absOfF = alloca double
  %f2 = load double* %f1
  %float_abs_result = call double @float_abs(double %f2)
  store double %float_abs_result, double* %absOfF
  %f3 = load double* %f1
  %floor_result = call double @floor(double %f3)
  store double %floor_result, double* %floorOfF
  store double 0.000000e+00, double* %result
  %absOfF4 = load double* %absOfF
  %floorOfF5 = load double* %floorOfF
  %flt_subtmp = fsub double %absOfF4, %floorOfF5
  %flt_sgetmp = fcmp oge double %flt_subtmp, 5.000000e-01
  br i1 %flt_sgetmp, label %then, label %else

merge:                                  ; preds = %else, %then
  %result8 = load double* %result
  ret double %result8

then:                                   ; preds = %entry
```

```
  %floorOfF6 = load double* %floorOfF
  %flt_addtmp = fadd double %floorOfF6, 1.000000e+00
  store double %flt_addtmp, double* %result
  br label %merge

else:                                    ; preds = %entry
  %floorOfF7 = load double* %floorOfF
  store double %floorOfF7, double* %result
  br label %merge
}

define i32 @main() {
entry:
  %b = alloca double
  %a = alloca double
  %float_abs_result = call double @float_abs(double -4.300000e+00)
  store double %float_abs_result, double* %a
  %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmt, i32 0, i32 0), i8*
getelementptr inbounds ([35 x i8]* @string, i32 0, i32 0))
  %a1 = load double* %a
  %printf2 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmt1, i32 0, i32 0), double %a1)
  %float_pow_result = call double @float_pow(double 4.500000e+00, i32 3)
  store double %float_pow_result, double* %b
  %printf3 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmt3, i32 0, i32 0), i8*
getelementptr inbounds ([34 x i8]* @string2, i32 0, i32 0))
  %b4 = load double* %b
  %printf5 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @fmt4, i32 0, i32 0), double %b4)
  ret i32 0
}
```

Our full test suite is included in the Appendix.

6.3 Division of Labor - Testing

- Rachel - Modified testall.sh (integration tests) and testfront.sh (frontend tests) from MicroC and set up test directories to create MatchaScript's regression test framework. Wrote test scripts for features she implemented.
- Kimberly - Wrote most fail tests and test cases for specific features. Wrote test scripts for features she implemented.
- Rebecca - Wrote test scripts for all MatchaScript features she implemented. Wrote tests for the standard library.
- Jorge - Wrote test scripts

7 Lessons Learned and Conclusions

Kimberly Hou

Throughout the semester Prof. Edwards warned us to implement the entire compiler feature by feature, even if it means breaking down or re-doing the components involved. While we agreed in our minds, we ended up working layer by layer of the compiler instead of following his advice, working first on the scanner/AST/parser until we were happy with it, and then looking at the semantic analyzer, and then working on codegen. It was only in the last couple weeks of the project that we learned we needed to work on all parts of the compiler at the same time, and once we agreed on an SAST, Rachel worked on Codegen while I worked on the Semantic Analyzer. Obviously some features are more difficult than others to implement, and the way we went about working did not lend itself to timeliness or efficiency. By working on the compiler in its entirety per feature, we would always have a working compiler at any given time. This is perhaps one of the more underrated and underappreciated facts that I learned only after most of the semester went by.

Rebecca Mahany

I think the biggest lesson I learned was about having realistic expectations. We went into this project with an extremely ambitious number of features that, as it turned out, just weren't feasible to implement in one semester (e.g., garbage collection). If I had to do it over again, I would do more initial research during the project proposal stage to create realistic project goals.

I also wish that I had started working on the project sooner (which is, probably, what every group ever says). I thought I didn't have enough knowledge yet to begin working on MatchaScript and wanted to learn more in class before beginning, but I don't think that was actually the case. For example, one of the most useful things I did in order to be able to start working on MatchaScript was sitting down with Rachel and going through MicroC's codegen line by line until we were sure we understood all of it. Even though we hadn't had the lecture on codegen yet, we were then much more able to begin adding features to our codebase from end to end.

Finally, I really liked the suggested approach of iteratively adding features to our language. To me, it was a very efficient workflow, and working in our own branches on GitHub meant that we could all be working on separate features without breaking master or each other's code.

Jorge Orbay

I agree with Rebecca's points above. I would emphasize that if an ambitious number of features is being attempted, you should be especially wary of bottlenecks. If a feature proves to be exceedingly difficult, temporarily shelve it and try to implement another feature. Throwing more time at a difficult feature, especially when there are others further down the line, is extremely costly.

Rachel Yang

Aside from what my teammates noted (don't get caught up in one difficult feature, always have a compiler working from end-to-end at a time), my biggest "lessons learned" is that it's really important to specify the interfaces between parts of the code, so that multiple people can work on different parts of the code that depend on the interface at the same time. Once we figured out our SAST, Kimberly and I were able to make simultaneous progress on Analyzer and Codegen.

8 Appendix: Full Code Listing

scanner.mll

```
(* Ocamllex scanner for MatchaScript *)

{ open Parser }

let alpha = ['a'-'z' 'A'-'Z']
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let escape_char = ''' (escape) '''
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let digit = ['0'-'9']
let id = alpha (alpha | digit | '_')*
let string = '"' ( (ascii | escape)* as s) '"'
let char = ''' ( ascii | digit ) '''
let float = (digit+) ['.'] digit*
let int = digit+
let whitespace = [' ' '\t' '\r' '\n']

rule token = parse
 whitespace { token lexbuf }
| "/*"   { comment lexbuf }
| "//"   { line_comment lexbuf }
| '('    { LPAREN }
| ')'    { RPAREN }
| '{'    { LBRACE }
| '}'    { RBRACE }
| ';'    { SEMI }
| ','    { COMMA }
```

```
(* Operators *)
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '%'     { MOD }
| '='       { ASSIGN }
| "=="      { EQ }
| "!="      { NEQ }
| '<'     { LT }
| "<="    { LEQ }
| ">"     { GT }
| ">="      { GEQ }
| "&&"      { AND }
| "||"     { OR }
| "!"      { NOT }
| '.'    { DOT }
| '['    { LBRACKET }
| ']'    { RBRACKET }

(* Function *)
| "function" { FUNCTION }

(* Branch control *)
| "if"       { IF }
| "else"      { ELSE }
| "for"       { FOR }
| "while"      { WHILE }
| "do"       { DO }
| "return"     { RETURN }

(* Data types *)
| "int"    { INT }
| "float" { FLOAT }
| "char"  { CHAR }
| "string" { STRING }
| "bool"  { BOOL }
| "void"  { VOID }
| "true"  { TRUE }
| "false" { FALSE }
| "fun"     { FUN }
```

```
(* Include *)
| "#"      { POUND }
| "include"    { INCLUDE }
| "ms"      { MS }

(* Literals *)
| int as lxm    { INTLIT(int_of_string lxm) }
| float as lxm  { FLOATLIT(float_of_string lxm) }
| id as lxm    { ID(lxm) }
| string      { STRINGLIT(s) }
| char as lxm  { CHARLIT( String.get lxm 1 ) }
| eof         { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
 "*/" { token lexbuf }
| _   { comment lexbuf }

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

parser.mly

```
%{
open Ast
%}

%token FUNCTION INCLUDE MS
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA LBRACKET RBRACKET DOT POUND
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE DO
%token INT FLOAT BOOL CHAR STRING FUN VOID
```

```
%token <int> INTLIT
%token <float> FLOATLIT
%token <char> CHARLIT
%token <string> STRINGLIT
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT NEG

%start program
%type <Ast.program> program

%%

program:
 includes stmt_list EOF { Program($1, $2) }

/*****************
 INCLUDE
*****************/
includes:
  /* nothing */   { [] }
  |  include_list { $1 }

include_list:
    include_decl          { [$1] }
  |  include_list include_decl { $1@[$2] }

include_decl:
 POUND INCLUDE STRINGLIT SEMI { Include($3) }

/*********
typs
```

```
**********/
typ:
   INT   { Int }
 | FLOAT { Float }
 | BOOL  { Bool }
 | CHAR  { Char }
 | STRING { String }
 | VOID  { Void }
 | FUN   { Fun }

/*********
Variables
**********/
vdecl:
   typ ID SEMI { match $1 with
            Int -> ($1, $2, IntLit(0))
           | Float -> ($1, $2, FloatLit(0.0))
           | Bool -> ($1, $2, BoolLit(false))
           | String -> ($1, $2, StringLit(""))
          }
 | typ ID ASSIGN expr SEMI { ($1, $2, $4) }

/*********
Functions
**********/
fdecl:
  FUNCTION typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
   { { fdReturnType = $2;
  fdFname = $3;
  fdFormals = $5;
  fdBody = $8 } }

fexpr:
 FUNCTION typ LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
 { { feReturnType = $2;
  feFormals = $4;
  feBody = $7 } }

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

```
formal_list:
   typ ID           { [($1,$2)] }
 | formal_list COMMA typ ID { ($3,$4) :: $1 }


/*********
Statements
**********/
stmt_list:
   stmt  { [$1] }
 | stmt_list stmt { $1@[$2] } /* append statement to end of statement list */

stmt:
   expr SEMI { ExprStmt $1 }
 | vdecl { VarDecl($1) }
 | fdecl { FunDecl($1) }
 | RETURN expr SEMI { Return $2 }
 | RETURN SEMI { Return Noexpr }
 | LBRACE stmt_list RBRACE { Block($2) }
 | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
 | IF LPAREN expr RPAREN stmt ELSE stmt   { If($3, $5, $7) }
 | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
    { For($3, $5, $7, $9) }
 | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
 | DO stmt WHILE LPAREN expr RPAREN SEMI { DoWhile ($2, $5) }

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

callee:
   callee LPAREN actuals_opt RPAREN  { CallExpr($1, $3) }
 | ID        { Id($1) }

expr:
   INTLIT       { IntLit($1)       }
 | FLOATLIT      { FloatLit($1)      }
 | CHARLIT      { CharLit($1)      }
 | STRINGLIT     { StringLit($1)     }
```

```
  | TRUE          { BoolLit(true)      }
  | FALSE         { BoolLit(false)     }
  | ID          { Id($1)          }
  | fexpr        { FunExpr($1)       }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr MOD    expr { Binop($1, Mod,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr       { Unop(Not, $2) }
  | expr ASSIGN expr  { Assign($1, $3) }
  | callee LPAREN actuals_opt RPAREN { CallExpr($1, $3) }
  | LPAREN expr RPAREN { $2 }

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

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

ast.ml

```
print(“Hello wo(* Abstract Syntax Tree *)

(* Style Guide:
   AST:
     - typecategory
```

```
    - SpecificType
  Parser:
    - TERMINAL
    - non_terminal
*)

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

type uop = Neg | Not

type typ = Int | Float | Bool | Char | Void | Fun | String

type bind = typ * string

and vdecl = typ * string * expr

and expr =
  IntLit of int
 | FloatLit of float
 | BoolLit of bool
 | CharLit of char
 | StringLit of string
 | FunExpr of fexpr
 | Id of string
 | Binop of expr * op * expr
 | Unop of uop * expr
 | Assign of expr * expr
 | CallExpr of expr * expr list
 | Noexpr

and caseType = Default | CaseType of expr

and stmt =
 | Block of stmt list
 | ExprStmt of expr
 | VarDecl of vdecl
 | FunDecl of fdecl
 | Return of expr
 | If of expr * stmt * stmt
 | For of expr * expr * expr * stmt
 | While of expr * stmt
 | DoWhile of stmt * expr

and fexpr = {
 feReturnType : typ;
 feFormals : bind list;
 feBody: stmt list;
}

and fdecl = {
 fdReturnType : typ;
 fdFname : string;
 fdFormals : bind list;
 fdBody : stmt list;
}

type include_stmt = Include of string

(* type program = include_stmt list * constructs *)
type program = Program of include_stmt list * stmt list

(* Pretty-printing functions *)
let string_of_op = function
   Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

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

let string_of_typ = function
   Int -> "int"
  | Float -> "float"
  | Bool -> "bool"
  | Char -> "char"
  | String -> "string"
  | Void -> "void"
  | Fun -> "fun"

let string_of_bind (t, id) = string_of_typ t ^ " " ^ id

let rec string_of_expr = function
   IntLit(i) -> string_of_int i
  | FloatLit(f) -> string_of_float f
  | BoolLit(b) -> if b then "true" else "false"
  | CharLit(c) -> "\'" ^ String.make 1 c ^ "\'"
  | StringLit(s) -> "\"" ^ s ^ "\""
  | FunExpr(f) -> string_of_fexpr f
  | Id(s) -> s
  | Binop(e1, o, e2) ->
    string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
  | CallExpr(call_expr, args) ->
```

```
    string_of_expr call_expr ^ "(" ^ String.concat ", " (List.map string_of_expr args) ^ ")"
  | Noexpr -> ""

and string_of_caseType = function
    Default -> "default"
  | CaseType(c) -> "case " ^ string_of_expr c

and string_of_stmt = function
    Block(stmts) ->
     "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | ExprStmt(expr) -> string_of_expr expr ^ ";\n"
  | VarDecl(v) -> string_of_vdecl v
  | FunDecl(fd) -> string_of_fdecl fd
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
     string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
     "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
     string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | DoWhile(s, e) -> "do " ^ string_of_stmt s ^ " while (" ^ string_of_expr e ^ ");"

and string_of_vdecl (typ, str, expr) =
  if expr = Noexpr then string_of_typ typ ^ " " ^ str ^ ";\n"
  else string_of_typ typ ^ " " ^ str ^ " = " ^ string_of_expr expr ^ ";\n"

and string_of_fexpr fexpr =
  "function " ^ string_of_typ fexpr.feReturnType ^ " "
  ^ "(" ^ String.concat ", " (List.map string_of_bind fexpr.feFormals) ^
  ")\n{\n" ^ String.concat "" (List.map string_of_stmt fexpr.feBody) ^ "}"

and string_of_fdecl fdecl =
  "function " ^ string_of_typ fdecl.fdReturnType ^ " " ^
  fdecl.fdFname ^ "(" ^ String.concat ", " (List.map string_of_bind fdecl.fdFormals) ^
  ")\n{\n" ^ String.concat "" (List.map string_of_stmt fdecl.fdBody) ^ "}"

let string_of_include = function
  Include(s) -> "#include<" ^ s ^ ">;\n"

let string_of_program prog = match prog with
  Program(includes, stmts) ->
    String.concat "" (List.map string_of_include includes) ^ "\n";
    String.concat "" (List.map string_of_stmt stmts) ^ "\n";
```

```
(* Abstract Syntax Tree *)

(* Style Guide:
    AST:
      - typecategory
      - SpecificType
    Parser:
      - TERMINAL
      - non_terminal
*)

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

type uop = Neg | Not

type typ = Int | Float | Bool | Char | Void | Fun | String

type bind = typ * string

and vdecl = typ * string * expr

and expr =
    IntLit of int
  | FloatLit of float
  | BoolLit of bool
  | CharLit of char
  | StringLit of string
  | FunExpr of fexpr
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of expr * expr
  | CallExpr of expr * expr list
  | Noexpr

and caseType = Default | CaseType of expr

and stmt =
  | Block of stmt list
  | ExprStmt of expr
  | VarDecl of vdecl
  | FunDecl of fdecl
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | DoWhile of stmt * expr

and fexpr = {
  feReturnType : typ;
  feFormals : bind list;
  feBody: stmt list;
```

```
}

and fdecl = {
  fdReturnType : typ;
  fdFname : string;
  fdFormals : bind list;
  fdBody : stmt list;
}

type include_stmt = Include of string

(* type program = include_stmt list * constructs *)
type program = Program of include_stmt list * stmt list

(* Pretty-printing functions *)
let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

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

let string_of_typ = function
    Int -> "int"
  | Float -> "float"
  | Bool -> "bool"
  | Char -> "char"
  | String -> "string"
  | Void -> "void"
  | Fun -> "fun"

let string_of_bind (t, id) = string_of_typ t ^ " " ^ id

let rec string_of_expr = function
    IntLit(i) -> string_of_int i
  | FloatLit(f) -> string_of_float f
  | BoolLit(b) -> if b then "true" else "false"
  | CharLit(c) -> "\'" ^ String.make 1 c ^ "\'"
  | StringLit(s) -> "\"" ^ s ^ "\""
  | FunExpr(f) -> string_of_fexpr f
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
  | CallExpr(call_expr, args) ->
      string_of_expr call_expr ^ "(" ^ String.concat ", " (List.map string_of_expr
args) ^ ")"
  | Noexpr -> ""

and string_of_caseType = function
    Default -> "default"
  | CaseType(c) -> "case " ^ string_of_expr c

and string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | ExprStmt(expr) -> string_of_expr expr ^ ";\n"
  | VarDecl(v) -> string_of_vdecl v
  | FunDecl(fd) -> string_of_fdecl fd
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | DoWhile(s, e) -> "do " ^ string_of_stmt s ^ " while (" ^ string_of_expr e ^ ");"

and string_of_vdecl (typ, str, expr) =
  if expr = Noexpr then string_of_typ typ ^ " " ^ str ^ ";\n"
  else string_of_typ typ ^ " " ^ str ^ " = " ^ string_of_expr expr ^ ";\n"

and string_of_fexpr fexpr =
  "function " ^ string_of_typ fexpr.feReturnType ^ " "
  ^ "(" ^ String.concat ", " (List.map string_of_bind fexpr.feFormals) ^
  ")\n{\n" ^ String.concat "" (List.map string_of_stmt fexpr.feBody) ^ "}"

and string_of_fdecl fdecl =
  "function " ^ string_of_typ fdecl.fdReturnType ^ " " ^
  fdecl.fdFname ^ "(" ^ String.concat ", " (List.map string_of_bind fdecl.fdFormals)
^
  ")\n{\n" ^ String.concat "" (List.map string_of_stmt fdecl.fdBody) ^ "}"
```

```
let string_of_include = function
  Include(s) -> "#include<" ^ s ^ ">;\n"

let string_of_program prog = match prog with
  Program(includes, stmts) ->
    String.concat "" (List.map string_of_include includes) ^ "\n";
    String.concat "" (List.map string_of_stmt stmts) ^ "\n";
```

sast.ml

```
open Ast

type sbind = typ * string

type svdecl = typ * string * sexpr

and sexpr =
    SIntLit of int
      | SFloatLit of float
      | SBoolLit of bool
      | SCharLit of char
      | SStringLit of string
      | SFunExpr of sfexpr
      | SId of string * typ
      | SBinop of sexpr * op * sexpr * typ
      | SUnop of uop * sexpr * typ
      | SAssign of sexpr * sexpr * typ
      | SCallExpr of sexpr * sexpr list * typ
    | STernary of sexpr * sexpr * sexpr
      | SNoexpr

and sstmt =
    SBlock of sstmt list
  | SExprStmt of sexpr
  | SVarDecl of svdecl
  | SFunDecl of sfdecl
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt
  | SDoWhile of sstmt * sexpr

and sfexpr = {
  sfeReturnType : typ;
  sfeFormals : sbind list;
  sfeBody: sstmt list;
}

and sfdecl = {
  sfdReturnType : typ;
  sfdFname : string;
  sfdFormals : bind list;
  sfdBody : sstmt list;
}
(*
and scdecl = {
  scname : string;
  scproperties : svdecl list;
}
*)
and sprogram = sstmt list

(* Pretty-printing functions *)
let string_of_sbind (t, id) = string_of_typ t ^ " " ^ id

let rec string_of_sexpr = function
    SIntLit(i) -> string_of_int i
  | SFloatLit(f) -> string_of_float f
  | SBoolLit(b) -> if b then "true" else "false"
  | SCharLit(c) -> "\'" ^ String.make 1 c ^ "\'"
  | SStringLit(s) -> "\"" ^ s ^ "\""
  | SFunExpr(f) -> string_of_sfexpr f
  | SId(s,t) -> s ^ "#" ^ string_of_typ t ^ "#"
  | SBinop(e1, o, e2, t) ->
     "("^ string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
^")#"^string_of_typ t^"#"
  | SUnop(o, e, t) -> string_of_uop o ^ string_of_sexpr e ^"#"^string_of_typ t^"#"
  | SAssign(e1, e2, t) -> "("^string_of_sexpr e1 ^ " = " ^ string_of_sexpr e2
^")#"^string_of_typ t^"#"
  | SCallExpr(call_expr, args, t) ->
      string_of_sexpr call_expr ^ "(" ^ String.concat ", " (List.map string_of_sexpr
args) ^ ")" ^"#"^string_of_typ t^"#"
  | SNoexpr -> ""

and string_of_sstmt = function
    SBlock(sstmts) ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt sstmts) ^ "}\n"
  | SExprStmt(sexpr) -> string_of_sexpr sexpr ^ ";\n"
  | SVarDecl(v) -> string_of_svdecl v
  | SFunDecl(fd) -> string_of_sfdecl fd
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
  | SIf(e, s, SBlock([])) -> "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
      "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
      string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s

and string_of_svdecl (typ, str, sexpr) =
  if sexpr = SNoexpr then string_of_typ typ ^ " " ^ str ^ ";\n"
  else string_of_typ typ ^ " " ^ str ^ " = " ^ string_of_sexpr sexpr ^ ";\n"

and string_of_sfexpr sfexpr =
  "function " ^ string_of_typ sfexpr.sfeReturnType ^ " "
  ^ "(" ^ String.concat ", " (List.map string_of_sbind sfexpr.sfeFormals) ^
  ")\n{\n" ^ String.concat "" (List.map string_of_sstmt sfexpr.sfeBody) ^ "}"

and string_of_sfdecl sfdecl =
  "function " ^ string_of_typ sfdecl.sfdReturnType ^ " " ^
  sfdecl.sfdFname ^ "(" ^ String.concat ", " (List.map string_of_sbind
sfdecl.sfdFormals) ^
  ")\n{\n" ^ String.concat "" (List.map string_of_sstmt sfdecl.sfdBody) ^ "}"


and string_of_sprogram sstmts = String.concat "" (List.map string_of_sstmt sstmts) ^
"\n"
```

analyzer.ml

```
module S = Sast
module A = Ast
module E = Exceptions
module Str = Str
module StringMap = Map.Make(String)

type symbol_table = {
      parent: symbol_table option; (* option means a parent scope is optional *)
            name          : string;
      mutable variables   : A.vdecl list;
            return_type   : A.typ;
      mutable formals           : A.bind list;
      mutable      fun_names    : A.fdecl list;
}

(***************************************************
 * Environment in which we're doing semantic checking
 ****************************************************)
type translation_env = {
      scope : symbol_table;            (* tracks in-scope vars, functions, and
classes *)
      in_for       : bool;             (* whether in for loop context *)
      in_while     : bool;             (* whether in while loop context *)
}

let update_env_context tenv in_for in_while = {
      scope        = tenv.scope;
      in_for       = in_for;
      in_while     = in_while;
}

(*********************
* Helper functions
*********************)
let rec find_variable (scope : symbol_table) name =  (* takes in a scope of type
symbol_table *)
  try List.find (fun (_,n,_) -> n = name) scope.variables
  with Not_found ->
    match scope.parent with
      Some(parent) -> find_variable parent name (* keep searching each parent's scope
if not found until there's no more parent *)
    | _ -> raise Not_found

let rec find_formal (scope : symbol_table) name =  (* takes in a scope of type
symbol_table *)
  try List.find (fun (_,n) -> n = name) scope.formals
  with Not_found ->
    match scope.parent with
      Some(parent) -> find_formal parent name (* keep searching each parent's scope
if not found until there's no more parent *)
    | _ -> raise Not_found

let find_fun_vdecl (scope : symbol_table) name =
  try List.find (fun (_,n,_) -> n = name) scope.variables
  with Not_found -> raise Not_found

let check_call_fun_vdecls tenv i =  (* [identifying hello in] fun hello = function
fun... *)
   let vdecl = try find_fun_vdecl tenv.scope i
    with | Not_found -> (A.Void, "Not Found", A.IntLit(0)) in
      let get_vdecl_typ (typ,_,_) = typ in
         get_vdecl_typ vdecl

let find_fdecl (scope : symbol_table) name =
  let get_name f = f.A.fdFname in
  try List.find (fun x -> (get_name x) = name) scope.fun_names
  with Not_found -> raise Not_found

let find_parent_fdecl (scope : symbol_table) name =
  match scope.parent with
  | Some(parent) ->
      let get_name f = f.A.fdFname in
      try List.find (fun x -> (get_name x) = name) parent.fun_names
      with Not_found -> raise Not_found
  | _ -> raise Not_found

let search_vdecls tenv i =
   let vdecl = try find_variable tenv.scope i
    with | Not_found -> (A.Void, "Not Found", A.IntLit(0)) in
      let get_vdecl_typ (typ,_,_) = typ in
         get_vdecl_typ vdecl

let search_fdecls tenv i =
  let f = try find_fdecl tenv.scope i
  with | Not_found -> { A.fdReturnType = A.Void;
               A.fdFname = "A.Void";
               A.fdFormals = [];
                A.fdBody = []; } in
  let get_fname fdecl = fdecl.A.fdFname in
  let fname = get_fname f in
  if fname = "A.Void" then A.Void else A.Fun

let search_parent_fdecls tenv i =
  let f = try find_parent_fdecl tenv.scope i
  with | Not_found -> { A.fdReturnType = A.Void;
               A.fdFname = "A.Void";
               A.fdFormals = [];
                A.fdBody = []; } in
  let get_fname fdecl = fdecl.A.fdFname in
  let fname = get_fname f in
  if fname = "A.Void" then A.Void else A.Fun

let search_formals tenv i =
   let formal = try find_formal tenv.scope i
     with | Not_found -> (A.Void, "Void") in
       let get_formal_typ (typ,_) = typ in
         get_formal_typ formal

let get_id_type tenv name =
  let i = if name = "main" then "mainuserentryfunctionformatchascriptlanguage" else
name in
  let typ = search_vdecls tenv i in
  if typ = A.Void
  then begin
   let ft = search_fdecls tenv i in
     if ft = A.Void
     then (let t = search_formals tenv i in
           if t = A.Void
           then (let t = search_parent_fdecls tenv i in
                 if t = A.Void then raise(E.UndeclaredIdentifier(i))
                 else t)
           else t)
     else ft
  end
  else typ

let check_id tenv i =
  if i = "main" then
  let n = "mainuserentryfunctionformatchascriptlanguage" in
  S.SId(n, get_id_type tenv i)
  else
  S.SId(i, get_id_type tenv i)

(* helper for check_fdecl, checks for fdecl name dup *)
let rec find_fdecl_name (scope : symbol_table) name =  (* takes in a scope of type
symbol_table *)
  if (scope.name <> name)
  then match scope.parent with
      Some(parent) -> find_fdecl_name parent name
    | _ -> raise Not_found
  else raise(E.DuplicateFunction(name))

let rec check_block tenv sl = match sl with
        [] -> S.SBlock([S.SExprStmt(S.SNoexpr)]), tenv (* empty block *)
       | _ -> let sl, _ = check_stmt_list tenv sl in
        S.SBlock(sl), tenv

and check_call_fun tenv name args =
  let get_name f = f.A.fdFname in
  let fdecl = try List.find (fun x -> (get_name x) = name) tenv.scope.fun_names
  with Not_found -> (match tenv.scope.parent with
  | Some(parent) ->
      let get_name f = f.A.fdFname in
      try List.find (fun x -> (get_name x) = name) parent.fun_names
      with Not_found -> { A.fdReturnType = A.Void;
               A.fdFname = "A.Void";
               A.fdFormals = [];
                A.fdBody = []; }
       | _ -> Printf.printf "Found!\n"; { A.fdReturnType = A.Void;
               A.fdFname = "A.Void";
```

```
              A.fdFormals = [];
               A.fdBody = []; }
  | _ -> { A.fdReturnType = A.Void;
              A.fdFname = "A.Void";
              A.fdFormals = [];
               A.fdBody = []; } )
  in (* check if number and type of args are the same *) (
          let exprs = List.map (fun x-> A.string_of_expr x) args in
     if (List.length fdecl.A.fdFormals) = (List.length args) then
       (let get_s (s,_) = s in
        let sargs = List.map (fun x -> get_s (check_expr tenv x)) args in (* get list
of arguments *)
        let sarg_types = List.map (fun x -> get_sexpr_type x) sargs in (* get list of
argument types *)
        let formal_types = List.map (fun x -> get_s x) fdecl.A.fdFormals in (* get
list of parameter types *)
        let compare = List.map2(fun x y -> if x <> y then
raise(E.IncorrectTypeOfArgument(A.string_of_typ x, A.string_of_typ y)) else x)
sarg_types formal_types in
        let result = if (compare=sarg_types) then  { A.fdReturnType = A.Void;
              A.fdFname = name;
              A.fdFormals = [];
               A.fdBody = []; }
 else raise(E.IncorrectNumberOfArguments) in result)
         else raise(E.IncorrectNumberOfArguments))

 and check_call_wrapper tenv i args =
  let rec check_call_helper tenv (scope : symbol_table) name args =
    if (scope.name <> name)      (* search for function name in scope *)
    then match scope.parent with
      Some(parent) -> check_call_helper tenv parent name args
    | _ -> raise Not_found
    else (* check if number and type of args are the same *) (
        let exprs = List.map (fun x-> A.string_of_expr x) args in
     if (List.length scope.formals) = (List.length args) then
       (let get_s (s,_) = s in
        let sargs = List.map (fun x -> get_s (check_expr tenv x)) args in (* get list
of arguments *)
        let sarg_types = List.map (fun x -> get_sexpr_type x) sargs in (* get list of
argument types *)
        let formal_types = List.map (fun x -> get_s x) scope.formals in (* get list
of parameter types *)
        let compare = List.map2(fun x y -> if x <> y then
raise(E.IncorrectTypeOfArgument(A.string_of_typ x, A.string_of_typ y)) else [])
sarg_types formal_types in
        let result = if (compare = []) then name else raise Not_found in result)
         else raise(E.IncorrectNumberOfArguments)) in
  let res = try check_call_helper tenv tenv.scope i args
```

```
   with Not_found -> "A.Void" in
     res

and check_fexpr tenv f =
  let get_bind_string (_,s) = s in
  let formals_map = List.fold_left (fun m formal ->  (* check formal dups within
current function *)
    if StringMap.mem (get_bind_string formal) m
    then raise(E.DuplicateFormal(get_bind_string formal))
    else StringMap.add (get_bind_string formal) formal m) StringMap.empty
f.A.feFormals in
  let scope' = (* create a new scope *)
       {
       parent = Some(tenv.scope); (* parent may or may not exist *)
       variables = [];
       name = "anon"; (* anonymous function *)
       return_type = f.A.feReturnType;
       formals = f.A.feFormals;
       fun_names = [];
       } in
  let tenv' =
       { tenv with scope = scope'; } in
  let get_ssl (sstmt_list, _) = sstmt_list in
  let sslp = check_stmt_list tenv' f.A.feBody in scope'.variables <- List.rev
scope'.variables;
  let sfexpr = {
       S.sfeReturnType = f.A.feReturnType;
       S.sfeFormals = f.A.feFormals;
       S.sfeBody = get_ssl sslp;
} in S.SFunExpr(sfexpr)

(* numbers and string concatenation are supported *)
and check_math_binop se1 op se2 = function
       | (A.Int, A.Float)
       | (A.Float, A.Int)
       | (A.Float, A.Float) -> S.SBinop(se1, op, se2, A.Float)
       | (A.Int, A.Int) -> S.SBinop(se1, op, se2, A.Int)
       | _ -> raise(E.InvalidBinopEvalType)

and check_equal_binop se1 op se2 t1 t2 = (* no floats or functions *)
  if (t1 = A.Float || t2 = A.Float || t1 = A.Fun || t2 = A.Fun)
  then raise(E.InvalidBinopEvalType)
  else if t1 = t2 then S.SBinop(se1, op, se2, A.Bool)
       else raise(E.InvalidBinopEvalType)

and check_compare_binop se1 op se2 = function (* only numbers supported *)
       | (A.Int, A.Float)
       | (A.Float, A.Int)
```

```
      | (A.Float, A.Float)
      | (A.Int, A.Int)    -> S.SBinop(se1, op, se2, A.Bool)
      | _ -> raise(E.InvalidBinopEvalType)

and check_bool_binop se1 op se2 = function (* only boolean supported *)
      | (A.Bool, A.Bool)  -> S.SBinop(se1, op, se2, A.Bool)
      | _ -> raise(E.InvalidBinopEvalType)

and check_binop tenv e1 op e2 =
  let se1, _ = check_expr tenv e1 in
  let se2, _ = check_expr tenv e2 in
  let t1 = get_sexpr_type se1 in
  let t2 = get_sexpr_type se2 in
  match op with
        A.Add | A.Sub | A.Mult | A.Div | A.Mod      -> check_math_binop se1 op se2
(t1, t2)
      | A.Equal | A.Neq                         -> check_equal_binop se1 op se2 t1 t2
      | A.Less | A.Leq | A.Greater | A.Geq          -> check_compare_binop se1 op
se2 (t1, t2)
      | A.And | A.Or                                -> check_bool_binop se1 op se2
(t1, t2)
      | _ -> raise(E.InvalidBinaryOperator)

and check_unop tenv uop e =
  let se, _ = check_expr tenv e in
  let t = get_sexpr_type se in
  let check_num_unop uop se = function (* numbers only *)
      | A.Int -> S.SUnop(uop, se, A.Int)
      | A.Float -> S.SUnop(uop, se, A.Float)
      | _ -> raise(E.InvalidUnopEvalType) in
  let check_bool_unop uop se = function (* bool only *)
      | A.Bool -> S.SUnop(uop, se, A.Bool)
      | _ -> raise(E.InvalidUnopEvalType) in
  match uop with
        A.Neg             -> check_num_unop uop se t
      | A.Not             -> check_bool_unop uop se t
      | _          -> raise(E.InvalidUnaryOperator)

and check_assign tenv e1 e2 =
  let se1, _ = check_expr tenv e1 in
  let se2, _ = check_expr tenv e2 in
  let t1 = get_sexpr_type se1 in
  let t2 = get_sexpr_type se2 in
  if t1 = t2 (* assignment types must be exactly the same *)
     then S.SAssign(se1, se2, t1)
     else raise(E.AssignmentTypeMismatch(A.string_of_typ t1, A.string_of_typ t2))

and check_call tenv e args =
 let se, _ = check_expr tenv e in
 let str = A.string_of_expr e in
 if str <> "print" then
    (let r = Str.regexp "\\([A-Za-z_]+\\)" in    (* in currying examples, parser may
think the called function name in pokemonMotto("Ash")("!") is pokemonMotto("Ash") -->
parse to pokemonMotto and then check the function *)
    let num = try Str.search_forward r str 0 with
      | Not_found -> raise(E.UndefinedFunction(str)) in
    let rname = Str.matched_string str in
    let name = if rname = "main" then "mainuserentryfunctionformatchascriptlanguage"
else rname in
    let result = check_call_wrapper tenv name args in
    if result = "A.Void"
    then begin
      let resf = check_call_fun tenv name args in
      let get_fname fdecl = fdecl.A.fdFname in
      let res = get_fname resf in
      if res = "A.Void" then (
        let first_word = try Str.first_chars (A.string_of_expr e) 8 with
       | Invalid_argument(err) -> A.string_of_expr e in
          if first_word <> "function" then ( (* check for anonymous funs *)
            let t = check_call_fun_vdecls tenv name in
            if t = A.Void then raise(E.UndefinedFunction(name))
            else (
              let get_s (s,_) = s in
              let sargs = List.map (fun x -> get_s (check_expr tenv x)) args in
              S.SCallExpr(se, sargs, tenv.scope.return_type)
              ))
          else
            let get_s (s,_) = s in
            let sargs = List.map (fun x -> get_s (check_expr tenv x)) args in
            S.SCallExpr(se, sargs, tenv.scope.return_type)
            )
      else (let get_s (s,_) = s in
          let sargs = List.map (fun x -> get_s (check_expr tenv x)) args in
          S.SCallExpr(se, sargs, tenv.scope.return_type))
    end
    else
    let get_s (s,_) = s in
    let sargs = List.map (fun x -> get_s (check_expr tenv x)) args in
    S.SCallExpr(se, sargs, tenv.scope.return_type)
   ) else
    let get_s (s,_) = s in
    let sargs = List.map (fun x -> get_s (check_expr tenv x)) args in
    S.SCallExpr(se, sargs, tenv.scope.return_type)


(*******************
 * Check Expressions
 ********************)
and check_expr tenv = function
        A.IntLit i       -> S.SIntLit(i), tenv
      | A.BoolLit b      -> S.SBoolLit(b), tenv
      | A.FloatLit f            -> S.SFloatLit(f), tenv
      | A.CharLit c             -> S.SCharLit(c), tenv
      | A.StringLit s           -> S.SStringLit(s), tenv
      | A.FunExpr f      -> check_fexpr tenv f, tenv
      | A.Id i           -> check_id tenv i, tenv
      | A.Binop(e1,op,e2) -> check_binop tenv e1 op e2, tenv
      | A.Unop(uop, e)   -> check_unop tenv uop e, tenv
      | A.Assign(e1, e2)  -> check_assign tenv e1 e2, tenv
      | A.Noexpr         -> S.SNoexpr, tenv
      | A.CallExpr(e, args)     -> check_call tenv e args, tenv

(* for check_expr, add in check_call *)
and get_sexpr_type = function
        S.SIntLit(_)            -> A.Int
      | S.SBoolLit(_)           -> A.Bool
      | S.SFloatLit(_)   -> A.Float
      | S.SCharLit(_)           -> A.Char
      | S.SStringLit(_)  -> A.String
      | S.SFunExpr(_)           -> A.Fun
      | S.SId(_, t)      -> t
      | S.SBinop(_,_,_,t) -> t
      | S.SUnop(_,_,t)   -> t
      | S.SAssign(_,_,t)  -> t
      | S.SCallExpr(_,_,t)      -> t
      | S.SNoexpr        -> A.Void

(***********************
 * Check Statements
 ***********************)

(* check exprstmt by just checking expr *)
and check_expr_stmt tenv e =
  let sexpr, tenv = check_expr tenv e in
    S.SExprStmt(sexpr), tenv

(* check vdecl dup, vdecl type, and then add to symbol table *)
and check_vdecl tenv v =
  let get_v_expr (_,_, e) = e in (* helper to get expr from vdecl tuple *)
  let vexpr = get_v_expr v in
  let vsexpr, _ = check_expr tenv vexpr in
  let vstyp = get_sexpr_type vsexpr in
  let get_v_typ (t,_,_) = t in  (* helper to get typ of vdecl *)
  let vtyp = get_v_typ v in
  let get_v_name (_,n,_) = n in
  let vname = get_v_name v in
  if (vtyp = vstyp || vstyp = A.Void) (* check declared type of vdecl with its actual
type, or if vdecl hasn't been initialized yet *)
  then (tenv.scope.variables <- v:: tenv.scope.variables;
    S.SVarDecl(vtyp, vname, vsexpr), tenv)  (* add the vdecl to symbol table *)
  else raise(E.VariableDeclarationTypeMismatch(vname))
  and check_vdecl_st tenv v =
    let get_v_name (_,n,_) = n in
    let vname = get_v_name v in
      try List.find (fun x->(get_v_name x)=vname) tenv.scope.variables (* check to
see if local variable already exists *)
      with | Not_found    -> v
           |  _     -> raise(E.DuplicateLocal(vname))

and check_fdecl tenv f =
  try find_fdecl_name tenv.scope f.A.fdFname  (* check name dups *)
  with | Not_found ->
  let get_bind_string (_,s) = s in
  let formals_map = List.fold_left (fun m formal ->  (* check formal dups within
current function *)
    if StringMap.mem (get_bind_string formal) m
    then raise(E.DuplicateFormal(get_bind_string formal))
    else StringMap.add (get_bind_string formal) formal m) StringMap.empty
f.A.fdFormals in
  let nfname = if f.A.fdFname = "main" then
"mainuserentryfunctionformatchascriptlanguage" else f.A.fdFname in  (* in codegen,
the entire program is wrapped in a "main" function, which means technically no other
function the user writes can be called "main" without a duplicate function error.
Thus, we rename any user-defined function as this name. No other user program may
define this particular function; this is so a function called "main" can be defined
by user *)
  let new_fun = {
      A.fdReturnType = f.A.fdReturnType;
      A.fdFname = nfname;
      A.fdFormals = f.A.fdFormals;
      A.fdBody = f.A.fdBody;
  } in
  tenv.scope.fun_names <- tenv.scope.fun_names@[new_fun];
  match tenv.scope.parent with
    | Some(parent) -> ( let nfname = if f.A.fdFname = "main" then
"mainuserentryfunctionformatchascriptlanguage" else f.A.fdFname in
      let new_fun = {
       A.fdReturnType = f.A.fdReturnType;
       A.fdFname = nfname;
       A.fdFormals = f.A.fdFormals;
       A.fdBody = f.A.fdBody;
         } in
       parent.fun_names <- parent.fun_names@[new_fun]);
    let nfname = if f.A.fdFname = "main" then
"mainuserentryfunctionformatchascriptlanguage" else f.A.fdFname in
  let new_fun = {
       A.fdReturnType = f.A.fdReturnType;
       A.fdFname = nfname;
       A.fdFormals = f.A.fdFormals;
       A.fdBody = f.A.fdBody;
  } in
  let scope' = (* create a new scope *)
       {
       parent = Some(tenv.scope); (* parent may or may not exist *)
       variables = [(A.Fun, "print", A.Id("print"))];
       name = nfname;
       return_type = f.A.fdReturnType;
       formals = f.A.fdFormals;
       fun_names = [new_fun];
       } in
  let tenv' =
       { tenv with scope = scope'; } in
  let get_ssl (sstmt_list, _) = sstmt_list in
    let sslp = check_stmt_list tenv' f.A.fdBody in scope'.variables <- List.rev
scope'.variables;
    let sfdecl = {
      S.sfdReturnType = f.A.fdReturnType;
      S.sfdFname = nfname;
      S.sfdFormals = f.A.fdFormals;
      S.sfdBody = get_ssl sslp;
    } in
      S.SFunDecl(sfdecl), tenv (* return the original tenv after checking the fdecl
*)

and check_return tenv e =
  let sexpr, _ = check_expr tenv e in
    let t = get_sexpr_type sexpr in
      if t = tenv.scope.return_type then S.SReturn(sexpr), tenv
      else raise (E.ReturnTypeMismatch(A.string_of_typ t, A.string_of_typ
tenv.scope.return_type))

and check_if tenv e s1 s2 =
  let pred, _ = check_expr tenv e in
    let t = get_sexpr_type pred in
      let ifstmt, _ = check_stmt tenv s1 in
        let elsestmt, _ = check_stmt tenv s2 in
          if t = A.Bool then S.SIf(pred, ifstmt, elsestmt), tenv
          else raise(E.InvalidIfStatementCondition)

and check_for tenv e1 e2 e3 s =
  let restore = tenv.in_for in
    let tenv = update_env_context tenv true tenv.in_while in (* in for loop *)
    let se1, _ = check_expr tenv e1 in
    let se2, _ = check_expr tenv e2 in
    let se3, _ = check_expr tenv e3 in
    let forblock, _ = check_stmt tenv s in
    let typ = get_sexpr_type se2 in
    let result =  (* condition can be boolean or nothing, e.g. for(;;) *)
      if (typ = A.Bool || typ = A.Void) then S.SFor(se1, se2, se3, forblock)
      else raise(E.InvalidForStatementCondition) in
    let tenv = update_env_context tenv restore tenv.in_while (* out of for loop*)
    in result, tenv

and check_while tenv e s =
  let restore = tenv.in_while in
    let tenv = update_env_context tenv tenv.in_for true in
    let se, _ = check_expr tenv e in
    let typ = get_sexpr_type se in
    let whileblock, _ = check_stmt tenv s in
    let result =
      if (typ = A.Bool || typ = A.Void) then S.SWhile(se, whileblock)
      else raise(E.InvalidWhileStatementCondition) in
    let tenv = update_env_context tenv tenv.in_for restore
    in result, tenv

and check_stmt tenv = function
        A.Block sl       -> check_block tenv sl
      | A.ExprStmt e             -> check_expr_stmt tenv e
      | A.VarDecl v      -> check_vdecl_st tenv v; check_vdecl tenv v
      | A.FunDecl f      -> check_fdecl tenv f
      | A.Return e       -> check_return tenv e
      | A.If(e, s1, s2)  -> check_if tenv e s1 s2
      | A.For(e1, e2, e3, s)     -> check_for tenv e1 e2 e3 s
      | A.While(e,s)             -> check_while tenv e s
    | A.DoWhile(s,e)    -> check_while tenv e s

(* To be used as entrypoint for parsing ast, which is a stmt list *)
and check_stmt_list tenv stmt_list =
  let ref = ref(tenv) in        (* create a pointer to env *)
    let rec iter = function     (* dereference and modify env on each pass *)
      h::t ->
        let sstmt, tenv = check_stmt !ref h in
          ref := tenv; sstmt::(iter t)
    | [] -> []
    in
      let sstmt_list = (iter stmt_list), !ref in
      sstmt_list          (* parse each statement in the stmt list *)
```

```
(***********************
 * Root environment setup
 ***********************)
let root_symbol_table : symbol_table = {
  parent      = None;
  name        = "anon";
  variables   = [(A.Fun, "print", A.Id("print"))]; (* built-in function print *)
  return_type = A.Void;
  formals     = [];
  fun_names   = [];
}

let print_symbol_table : symbol_table = {
  parent      = Some(root_symbol_table);
  name        = "print";
  variables   = [(A.Fun, "print", A.Id("print"))];
  return_type = A.Void;
  formals     = [(A.String, "input")];
  fun_names   = [];
}

let root_env : translation_env = {
  scope = print_symbol_table;
  in_for = false;
  in_while = false;
}

(**********************
* Includes
**********************)
(* for each file in the include_stmt list, recursively append the contents of the
file
to a stmt list, and then append that stmt list to the program stmt list *)
let rec process_all_includes_in_level (includes_list : A.include_stmt list)
(stmt_list : A.stmt list)=
  (* returns the includes appended to the program stmts *)
  let rec process_include_single (stmts: A.stmt list) (A.Include(incl) :
A.include_stmt) =
    let file_in = open_in incl in
    let lexbuf = Lexing.from_channel file_in in
    let A.Program(inner_incstmts, inner_stmts) = Parser.program Scanner.token lexbuf
in
    let (include_and_stmts : A.stmt list) = process_all_includes_in_level
inner_incstmts inner_stmts in
    ignore(close_in file_in);
    (include_and_stmts@stmts)
  in List.fold_left process_include_single stmt_list includes_list
```

```
(***********************
 * Program entry point
 ***********************)
let check_ast ast = match ast with
    A.Program(includes, stmts) ->
        let prog_and_includes = process_all_includes_in_level includes stmts in
        let (sast, _) = check_stmt_list root_env prog_and_includes in sast
        | _ -> raise(E.InvalidCompilerArgument)

(* Testing *)
let test_ok (sast : S.sstmt list) = match sast with  (* with MatchaScript.ml *)
  _ -> "okay\n"
```

codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/

*)

open Sast

module L = Llvm
module A = Ast
module Ana = Analyzer
module E = Exceptions

module StringMap = Map.Make(String)

let context = L.global_context ()
let the_module = L.create_module context "MatchaScript"
let llbuilder = L.builder       context
let i32_t  = L.i32_type       context
let i8_t   = L.i8_type        context
let i1_t   = L.i1_type        context
let fl_t   = L.double_type     context
let str_t  = L.pointer_type (L.i8_type context)
let void_t = L.void_type      context

(* Table for function forward declarations *)
let fwd_decls_hashtbl : (string, (L.llvalue * sfdecl)) Hashtbl.t = Hashtbl.create 10

let ltype_of_typ = function
    A.Int -> i32_t
  | A.Float -> fl_t
  | A.Bool -> i1_t
  | A.Char -> i8_t
  | A.Void -> void_t
  | A.String -> str_t
;;

let gen_type exp = match exp with
    SIntLit(_) -> A.Int
  | SFloatLit(_) -> A.Float
  | SBoolLit(_) -> A.Bool
(*  | SCharLit _ -> A.Char *)
  | SStringLit(_) -> A.String
  | SId(_, typ) -> typ
  | SBinop(_,_,_, typ) -> typ
  | SUnop(_, _, typ) -> typ
  | SCallExpr(_,_,typ) -> typ
  | _ -> raise(Failure("llvm type of " ^ string_of_sexpr exp ^ " not yet supported"))
;;

(* Builtins *)
(* printf *)
let printf_func =
  let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  L.declare_function "printf" printf_t the_module
;;

(************************
Standard Library
************************)

(************************
Forward Declarations
************************)
let gen_func_fwd_decl (f : sfdecl) =
  let name = f.sfdFname
  and formal_types = Array.of_list (List.map (fun (t,_) -> ltype_of_typ t)
f.sfdFormals) in
  let ftype = L.function_type (ltype_of_typ f.sfdReturnType) formal_types in
  (* Adds the pair (name, (LLVM fwd_declaration, ocaml_sfdecl)) to the StringMap m *)
  if Hashtbl.mem fwd_decls_hashtbl name
  then raise(E.DuplicateFunction("duplicate function " ^ name ))
  else Hashtbl.add fwd_decls_hashtbl name (L.define_function name ftype the_module,
f)

let codegen_func_fwd_decls (sast : sstmt list) =
  let get_fdecls_for_fwd_decl_generation sstmt = match sstmt with
      SFunDecl(f) -> gen_func_fwd_decl f
    | _ -> ()
  in List.iter get_fdecls_for_fwd_decl_generation sast
;;

(************************
Function Definitions
************************)
let build_function_body f_build =
  let (the_function, _) = Hashtbl.find fwd_decls_hashtbl f_build.sfdFname in
  let llbuilder = L.builder_at_end context (L.entry_block the_function) in
  (* Construct the function's "locals": formal arguments and locally declared
variables.
    Allocate each on the stack, initialize their value, if appropriate, and remember
their
    values in the "locals" map *)
  let local_vars =
    let add_formal m (t, n) p = L.set_value_name n p;
      let local = L.build_alloca (ltype_of_typ t) n llbuilder in
      ignore (L.build_store p local llbuilder);
      StringMap.add n local m
   in

    let add_local m (t, n) =
      let local_var = L.build_alloca (ltype_of_typ t) n llbuilder in
      StringMap.add n local_var m
    in
    let f_formals = List.fold_left2 add_formal StringMap.empty f_build.sfdFormals
      (Array.to_list (L.params the_function)) in

    let f_locals =
      let extract_locals_from_fbody fbody =
        let handle_vdecl locals_list stmt = match stmt with
            SVarDecl(typ, id, expr) -> (
              (* check if this vdecl is already in the locals_list *)
              if List.exists (fun (ltyp, lid) -> lid = id) locals_list
              then raise(E.DuplicateLocal("duplicate local " ^ id ^ " in function " ^
f_build.sfdFname))
```

```
              else (typ, id) :: locals_list
              )
          | _ -> locals_list
        in
        List.fold_left handle_vdecl [] fbody  (* fbody is a stmt list *)
      in extract_locals_from_fbody f_build.sfdBody
   in
    (* extract locals from the stmt list of the function *)
    List.fold_left add_local f_formals f_locals
  in
  (* Return the value for a variable or formal argument *)
  let var_lookup n = try StringMap.find n local_vars
                  with Not_found -> raise (Failure "Variable not found")
  in
  (* Expressions *)
  let rec codegen_sexpr llbuilder = function
      SIntLit i -> L.const_int i32_t i
    | SFloatLit f -> L.const_float fl_t f
    | SBoolLit b -> L.const_int i1_t (if b then 1 else 0)
(*    | SCharLit c -> L.const_int i8_t c *)
    | SStringLit s -> L.build_global_stringptr s "string" llbuilder
    | SNoexpr -> L.const_int i32_t 0
    | SId(s, typ) -> L.build_load (var_lookup s) s llbuilder
    | SBinop (e1, op, e2, typ) ->
        let e1' = codegen_sexpr llbuilder e1
        and e2' = codegen_sexpr llbuilder e2
        and typ = typ in

        let int_ops e1 op e2 = match op with
            A.Add     -> L.build_add e1 e2 "addtmp" llbuilder
          | A.Sub     -> L.build_sub e1 e2 "subtmp" llbuilder
          | A.Mult    -> L.build_mul e1 e2 "multtmp" llbuilder
          | A.Div     -> L.build_sdiv e1 e2 "divtmp" llbuilder
          | A.Mod     -> L.build_srem e1 e2 "sremtmp" llbuilder
          | A.And     -> L.build_and e1 e2 "andtmp" llbuilder
          | A.Or      -> L.build_or e1 e2 "ortmp" llbuilder
          | A.Equal   -> L.build_icmp L.Icmp.Eq e1 e2 "eqtmp" llbuilder
          | A.Neq     -> L.build_icmp L.Icmp.Ne e1 e2 "neqtmp" llbuilder
          | A.Less    -> L.build_icmp L.Icmp.Slt e1 e2 "lesstmp" llbuilder
          | A.Leq     -> L.build_icmp L.Icmp.Sle e1 e2 "leqtmp" llbuilder
          | A.Greater -> L.build_icmp L.Icmp.Sgt e1 e2 "sgttmp" llbuilder
          | A.Geq     -> L.build_icmp L.Icmp.Sge e1 e2 "sgetmp" llbuilder
          | _         -> raise (Failure("unsupported operator for integer
arguments"))

        and float_ops e1 op e2 = match op with
            A.Add     -> L.build_fadd e1 e2 "flt_addtmp" llbuilder
          | A.Sub     -> L.build_fsub e1 e2 "flt_subtmp" llbuilder
          | A.Mult    -> L.build_fmul e1 e2 "flt_multmp" llbuilder
          | A.Div     -> L.build_fdiv e1 e2 "flt_divtmp" llbuilder
          | A.Mod     -> L.build_frem e1 e2 "flt_sremtmp" llbuilder
          | A.Equal   -> L.build_fcmp L.Fcmp.Oeq e1 e2 "flt_eqtmp" llbuilder
          | A.Neq     -> L.build_fcmp L.Fcmp.One e1 e2 "flt_neqtmp" llbuilder
          | A.Less    -> L.build_fcmp L.Fcmp.Ult e1 e2 "flt_lesstmp" llbuilder
          | A.Leq     -> L.build_fcmp L.Fcmp.Ole e1 e2 "flt_leqtmp" llbuilder
          | A.Greater -> L.build_fcmp L.Fcmp.Ogt e1 e2 "flt_sgttmp" llbuilder
          | A.Geq     -> L.build_fcmp L.Fcmp.Oge e1 e2 "flt_sgetmp" llbuilder
          | _         -> raise (Failure("unsupported operation for floating point
arguments"))
        in
        let match_types e1 = match gen_type e1 with
            A.Int | A.Bool -> int_ops e1' op e2'
          | A.Float -> float_ops e1' op e2'
          | _ -> raise(Failure("Invalid Binop types at " ^
                  string_of_sexpr e1 ^ " " ^ A.string_of_op op ^ " " ^
string_of_sexpr
                  e2 ^ ": Can only do Binop on Int, Bool, or Float"))
        in
        match_types e1
    | SUnop(op, e, typ) ->
        let e' = codegen_sexpr llbuilder e in
        let int_unops op e' = match op with
            A.Neg     -> L.build_neg e' "tmp" llbuilder
          | A.Not     -> L.build_not e' "tmp" llbuilder
        and float_unops op e' = match op with
            A.Neg     -> L.build_fneg e' "tmp" llbuilder
          | _         -> raise(Failure("Invalid " ^ A.string_of_uop op ^ " at " ^
string_of_sexpr e))
        in
        let match_types e = match gen_type e with
            A.Int -> int_unops op e'
          | A.Float -> float_unops op e'
          | _ -> raise(Failure("Invalid Unop type at " ^ string_of_sexpr e))
        in
        match_types e
    | SAssign (SId(s,_), e, t) ->
        let e' = codegen_sexpr llbuilder e in
        ignore (L.build_store e' (var_lookup s) llbuilder); e'
    | SCallExpr (SId("print", _), [e], typ) ->
        let int_format_str llbuilder = L.build_global_stringptr "%d\n" "fmt"
llbuilder;
        and float_format_str llbuilder = L.build_global_stringptr "%f\n" "fmt"
llbuilder;
        and char_format_str llbuilder = L.build_global_stringptr "%c\n" "fmt"
llbuilder;
        and str_format_str llbuilder = L.build_global_stringptr "%s\n" "fmt"
```

```
llbuilder in

        let format_str e_typ llbuilder = match e_typ with
            A.Int -> int_format_str llbuilder
          | A.Float -> float_format_str llbuilder
          | A.Char -> char_format_str llbuilder
          | A.String -> str_format_str llbuilder
          | _ -> raise (Failure "Invalid printf type")
        in

        let e' = codegen_sexpr llbuilder e
        and e_type = gen_type e in
        L.build_call printf_func [| format_str e_type llbuilder; e' |]
            "printf" llbuilder
    | SCallExpr (SId(fname, _), act, typ) ->
        let (llvm_fdef, ocaml_sfdecl) = Hashtbl.find fwd_decls_hashtbl fname in
        let actuals = List.rev (List.map (codegen_sexpr llbuilder) (List.rev act)) in
        let result = (match ocaml_sfdecl.sfdReturnType with A.Void -> ""
                                                          | _     -> fname ^ "_result")
in
        L.build_call llvm_fdef (Array.of_list actuals) result llbuilder
      in

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

  (* Statements *)
  let rec codegen_sstmt llbuilder = function
      SBlock sl -> List.fold_left codegen_sstmt llbuilder sl
    | SExprStmt e -> ignore (codegen_sexpr llbuilder e); llbuilder
    | SReturn e -> ignore (match f_build.sfdReturnType with
                              A.Void -> L.build_ret_void llbuilder
                            | _      -> L.build_ret (codegen_sexpr llbuilder e)
llbuilder); llbuilder
    | SVarDecl (typ, id, se) -> (* Assign the sexpr to id *)
        let e' = codegen_sexpr llbuilder se in
        ignore (L.build_store e' (var_lookup id) llbuilder); llbuilder
    | SIf (predicate, then_stmt, else_stmt) ->
        let bool_val = codegen_sexpr llbuilder predicate in
        let merge_bb = L.append_block context "merge" the_function in
        let then_bb = L.append_block context "then" the_function in
        add_terminal (codegen_sstmt (L.builder_at_end context then_bb) then_stmt)
        (L.build_br merge_bb);
        let else_bb = L.append_block context "else" the_function in
        add_terminal (codegen_sstmt (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 llbuilder);
        L.builder_at_end context merge_bb
    | SWhile (predicate, body) ->
        let pred_bb = L.append_block context "while" the_function in
        ignore (L.build_br pred_bb llbuilder);
        let body_bb = L.append_block context "while_body" the_function in
        add_terminal (codegen_sstmt (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 = codegen_sexpr pred_builder predicate in
        let merge_bb = L.append_block context "merge" the_function in
        ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
        L.builder_at_end context merge_bb
    | SDoWhile(body, pred) -> codegen_sstmt llbuilder
        ( SBlock [ SBlock [ body ] ; SWhile(pred, body) ] )
    | SFor (e1, e2, e3, body) -> codegen_sstmt llbuilder
        ( SBlock [SExprStmt e1 ; SWhile (e2, SBlock [body ; SExprStmt e3]) ] )
    | SFunDecl(_) -> llbuilder
  in

  (* Build the code for each statement in the function *)
  let llbuilder = codegen_sstmt llbuilder (SBlock f_build.sfdBody) in

  (* Add a return if the last block falls off the end *)
  add_terminal llbuilder (match f_build.sfdReturnType with
      A.Void -> L.build_ret_void
    | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
;;
let codegen_func_defs (sast : sstmt list) =
  let gen_func_def sstmt = match sstmt with
      SFunDecl(f) -> build_function_body f
    | _ -> ()
  in List.iter gen_func_def sast
;;

(*************************
Main
*************************)
let codegen_build_main (sast : sstmt list) =
  let wrap_sstmt_list (sl : sstmt list) =
    let sfdecl_main : sfdecl = {
      sfdReturnType = A.Int;
      sfdFname = "main";
      sfdFormals = [];
      (* Append a return statement to the body *)
      sfdBody = sl@[SReturn(SIntLit(0))];
      }
    in
    sfdecl_main;
  in
  let prog_main = wrap_sstmt_list sast in

  (* forward declare main function *)
  let _ = gen_func_fwd_decl prog_main in

  (* build_function_body main function *)
  let _ = build_function_body prog_main in
  ()
;;

(************************
Program entry point
************************)
let translate (sast : sstmt list) =
  let _ = codegen_func_fwd_decls sast in
  let _ = codegen_func_defs sast in
  let _ = codegen_build_main sast in
  the_module
;;
```

MatchaScript.ml

print(“Hello world!”);

```
(* Top-level of the MatchaScript compiler: scan & parse the input,
   check the resulting AST, generate LLVM IR, and dump the module *)

type action = Ast | Sast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);      (* Pretty-print the AST *)
                              ("-s", Sast);     (* Pretty-print the SAST *)
                              ("-l", LLVM_IR);  (* Generate LLVM, don't check *)
                              ("-c", Compile) ] (* Generate, check LLVM IR *)
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  let sast = Analyzer.check_ast ast in
  match action with
    Ast -> print_string (Ast.string_of_program ast)
  | Sast -> print_string (Sast.string_of_sprogram 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)
```

stdlib.ms

```
/***************
Math functions
***************/


/** Absolute value for integers **/
function int int_abs(int i) {
  if (i > 0) {
    return i;
  } else {
    return -1 * i;
  }
}


/** Absolute value for floats **/
function float float_abs(float i) {
  if (i > 0.0) {
    return i;
  }
  return -1.0 * i;
}


/** Get max of two ints **/
function int int_max(int a, int b){
  if (a > b){
    return a;
  } else {
    return b;
  }
}


/** Get max of two floats **/
/*
```

```
function float float_max(float a, float b){
  if (a > b){
    return a;
  } else {
    return b;
  }
}
*/

/** Get min of two ints **/
function int int_min(int a, int b){
  if (a < b){
    return a;
  } else {
    return b;
  }
}


/** Get min of two floats **/
/*
function float float_min(float a, float b){
  if (a < b){
    return a;
  } else {
    return b;
  }
}
*/

/** Raise an integer to a power **/
function int int_pow(int base, int pow){
  int i = 1;
  int result = base;
  while (i < pow){
    result = result * base;
    i = i + 1;
  }
  return result;
}


/** Raise a float to a power **/
function float float_pow(float base, int pow){
  int i = 1;
  float result = base;
  while (i < pow){
    result = result * base;
    i = i + 1;
  }
  return result;
}


/** Get ceiling of float **/
function float ceil(float f) {
  float remainder = f % 1.0;
  float new_f = f + 1.0 - remainder;
  return new_f;
}


/** Get floor of float, returns int **/
function float floor(float f) {
  float remainder = f % 1.0;
  float new_f = f - remainder;
  return new_f;
}


/** Round a float to the decimal place specified **/
function float round(float f) {
  float absOfF = float_abs(f);
  float floorOfF = floor(f);
  float result;

  /* Determine whether we are closer to the ceiling of f */
  if (absOfF - floorOfF >= 0.5) {
    result = floorOfF + 1.0;
  /* Else we are closer to the floor of f */
  } else {
    result = floorOfF;
  }
  return result;
}
```

Test Cases

```
int a = 1;
a = a + 1;
print(a);
```

```
test-assign-expr.ms
```

```
function void testBoolean(bool x, bool y) {
  if (x == y) {
    print("X is equal to Y");
  } else {
    print("X is not equal to Y");
  }

  bool z = true;
  if (x && z) {
    print("X and Z are both true");
  } else if (x || z) {
    print("X or Z is true");
  } else {
    print("X and Z are both false");
  }
}

testBoolean(true, false);
```

```
test-boolean.ms
```

```
int blargh = 4; //my dream is to fly over the rainbow so high
int niargh = 5; //hey!
print("Hello, world");
```

```
test-comments.ms
```

```
int a = 3;
int b;
b = a;
print(b); //should be getting a 3 here
```

```
test-declstmt-assign.ms
```

```
print(1.2+3.05);
```

```
test-float-plus-float.ms
```

```
float a = 1.1;
print(a);
```

test-floats.ms

```
int i;
for (i = 0; i < 5; i = i+1)
{
	print("take me to church");
}
```

test-for-all-mankind.ms

```
function int printFourWords(string a, string b, string c, string d) {
	print(a);
	print(b);
	print(c);
	print(d);
}

printFourWords("I","took","a","pill");
```

test-func-args-multiple.ms

```
function int returnFour() {
	return 4;
}

int four = returnFour();
print(four);
```

test-function-return-int.ms

```
function void main() {
  print("A real program.");
}

function int changeBool() {
	bool a = true;
  a = false;
  return 0;
}

main();
```

test-main-function.ms

```
function void myfunc() {
    int i = 10;
    int j = i % 3;
    print(j);

    int k = i % 2;
    print(k);

    int l = i % 5;
    print(l);
}

myfunc();
```

test-modulo.ms

```
int b;
int c;
int d;
int e;

int a = b = c = d = e = 5;

int i = 0;
while (i < 3) {
print(a);
print(b);
print(c);
print(d);
print(e);
i = i + 1;
}
```

test-multiple-assignment.ms

```
function void primeNumberChecker(int a) {
       print("Current number:");
```

```
        print(a);
        int counter = 2;
        int prime = 1;
        int current_a = a;
        int b_mod = 0;
        if (a == 1)
        {
                print("This number is prime");
        }
        if (a<1)
        {
                print("A number greater than 0 please");
        }
        if (a>1)
        {
                while (counter <= current_a)
                {
                        b_mod = current_a % counter;
                        if (b_mod ==0)
                        {
                                if (counter != a)
                                {
                                        prime = 0;
                                        print(counter);
                                        current_a = current_a / counter;
                                }
                                else
                                {
                                        counter = counter+1;
                                }
                        }
                        else
                        {
                                counter = counter+1;
                        }
                }

                if (prime==1)
                {
                        print("it's prime");
                }
        }

}
primeNumberChecker(5);
primeNumberChecker(27);
primeNumberChecker(43);
```

test-prime-fact.ms

```
function int primeNumberChecker(int a) {
	print(a);
	int counter = 2;
	int current = 1;
	int b_mod = 0;
	if (a == 1)
	{
		print("this is prime");
	}
	if (a<1)
	{
		print("A number greater than 0 please");
	}
	if (a>1)
	{
		while (counter < a)
		{
			b_mod = a % counter;
			if (b_mod ==0)
			{
				current = 0;
			}
			counter = counter + 1;
		}

		if (current==1)
		{
			print("it's prime");
		}
		else
		{
			print("it's not prime");
		}
		//print("this worked");
	}
}
primeNumberChecker(5);
primeNumberChecker(25);
```

test-prime-number-checker.ms

```
function int fib_rec (int n) {
	if (n == 0) {
		return 0;
	}
	if (n == 1) {
		return 1;
```

```
       }
       return fib_rec(n-1) + fib_rec(n-2);
}

int result = fib_rec(6);
print(result);
```

```
test-recursive-fib.ms
```

```
function int funcOne(string a, string b, string c, string d) {
       int e = 1;
       int f = 3;
       print("e is ");
       print(e);
}

function int funcTwo(string a, string b, string c, string d) {
       int e = 1;
       int f = 3;
}
```

```
test-same-locals-in-different-functions.ms
```

```
int a;
a = 5;

print(a);
```

```
test-separate-declaration-and-assignment.ms
```

```
function void aFunction(int q, float s) {
    print(q);
    print(s);
    print("hello there!");
}

function void my_main() {
    int a = 1;
    string c = "c";
    float d = 1.0;
    int e = a;
    print(e);
    aFunction(e, 2.2);
}

my_main();
```

test-vdecl-bind.ms

```
int i = 0;
do {
    print(i);
    i = i + 1;
}
while ( i < 4 );


int k = 1;
while (k < 5) {
    print(k);
    k = k + 1;
}
```

test-while-do-while.ms

```
function int whileLoop(int num) {
  int i = 0;
  if (num <= 0) {
    print("Next time run this program with a number higher than zero.");
    return -1;
  } else {
    while (i < num) {
      print(i);
      i = i + 1;
    }
  }
  return 0;
}
whileLoop(5);
```

test-while-loop.ms

FAIL TESTS

```
char a = 5; /* can't assign int to char */
```

fail-assignment-type-mismatch-char.ms

```
int a = "5";
```

fail-assignment-type-mismatch-int.ms

```
function void stringAssign() {
    string a = "I am a string but you won't see me";
    string b = "hey I am a string hello";
    a = b;
    print(a);
}

function void somethingElse() {
    print("Something else");
}

function void stringAssign() {
    string a = "I am a string but you won't see me";
    string b = "hey I am a string hello";
    a = b;
    print(a);
}
```

fail-duplicate-function-same-args.ms

```
int niceValue = 0;

function int nice() {
       int newNice = 10;
       niceValue = newNice;
       return 0;
}
int niceValue = -10;
```

fail-duplicate-global.ms

```
int niceValue = 0;

function int nice() {
       int newNice = 10;
       int newNice = 12;
```

```
        //niceValue = newNice;
        return 0;
}
```

fail-duplicate-local.ms

```
function void scoop(string flavor) {
        print(flavor);
}

function void main() {
        scoop("chocolate", "vanilla");
}
main();
```

fail-incorrect-num-args.ms

```
function void scoop(int one, int two) {
        print(one);
}

function void main() {
        scoop("chocolate", "vanilla");
}
main();
```

fail-incorrect-type-args.ms

```
function int main() {
        int a = 10;
        int b = 20;
        int wut = a @ b;
        return wut;
}
main();
```

fail-invalid-binary-op.ms

```
function void main() {
        string c = "a" / "b";
        print(c);
}
main();
```

```
fail-invalid-binary-eval.ms
```

```
print("hello" + 3);
```

```
fail-invalid-binop-string-int.ms
```

```
function int main() {
	int a = 10;
	int b = 20;
	if (^a) {
		a = b;
	}
	return a;
}
main();
```

```
fail-invalid-unary-op.ms
```

```
function void main() {
	string c = "not";
	if (!c) {
		print(c);
	}
}
main();
```

```
fail-invalid-unop-eval.ms
```

```
function int boo() {
	string hoo = "hoo";
	return hoo;
}
boo();
```

```
fail-return-type-mismatch.ms
```

```
function void main() {
	char a = 'a';
	hello();
}
```

```
fail-undefined-function.ms
```

```
function void main() {
        char b = 'b';
        print(hello());
}
```

```
fail-undefined-printed-function.ms
```

```
function void main() {
        print(a);
}
```

```
fail-unknown-id.ms
```

```
function void outer() {
        function void inner() {
                int a = 5;
        }
        print(a);      /* a not in scope */
}

outer();
```

```
fail-var-not-in-s
```

9 References

- Flow
    - https://github.com/facebook/flow
    - https://code.facebook.com/posts/1505962329687926/flow-a-new-static-type-checker-for-javascript/
- JavaScript: The Definitive Guide 6th Edition - David Flanagan
- JavaScript AST Explorer
    - http://astexplorer.net/