StockQ

Jesse Van Marter jjv2121
Ricardo Martinez rmm2222

Table of Contents

1. Introduction
2. Tutorial
3. Language Reference Manual
4. Project Plan
5. Architecture
6. Test Suite
7. Conclusion and Lessons Learned
8. Full Code Listing

Introduction

StockQ is a general purpose programming language that compiles to LLVM Intermediate Representation (LLVM IR). StockQ syntactically looks like a mix of C and Python. It is an imperative language with a program structure similar to Python (i.e. no main method required).

StockQ is a general purpose language that excels in giving the user tools to quickly run mathematical calculations, making the language potentially useful in financial operations. Since the language compiles to LLVM, it produces a very efficient bytecode that can be run on many different architecture. StockQ allows for lenient mathematical type interaction.

2. Tutorial

2.1 Environment

The environment for the compiler was an Ubuntu 16.04 virtual machine. All the testing was done on this VM. Opam was used to install and compile OCaml, as well as install the Llvm packages.

2.2 Hello World

In order to run the compiler, we need to run ‘make’ in the compiler directory. This will allow us to translate StockQ files to LLVM IR. The following program prints out ‘Hello World’ :

```
print(“Hello World”);
```

2.3 Basics

2.3.1 Program Structure

The structure of a StockQ program consists of functions and statements. The starting point are the statements run top-down. These statements can make calls to function in order to access them.

2.3.2 Variable Declaration

Variables are declared as follows:

```
int a;
float f;
```

A variable declaration may also initialize the variable:

```
int x = 84;
```

2.3.3 Print function

Printing is very easy in StockQ. Simply call the print() function on any primitive type (int, float, bool, string) or ID referencing a primitive type. For example:

```
print(5);
print(-4.3);
print("this is a string");

string mystr = "hi there!";
print(mystr);
```

2.4 Control Flow

2.4.1 If/Else

An if statement evaluates a boolean expression, and then executes a body of code depending on whether that expression was true or not. If statements may have an else attached to them, but this is not necessary.

```
bool b = true;
int a;
if (b) {
    print("Watermelons");
    a = 5;
} else {
    a = -10;
}
// a=5
```

2.4.2 For and While

For loops and

```
int x;
for(x=0; x < 5; x+=1) { ... }
```

While loops evaluate a boolean expression at the beginning of every loop (including the initial pass)

```
int x = 6;
while(x<=6.6){
        print(x);
        x+=1;
} // prints ‘6’ once
```

2.5 Functions

Functions are declared using the def keyword. A function declaration is also given a return type and a list of formal parameters. A function must return whatever return-type it was defined with. An example of a function declaration is:

```
def int add (int a, int b) {
        return a + b;
}
```

3. Language Reference Manual
StockQ

Jesse Van Marter jjv2121
Ricardo Martinez rmm2222

Language Reference Manual

Contents

Introduction **2**

Lexical Conventions **3**
- Tokens 3
- Identifiers 3
- Comments 3
- Whitespace 3
- Separators 3
- Reserved Words and Symbols 3
- Program Structure 4

Data Types **5**

Branch Control **6**

Expressions **8**
- Declaration and Assignment 8
- Arithmetic Operators 8
- Relational Operators 9
- Logic Operators 9

Functions **10**
- Function Example 10

Introduction

The financial industry has been a perfect space for the use of programming languages because of the heavy reliance on mathematical calculation and accuracy involved in gaining successful returns. We wanted to produce a lightweight, mathematical centered imperative language where users can quickly write and execute programs with low barriers to entry. We tried to make the syntax intuitive and easy to learn and wanted type interactions between ints and floats to be seamless.

Lexical Conventions

Tokens

Tokens are divided into identifiers, operators, separators, whitespace and reserved words.

Identifiers

Identifiers indicate function or variable names. StockQ identifiers are case-sensitive. Some Identifiers are reserved for keywords, listed below.
Identifiers must begin with a lowercase ‘a’-’z’ letter, but may include other listed characters (outside of whitespace) in the remainder of the name.

Comments

Both single and multi-line comments are supported in StockQ.

e.g. //this is a single line comment

/*this is a
multi-line comment/*

Whitespace

Whitespace includes spaces, tabs, and all types of newlines (CR, LF, CR LF) is ignored in StockQ.

Separators

;	statement delimiter
{}	function body/block separator
[]	indication of array
()	indication of list of argument(s)

Reserved Words and Symbols

Data Types

int
float

bool
string
void
true
false

Boolean Logic Operators

and
or
not

Branch Control and Loops

if
else
for
while
return

Functions

def
return

Built-in Functions

Print

Program Structure

StockQ programs contain statements and functions. These may be defined in any order. Program execution begins with the first statement in the file, and continues until the last statement has been read. Functions are called by statements. Functions themselves contain statements, and may also call functions (including themselves).

Data Types

Primitives:

int

- a string of numeric characters without a decimal point, and an optional sign character

float

- a string of numeric characters that can be before and/or after a decimal point, with an optional sign character

bool

- a binary variable where value can be either true or false
- the reserved words ‘true’ and ‘false’ are of type bool.
  e.g. bool b = true;

string

- a finite sequence of ASCII characters, enclosed in double quotes

e.g. string str = “we are StockQ”;

void

- a void type to indicate a function that doesn’t return anything
- can not be used as a variable type

Non-Primitives:

Array

- Array Declaration
  - Arrays are declared in the following method:
  - int[] arr;
  - float[] farr;

  - Arrays may also be initialized on the same line:
  - int[] arr = int[5];

- Array Initialization
  - Arrays are initialized with a size indicating how many elements they hold.
  - All arrays hold values of 0 until they are directly accessed and assigned.

- Array Access
  - Arrays are accessed with the [<index>] operator
  - int[] a = int[1];

- a[0] = 5;

Branch Control

if

- conditional if statement are followed by a boolean expression
- evaluates block immediately afterwards if boolean expression evaluates to true

e.g. if(x==2) {return true}

else

- An else statement may follow an if statement, will be executed immediately if other boolean expressions evaluate to false
- A statement list then follows

e.g. if(x==2) {return true} else {return false}

else if

- An else if statement may follow an if block, allowing for multiple logical blocks, of which only one will execute

e.g.

```
int x = 0;
if (x == 2) {print("hello");} else if (x < 2)
{ print("goodbye"); } else { print("a third option?"); }
// prints "goodbye"
```

for

The for statement allows for looping over a range of values. The format is as follows:

e.g. for (initialization; termination; update) { stmt }

The initialization begins the for statement and is executed only once (before the loop begins).

The termination is a boolean expression that is checked for before each loop. When it returns false, the loop terminates.

The update is an expression that occurs once after each loop, and should modify the variable(s) being checked for in the termination.

e.g. for(x = 0; x < 10; x += 1) { … }

while

The while statement is used for looping so long as a boolean expression inside of the while statement evaluates to true. The syntax is as follows:

e.g. while (expression) { … }

return

The return keyword is used to return a value inside of a function, and indicates the end of the current block.

e.g. def int myfunc () { return 5; }

Expressions

Declaration and Assignment

The general syntax for declaration is as follows:

- <type> <identifier>;
- examples: int x; float number; string name;
- Arrays can be declared as follows - <type>[] <identifier>;
  e.g. int a; int[] myArray;

The syntax for assignment:

- Any identifier being assigned must have been declared
- <identifier> = <value>
- Arrays need to be initialized with <type>[length]
  a=10; myArray = int[10]

These two operations can be combined in one line
e.g. int a = 10; float[] f = float[10]

Arithmetic Operators

+	addition
-	subtraction
*	multiplication
/	division
%	modulus
+=	increment
-=	decrement
*=	multiply and assignment
/=	division and assignment
%=	modulus and assignment

Addition | Subtraction | Multiplication | Division | Modulus

- These operators work on ints and floats, and any combination of the two.
- Binary operations using these operators on a mixture of floats and ints will cast the result internally to a float.

- The arithmetic operators followed by an equal sign '=' indicate binary operation, and then assign.

Relational Operators

==	logical equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Relational Operators

== and != can be used for all primitive data types (int, float, string, bool). The less than, less than or equal to, greater than, and greater than or equal to can only be used on int and float data types.

Logic Operators

and

- logical intersection of two expressions

e.g. true and true evaluates to 1 (true), all else returns false

or

- logical union of two expressions

e.g. true or false evaluates to 1 (true)

not

- logical negation of an expression

e.g. not true evaluates to 0

Example:

```
if (not true and (false or true)){
        print("This expression is true!");
else{
```

```
            print('This expression is false");
    }
// print "This expression is false"
```

Functions

Functions can be defined anywhere in a StockQ program, before or after they are called. Functions take a list of formals as well as a return type, which they must return a value of at the end of the block.

A program does not begin inside of a 'main' function, but no function named 'main' may be defined by the user.

def <type> myfunc(...){...}

- establishes a user-defined function that will return a value of type <type>

void

- used in the return type when the function does not return a value
- in this case, no return statement is required at the end of a function body

return

- caller of the function and indicates end of block. Nothing can follow return in a block

Function Example

```
e.g. def float max( float a, float b ) {
        //Statement list
        if( a > b ) {return a;} else {return b;}
}
```

4. Project Plan

4.1 Planning Process

The planning process began when our group was formed back in September. At the time, we had 4 members in our group and tried to meet weekly in order to have a good understanding of where our language was headed. At first, the goals of our weekly meetings consisted of developing a general understanding of compilers and how they worked. As the semester got going, we began to divvy up work and make deliverables for upcoming team or TA meetings.

4.2 Development Process

We tried using general project deadlines (scanner/parser finished, print “Hello world”) as guidelines for our progress and deliverables. With this in mind we tried to assign we tried to delegate parts of the project to team members for each upcoming meeting. Early on, our team made the mistake of being too lax on missed deadlines. As the semester progressed and these missed deadlines became regular occurrences for two of our team members, we became much more explicit with clearly specified deliverables at specified deadlines. When project progress was becoming noticeably delayed because of unfulfilled deliverables that were promised to be finished at a certain date, we realized we needed a drastic overhaul of our methods and decided to split from the group.

As a two person group, our development process continued in a mostly pair-programming manner, where the two of us would spend lots of time reviewing and implementing features together. Although this may not have been optimal in terms of production, it allowed for us to both gain a full understanding of each step in the overall process. We also benefitted, of course, from having more than one set of eyes to identify bugs and make optimizations.

4.6 Testing Process

For our testing process, we set up a test suite that would test our scanner, parser, and compiler. Tests would be written, and a pass of the test suite would ensure that all of the tests succeeded. This agile approach allowed us to identify problems that would arise after implementing new features when old tests would fail. New tests were added testing new functionalities as those functionalities were added in the compiler.

4.7 Team Roles

We found it beneficial for both team members to have a full understanding of all levels of our compiler, and therefore, both of us took on a little of each role. After becoming a two-member group, we found it was easier for us to organize among ourselves and communicate ideas, making the team manager role less important.

4.8 Software Development

The following software development tools were used in our project:

- Ubuntu 16.04 VM running in VirtualBox
- LLVM 3.8 installed via Opam
- OCaml compiled via Opam
- Git & Github for version control

5. Architecture

5.1 Overview

The compiler consists of the following parts:

- scanner.mll
- parser.mly
- ast.ml
- utils.ml
- semant.ml
- codegen.ml
- stockq.ml
- Stockq.sh

5.2 Scanner

The scanner is generated with ocamllex, and converts a sequence of characters into a sequence of tokens.

5.3 Parser

Our parser is generated with ocamlyacc, and builds the abstract syntax tree. Our parser does not take into account order when construction function and statement lists, which allows local variables to be declared, for example, after other statements.

5.5 Utils

Our utils file contains many debugging and printing functions, which are used in codegen.ml and stockq.ml. These functions are primarily used when a StockQ file is compiled with the -a (ast output) or -p (parser output) options.

5.6 Semant

The semant file takes in an ast and raises an exception if a problem is found with the ast. If the ast is a valid ast, the semant returns (). The semant uses StringMaps to keep track of function and variable declarations, and refers back to these to obtain types of expressions.

5.7 Codegen

Codegen takes in a valid ast and returns an LLVM module that is equivalent to the StockQ code interpreted. Our codegen does very little analysis, but does refer to utils to obtain types of expressions. It is this functionality that allows us to cast types internally, for example. The codegen file's entry point is a function checker, and although we do not have a main function in our language, we pass all the functionless statements in as a "fake-main" method.

5.8 Stockq

The stockq.ml file is the front-end for the compiler, offering the options to print out scanner output, parser output, ast output, or llvm output that is checked or not checked. The stockq.sh file is a bash file that takes in a file and compiles that. It is used in the test suite.

6. Test Suite

6.1 Testing Process

Tests were added as functionalities were added in the compiler. This allowed us to keep all of our tests as we progressed, and make sure that new functionality implementations would not break old tests. If old tests did break, we would be more informed as to how it broke.

6.2 Test Suite Capabilities

In our Makefile (both in the compiler and test directories), we included functionality that allowed us to perform every test present by running “make test”. We would run this after any significant change to our compiler files, which allowed us to keep on top of bugs and catch them as they came up. By default, ‘make test’ would run the scanner, parser, and compiler suites. These can be run individually as well, by executing ‘make scanner’, ‘make parser’, or ‘make compiler’ inside the test directory.

Each test consists of an .sq file and a .out file.

7. Conclusions & Lessons Learned

7.1 Jesse Van Marter

This project was a good lesson in the challenge of effective team management. Often times a team of four people working on one endeavor will not create as much as the sum of those same four people working individually. However, with ineffective team management, it is possible for that group to produce slower than one person if there is not a well-defined method for assigning tasks and evaluating deliverables. Important tasks may not be picked up and deliverables may not be up to standards, or even delivered at all. Without these methods in place, it is possible to think your project is in a more developed state than it actually is-- which can be much worse than simply knowing a project is behind. Setting more streamlined and easily testable/executable goals helped the process towards the end of the project.

7.2 Ricardo Martinez

Although I came to appreciate OCaml very much by the end of the semester, I realized that the understanding of the material didn't necessarily translate to an effective workflow within the team. If I could take this class again, I would start on the project much earlier in order to test out the working environment of the group and make any reparations with time to spare.

With regards to the project, I quickly became aware of all that goes into making a language, including many of the tasks that seemed easy at first, but became much more difficult upon deeper inspection. Although we didn't revolutionize the wheel, I am proud with what we set out to do and accomplished.

8. Full Code Listing

Scanner.mll

```
(* Ocamllex scanner for StockQ *)

{
   open Parser
   let lineno = ref 1
}

(* need to exclude newline for single line comments *)
let whitespace = [' ' '\t' '\r' '\n']
let alpha = ['a'-'z' 'A'-'Z']
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let digit = ['0'-'9']
let int = digit+
let float = (digit*) ['.'] digit+
let char = "'" ( ascii | digit ) "'
let string = "\"" ((ascii)* as s) "\""
let id = alpha (alpha | digit | '_')*


rule token = parse

(* white space *)
| whitespace    { token lexbuf }

(* comments *)
| "/*"          { comment lexbuf }
| "//"          { line_comment lexbuf }

(* symbols *)
| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
| '}'       { RBRACE }
| '['       { LBRACKET }
| ']'       { RBRACKET }
| ';'       { SEMI }
| ','       { COMMA }
```

```
(* operators *)
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '%'       { MODULO }
| "+="      { PLUSEQ }
| "-="      { MINUSEQ }
| "*="      { TIMESEQ }
| "/="      { DIVIDEEQ }
| "%="      { MODULOEQ }
| '='       { ASSIGN }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| '>'       { GT }
| ">="      { GEQ }
| "and"     { AND }
| "or"      { OR }
| "not"     { NOT }
| "true"    { TRUE }
| "false"   { FALSE }

(* branch control *)
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "while"   { WHILE }

(* function operation *)
| "return"  { RETURN }
| "def"     { DEF }

(* primitives *)
| "int"     { INT }
| "float"   { FLOAT }
| "bool"    { BOOL }
| "string"  { STRING }
| "void"    { VOID }

(* data types *)
```

```
| "struct"  { STRUCT }
| "array"   { ARRAY }

(* literals *)
| int as lxm       { INT_LITERAL(int_of_string lxm) }
| float as lxm     { FLOAT_LITERAL(float_of_string lxm) }
| id as lxm        { ID(lxm) }
| string           { STRING_LITERAL(s) }
| eof              { EOF }
| _ as char        { raise (Failure("illegal character " ^ Char.escaped char)) }

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

and line_comment = parse
    ['\n' '\r'] { token lexbuf }
|   _          { line_comment lexbuf }
```

Parser.mly

```
/* Ocamlyacc parser for StockQ */

%{
open Ast
%}

%token INT FLOAT BOOL STRING VOID TRUE FALSE
%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA
%token AND NOT OR PLUS MINUS TIMES DIVIDE ASSIGN MODULO
%token PLUSEQ MINUSEQ TIMESEQ DIVIDEEQ MODULOEQ
%token EQ NEQ LT LEQ GT GEQ
%token IF ELSE FOR WHILE RETURN DEF
%token ARRAY STRUCT
%token <int> INT_LITERAL
%token <float> FLOAT_LITERAL
%token <string> STRING_LITERAL
%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 PLUSEQ MINUSEQ
%left TIMES DIVIDE MODULO TIMESEQ DIVIDEEQ MODULOEQ
%right NOT NEG

%start program
%type <Ast.program> program

%%

program:
   decls EOF { $1 }

decls:
   /* nothing */   { [], [] }
  | decls fdecl     { (fst $1 @ [$2]), snd $1 }
  | decls stmt      { fst $1, (snd $1 @ [$2]) }

stmt_list:
   stmt              { [$1] }
  | stmt_list stmt      { $2 :: $1 }


fdecl:
   DEF datatype ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
   { {
      ftype = $2;
      fname = $3;
      formals = $5;
      body = List.rev $8
   } }

formals_opt:
```

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

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

datatype:
  | primitive   { Primitive($1) }
  | array_type  { $1 }

array_type:
    primitive LBRACKET brackets RBRACKET { Arraytype($1, $3) }

brackets:
    /* nothing */             { 1 }
  | brackets RBRACKET LBRACKET  { $1 + 1 }

primitive:
    INT         { Int }
|   FLOAT       { Float }
|   BOOL        { Bool }
|   STRING      { String }
|   VOID        { Void }
|   ARRAY       { Array }
|   STRUCT      { Struct }


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

```
|   datatype ID ASSIGN expr SEMI        { Local($1, $2, $4) }

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

expr:
   INT_LITERAL        { IntLiteral($1) }
|   FLOAT_LITERAL      { FloatLiteral($1) }
|   STRING_LITERAL     { StringLiteral($1) }
|   TRUE               { BoolLiteral(true) }
|   FALSE              { BoolLiteral(false) }
|   ID                 { Id($1) }
|   expr PLUS   expr    { Binop($1, Add, $3) }
|   expr MINUS  expr    { Binop($1, Sub, $3) }
|   expr TIMES  expr    { Binop($1, Mult, $3) }
|   expr DIVIDE expr    { Binop($1, Div, $3) }
|   expr MODULO expr    { Binop($1, Mod, $3) }
|   expr PLUSEQ expr    { Binop($1, Addeq, $3) }
|   expr MINUSEQ  expr  { Binop($1, Subeq, $3) }
|   expr TIMESEQ  expr  { Binop($1, Multeq, $3) }
|   expr DIVIDEEQ expr  { Binop($1, Diveq, $3) }
|   expr MODULOEQ expr  { Binop($1, Modeq, $3) }
|   expr EQ     expr    { Binop($1, Equal, $3) }
|   expr NEQ    expr    { Binop($1, Neq, $3) }
|   expr LT     expr    { Binop($1, Less, $3) }
|   expr LEQ    expr    { Binop($1, Leq, $3) }
|   expr GT     expr    { Binop($1, Greater, $3) }
|   expr GEQ    expr    { Binop($1, Geq, $3) }
|   expr AND    expr    { Binop($1, And, $3) }
|   expr OR     expr    { Binop($1, Or, $3) }
|   NOT expr            { Unop (Not, $2) }
|   MINUS expr          { Unop(Neg, $2) }
|   ID ASSIGN   expr    { Assign($1, $3) }
|   expr ASSIGN expr    { ArrayAssign($1, $3) }
|   ID LPAREN actuals_opt RPAREN    { Call($1, $3) }
|   primitive bracket_args RBRACKET { ArrayCreate(Primitive($1), List.rev $2) }
|   expr bracket_args RBRACKET      { ArrayAccess($1, List.rev $2) }
|   LPAREN expr RPAREN  { $2 }
```

```
bracket_args:
   |      LBRACKET expr                    { [$2] }
   |      bracket_args RBRACKET LBRACKET expr { $4 :: $1 }

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

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

Ast.ml

```
(* Abstract Syntax Tree for StockQ *)

type op = Add | Sub | Mult | Div | Mod | Equal |
        Addeq | Subeq | Multeq | Diveq | Modeq |
        Neq | Less | Leq | Greater | Geq | And | Or

type uop = Neg | Not

type primitive = Int | Float | Bool | Void | Stock | Order |
              Portfolio | String | Array | Struct

type datatype =
  | Primitive of primitive
  | Arraytype of primitive * int

type bind = datatype * string

type expr =
    IntLiteral of int
  | FloatLiteral of float
  | StringLiteral of string
  | BoolLiteral of bool
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
```

```
  | Assign of string * expr
  | ArrayAssign of expr * expr
  | Call of string * expr list
  | ArrayCreate of datatype * expr list
  | ArrayAccess of expr * expr list
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Local of datatype * string * expr

type func_decl = {
        ftype : datatype;
    fname : string;
    formals : bind list;
    body : stmt list;
  }

type program = func_decl list * stmt list
```

Semant.ml

```
(* Semantic checking for the StockQ compiler *)

open Ast

module U = Utils

module StringMap = Map.Make(String)

(* Semantic checking of a program. Returns void if successful,
   throws an exception if something is wrong.
*)

let check (functions, stmts) =
```

```
let main_func_name = "main" in

(* Raise an exception if the given list has a duplicate *)
let report_duplicate exceptf list =
  let rec helper = function
    | n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
    | _ :: t -> helper t
    | [] -> ()
  in helper (List.sort compare list)
in

(* Raise an exception if a given binding is to a void type *)
let check_not_void exceptf = function
  | (Primitive(Void), n) -> raise (Failure (exceptf n))
  | _ -> ()
in

(* Raise an exception if the given rvalue type cannot be assigned to
   the given lvalue type *)
let check_assign lvaluet rvaluet err =
  if lvaluet == rvaluet then lvaluet else raise err
in

let prim_of_dt = function
  | Primitive(p)      -> p
  | Arraytype(p, _)   -> p
in

(**** Checking Functions ****)

if List.mem "print" (List.map (fun fd -> fd.fname) functions)
then raise (Failure ("function print may not be defined")) else ();

if List.mem main_func_name (List.map (fun fd -> fd.fname) functions)
then raise (Failure ("function main may not be defined")) else ();

report_duplicate (fun n -> "duplicate function " ^ n)
  (List.map (fun fd -> fd.fname) functions);
```

```
(* Function declaration for a named function *)
let print_type = Primitive(Int) in
let built_in_decls = StringMap.singleton "print"
  { ftype = Primitive(Void);
    fname = "print";
    formals = [(print_type, "x")];
    body = [] }
in

let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
                     built_in_decls functions
in

let function_decl s = try StringMap.find s function_decls
  with Not_found -> raise (Failure ("unrecognized function " ^ s))
in

let check_function func =
  (* check formals *)
  List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^
    " in " ^ func.fname)) func.formals;

  report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^ func.fname)
    (List.map snd func.formals);

  (* check locals *)
  let locals =
    let rec get_locals mylocals = function
      | [] -> mylocals
      | [Local (t, s, _)] -> get_locals [(t, s)] []
      | Local (t, s, _) :: r -> get_locals ((t, s) :: mylocals) r
      | _ :: r -> get_locals mylocals r
    in
    get_locals [] (if func.fname = main_func_name then stmts else func.body)
  in

  List.iter (check_not_void (fun n -> "illegal void local " ^ n ^
    " in " ^ func.fname)) locals;
```

```
  report_duplicate (fun n -> "duplicate local " ^ n ^ " in " ^ func.fname)
    (List.map snd locals);

  (* Datatype of each variable (formal, or local *)
  let symbols =
    List.fold_left (fun m (t, n) -> StringMap.add n (prim_of_dt t) m)
      StringMap.empty (func.formals @ locals)
  in

  let type_of_identifier s =
    try StringMap.find s symbols
    with Not_found -> raise (Failure ("undeclared identifier " ^ s))
  in

  (* Return the type of an expression or throw an exception *)
  let rec expr = function
    | IntLiteral _    -> Int
    | FloatLiteral _  -> Float
    | BoolLiteral _   -> Bool
    | StringLiteral _ -> String
    | Id s            -> type_of_identifier s
    | Binop(e1, op, e2) as e ->
       let t1 = expr e1
       and t2 = expr e2 in
       (match op with
         (* ops on Int/Float *)
         | Add | Sub | Mult | Div | Mod when t1 = Int && t2 = Int -> Int
         | Add | Sub | Mult | Div | Mod when t1 = Int && t2 = Float -> Float
         | Add | Sub | Mult | Div | Mod when t1 = Float && t2 = Int -> Float
         | Add | Sub | Mult | Div | Mod when t1 = Float && t2 = Float -> Float

         | Addeq | Subeq | Multeq | Diveq | Modeq when t1 = Int && t2 = Int -> Int
         | Addeq | Subeq | Multeq | Diveq | Modeq when t1 = Int && t2 = Float ->
Float
         | Addeq | Subeq | Multeq | Diveq | Modeq when t1 = Float && t2 = Int ->
Float
         | Addeq | Subeq | Multeq | Diveq | Modeq when t1 = Float && t2 = Float ->
Float
```

```
          (* ops on Bools *)
          | Equal | Neq when t1 = t2 -> Bool
          | Equal | Neq when (t1 = Int && t2 = Float) || (t1 = Float && t2 = Int) -> Bool
          | Less | Leq | Greater | Geq when t1 = t2 -> Bool
          | Less | Leq | Greater | Geq when (t1 = Int && t2 = Float) || (t1 = Float &&
t2 = Int) -> Bool

          | And | Or when t1 = Bool && t2 = Bool -> Bool
          | _ -> raise (Failure ("illegal binary operator " ^
              U.string_of_primitive t1 ^ " " ^ U.string_of_op op ^ " " ^
              U.string_of_primitive t2 ^ " in " ^ U.string_of_expr e))
        )
      | Unop(op, e) as ex -> let t = expr e in
        (match op with
          | Neg when t = Int -> Int
          | Neg when t = Float -> Float
          | Not when t = Bool -> Bool
          | _ -> raise (Failure ("illegal unary op " ^ U.string_of_uop op ^
                U.string_of_primitive t ^ " in " ^ U.string_of_expr ex))
        )
      | Assign(var, e) as ex ->
        let lt = type_of_identifier var
        and rt = expr e in
        check_assign lt rt (Failure ("illegal assignment " ^
          U.string_of_primitive lt ^ " = " ^ U.string_of_primitive rt ^
          " in " ^ U.string_of_expr ex))
      | ArrayAssign(e1, e2) -> expr e1
      | Call(fname, actuals) as call ->
        if fname = "print" then
          if List.length actuals != 1 then
            raise (Failure ("expecting 1 argument in print"))
          else
            let actual = List.hd actuals in
            let actual_t = expr actual in
            (match actual_t with
              | Int | Float | Bool | String -> actual_t
              | _ -> raise (Failure ("expecting int or float in print"))
            )
```

```
      else
        let fd = function_decl fname in
        if List.length actuals != List.length fd.formals then
          raise (Failure ("expecting " ^ string_of_int
            (List.length fd.formals) ^ " arguments in " ^
            U.string_of_expr call))
        else
          List.iter2 (fun (ft, _) e ->
            let ft = prim_of_dt ft in
            let et = expr e in
            ignore (check_assign ft et
              (Failure ("illegal actual argument found " ^
                U.string_of_primitive et ^ " expected " ^
                U.string_of_primitive et ^ " in " ^ U.string_of_expr e))))
            fd.formals actuals;
          prim_of_dt fd.ftype
  | ArrayCreate (t, el) -> prim_of_dt t
  | ArrayAccess (e, el) -> expr e
  | Noexpr -> Void
in

let check_bool_expr e = if expr e != Bool
  then raise (Failure ("expected Boolean expression in " ^ U.string_of_expr e))
  else ()
in

(* verify a statement or throw an exception *)
let rec stmt = function
  | Block sl ->
      let rec check_block = function
        | [] -> ()
        | [Return _ as s] -> stmt s
        | Return _ :: _ -> raise (Failure ("nothing may follow a return"))
        | Block sl :: ss -> check_block (sl @ ss)
        | s :: ss -> stmt s; check_block ss
      in
      check_block sl
  | Expr e -> ignore (expr e)
  | Return e ->
```

```
      let t = expr e in
      if t = (prim_of_dt func.ftype) then ()
      else
        raise (Failure ("return gives " ^ U.string_of_primitive t ^
          " expected " ^ U.string_of_primitive (prim_of_dt func.ftype) ^
          " in " ^ U.string_of_expr e))
    | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt b2
    | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
                             ignore (expr e3); stmt st
    | While(p, s) -> check_bool_expr p; stmt s
    | Local(_, var, e) ->
      ignore (expr e);
      (match (expr e) with
        | Void -> ()
        | _ ->
          let lt = type_of_identifier var
          and rt = expr e in
          ignore (check_assign lt rt (Failure ("illegal assignment " ^
            U.string_of_primitive lt ^ " = " ^ U.string_of_primitive rt ^
            " in " ^ U.string_of_expr e)))
      )
  in

  stmt (if func.fname = main_func_name then Block(stmts) else Block(func.body))

in
List.iter check_function functions;
let mainfdecl = { ftype = Primitive(Int); fname = main_func_name;
                  formals = []; body = [] } in
check_function mainfdecl
```

Utils.ml

```
open Ast
open Parser

module StringMap = Map.Make(String)
```

```
let string_of_op = function
  | Add     -> "Add"
  | Sub     -> "Sub"
  | Mult    -> "Mult"
  | Div     -> "Div"
  | Mod     -> "Mod"
  | Addeq   -> "Addeq"
  | Subeq   -> "Subeq"
  | Multeq  -> "Multeq"
  | Diveq   -> "Diveq"
  | Modeq   -> "Modeq"
  | Equal   -> "Equal"
  | Neq     -> "Neq"
  | Less    -> "Less"
  | Leq     -> "Leq"
  | Greater -> "Greater"
  | Geq     -> "Geq"
  | And     -> "And"
  | Or      -> "Or"

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

let string_of_primitive = function
  | Int     -> "int"
  | Float   -> "float"
  | Bool    -> "bool"
  | Void    -> "void"
  | Stock   -> "stock"
  | Order   -> "order"
  | Portfolio -> "portfolio"
  | String  -> "string"
  | Array   -> "array"
  | Struct  -> "struct"

let rec print_brackets = function
  | 1 -> "[]"
```

```
  | i -> "[]" ^ print_brackets (i - 1)

let string_of_datatype = function
  | Primitive(p)    -> string_of_primitive p
  | Arraytype(p, i) -> string_of_primitive p ^ print_brackets i

let string_of_id = function
  | Id(s)   -> s
  | _       -> ""

let rec string_of_expr = function
  | IntLiteral(i)       -> "IntLiteral(" ^ string_of_int i ^ ")"
  | FloatLiteral(f)     -> "FloatLiteral(" ^ string_of_float f ^ ")"
  | StringLiteral(s)    -> "StringLiteral(" ^ s ^ ")"
  | BoolLiteral(b)      -> "BoolLiteral(" ^ string_of_bool b ^ ")"
  | Id(s)               -> "Id(" ^ s ^ ")"
  | Binop(e1, op, e2)   ->
     let v1 = string_of_expr e1
     and v2 = string_of_expr e2
     and oper = string_of_op op in
     "Binop(" ^ v1 ^ ", " ^ oper ^ ", " ^ v2 ^ ")"
  | Unop(uop, e)        -> "Unop(" ^ string_of_uop uop ^ ", " ^
                           string_of_expr e ^ ")"
  | Assign(s, e)        -> "Assign(" ^ s ^ ", " ^ string_of_expr e ^ ")"
  | Call(s, el)         -> "Call(" ^ s ^ ", " ^
                          String.concat ", " (List.map string_of_expr el) ^
                          ")"
  | ArrayCreate(t, el)  -> "ArrayCreate"
  | Noexpr              -> "Noexpr"

let rec string_of_stmt = function
  | Block(sl)       ->  "Block(" ^
                       String.concat ", " (List.map string_of_stmt sl) ^ ")"
  | Expr(e)         ->  string_of_expr e
  | Return(e)       ->  "Return(" ^ string_of_expr e ^ ")"
  | If(e, s1, s2)   ->  "If(" ^ string_of_expr e ^ ") { " ^ string_of_stmt s1 ^
                       " } Else { " ^ 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 ^ " }"
  | Local(dt, s, e) -> "Local(" ^ string_of_datatype dt ^ ", " ^ s ^ ", " ^
                     string_of_expr e ^ ")"

(* let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n" *)

let string_of_func func =
  "def " ^ string_of_datatype func.ftype ^ " " ^ func.fname ^ " (" ^
  String.concat ", " (List.map snd func.formals) ^ ") {\n\t" ^
  String.concat "\n\t" (List.map string_of_stmt func.body) ^
  "\n}"

let rec string_of_program stor = function
  | ([], [])         -> String.concat "\n" (List.rev stor) ^ "\n"
  | ([], stmt :: tl) -> string_of_program (string_of_stmt stmt :: stor) ([], tl)
  | (func :: tl, []) -> string_of_program (string_of_func func :: stor) (tl, [])

    (* return all functions first, then statements *)
  | (func :: ftl, stmts) ->
      string_of_program (string_of_func func :: stor) (ftl, stmts)

let string_of_token = function
  | LPAREN  -> "LPAREN" | RPAREN    -> "RPAREN"
  | LBRACE  -> "LBRACE" | RBRACE    -> "RBRACE"
  | SEMI    -> "SEMI"   | COMMA     -> "COMMA"

  | PLUS    -> "PLUS"   | MINUS     -> "MINUS"
  | TIMES   -> "TIMES"  | DIVIDE    -> "DIVIDE"
  | MODULO  -> "MODULO"

  | PLUSEQ  -> "PLUSEQ"     | MINUSEQ   -> "MINUSEQ"
  | TIMESEQ -> "TIMESEQ"    | DIVIDEEQ  -> "DIVIDEEQ"
  | MODULOEQ -> "MODULOEQ"

  | ASSIGN  -> "ASSIGN"
```

```
  | EQ      -> "EQ"      | NEQ       -> "NEQ"
  | LT      -> "LT"      | LEQ       -> "LEQ"
  | GT      -> "GT"      | GEQ       -> "GEQ"
  | AND     -> "AND"     | OR        -> "OR"
  | NOT     -> "NOT"

  | LBRACKET    -> "LBRACKET"    | RBRACKET  -> "RBRACKET"

  | IF      -> "IF"
  | ELSE    -> "ELSE"
  | FOR     -> "FOR"
  | WHILE   -> "WHILE"
  | RETURN  -> "RETURN"
  | DEF     -> "DEF"

  | INT     -> "INT"     | FLOAT     -> "FLOAT"
  | BOOL    -> "BOOL"    | VOID      -> "VOID"
  | TRUE    -> "TRUE"    | FALSE     -> "FALSE"
  | STRUCT  -> "STRUCT" | ARRAY     -> "ARRAY"
  | STRING  -> "STRING"

  | INT_LITERAL _       -> "INT_LITERAL"
  | FLOAT_LITERAL _     -> "FLOAT_LITERAL"
  | ID _                -> "ID"
  | STRING_LITERAL _    -> "STRING_LITERAL"
  | EOF                 -> "EOF"

(* returns ast type of expr *)
let type_of_expr func all_funcs =

  let check_assign lvaluet rvaluet err =
    if lvaluet == rvaluet then lvaluet else raise err
  in

  let prim_of_dt = function
    | Primitive(p)      -> p
    | Arraytype(p, _)   -> p
  in
```

```
let print_type = Primitive(Int) in
let built_in_decls = StringMap.singleton "print"
  { ftype = Primitive(Void);
    fname = "print";
    formals = [(print_type, "x")];
    body = [] }
in

let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
                     built_in_decls all_funcs
in

let function_decl s = try StringMap.find s function_decls
  with Not_found -> raise (Failure ("unrecognized function " ^ s))
in

let locals =
  let rec get_locals mylocals = function
    | [] -> mylocals
    | [Local (t, s, _)] -> get_locals [(t, s)] []
    | Local (t, s, _) :: r -> get_locals ((t, s) :: mylocals) r
    | _ :: r -> get_locals mylocals r
  in
  get_locals [] func.body
in

(* Datatype of each variable (formal, or local *)
let symbols =
  List.fold_left (fun m (t, n) -> StringMap.add n (prim_of_dt t) m)
    StringMap.empty (func.formals @ locals)
in

let type_of_identifier s =
  try StringMap.find s symbols
  with Not_found -> raise (Failure ("undeclared identifier " ^ s))
in

let rec expr = function
```

```
  | IntLiteral _    -> Int
  | FloatLiteral _  -> Float
  | BoolLiteral _   -> Bool
  | StringLiteral _ -> String
  | Id s            -> type_of_identifier s
  | Binop(e1, op, e2) as e ->
     let t1 = expr e1
     and t2 = expr e2 in
     (match op with
        | Add | Sub | Mult | Div | Mod when t1 = Int && t2 = Int -> Int
        | Add | Sub | Mult | Div | Mod when t1 = Int && t2 = Float -> Float
        | Add | Sub | Mult | Div | Mod when t1 = Float && t2 = Int -> Float
        | Add | Sub | Mult | Div | Mod when t1 = Float && t2 = Float -> Float

        | Addeq | Subeq | Multeq | Diveq | Modeq when t1 = Int && t2 = Int -> Int
        | Addeq | Subeq | Multeq | Diveq | Modeq when t1 = Int && t2 = Float ->
Float
        | Addeq | Subeq | Multeq | Diveq | Modeq when t1 = Float && t2 = Int ->
Float
        | Addeq | Subeq | Multeq | Diveq | Modeq when t1 = Float && t2 = Float ->
Float

       | Equal | Neq when t1 = t2 -> Bool
       | Equal | Neq when (t1 = Int && t2 = Float) || (t1 = Float && t2 = Int) -> Bool
       | Less | Leq | Greater | Geq when t1 = t2 -> Bool
       | Less | Leq | Greater | Geq when (t1 = Int && t2 = Float) || (t1 = Float && t2
= Int) -> Bool
       | And | Or when t1 = Bool && t2 = Bool -> Bool
       | _ -> raise (Failure ("illegal binary operator " ^
          string_of_primitive t1 ^ " " ^ string_of_op op ^ " " ^
          string_of_primitive t2 ^ " in " ^ string_of_expr e))
     )
  | Unop(op, e) as ex -> let t = expr e in
     (match op with
       | Neg when t = Int -> Int
       | Neg when t = Float -> Float
       | Not when t = Bool -> Bool
       | _ -> raise (Failure ("illegal unary op " ^ string_of_uop op ^
            string_of_primitive t ^ " in " ^ string_of_expr ex))
```

```
      )
  | Assign(var, e) as ex ->
      let lt = type_of_identifier var
      and rt = expr e in
      check_assign lt rt (Failure ("illegal assignment " ^
        string_of_primitive lt ^ " = " ^ string_of_primitive rt ^
        " in " ^ string_of_expr ex))
  | Call(fname, actuals) as call ->
      if fname = "print" then
        if List.length actuals != 1 then
          raise (Failure ("expecting 1 argument in print"))
        else
          let actual = List.hd actuals in
          let actual_t = expr actual in
          (match actual_t with
            | Int | Float | Bool | String -> actual_t
            | _ -> raise (Failure ("expecting int or float in print"))
          )
      else
        let fd = function_decl fname in
        if List.length actuals != List.length fd.formals then
          raise (Failure ("expecting " ^ string_of_int
            (List.length fd.formals) ^ " arguments in " ^
            string_of_expr call))
        else
          List.iter2 (fun (ft, _) e ->
            let ft = prim_of_dt ft in
            let et = expr e in
            ignore (check_assign ft et
              (Failure ("illegal actual argument found " ^
                string_of_primitive et ^ " expected " ^
                string_of_primitive et ^ " in " ^ string_of_expr e))))
            fd.formals actuals;
          prim_of_dt fd.ftype
  | ArrayAccess(e, el) -> expr e
  | Noexpr -> Void
 in expr
```

Codegen.ml

```
open Ast

module L = Llvm
module U = Utils

module StringMap = Map.Make(String)

let translate (functions, stmts) =
  let context = L.global_context () in
  let the_module = L.create_module context "StockX"
  and i32_t      = L.i32_type    context
  and i8_t       = L.i8_type     context in
  let str_t      = L.pointer_type i8_t
  and i1_t       = L.i1_type     context
  and void_t     = L.void_type   context
  and double_t   = L.double_type context in

  (* defines what the func_name is of the "function-less" stmts *)
  let main_func_name = "main" in

  let ltype_of_typ = function
    | Int     -> i32_t
    | Float   -> double_t
    | Bool    -> i1_t
    | String  -> str_t
    | Void    -> void_t
    | _ -> i32_t
  in

  let ast_type_of_datatype = function
    | Primitive(p)    -> p
    | Arraytype(p, _) -> p
  in

  let type_of_datatype = function
    | Primitive(p)    -> ltype_of_typ p
    | Arraytype(p, _) -> ltype_of_typ p
  in
```

```
(* Declare printf(), which the print built-in function will call *)
let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func = L.declare_function "printf" printf_t the_module in

let function_decls =
  let function_decl m fdecl =
    let name = fdecl.fname and formal_types =
      Array.of_list (List.map (fun (t, _) -> type_of_datatype t) fdecl.formals)
    in
    let ftype = L.function_type (type_of_datatype fdecl.ftype) formal_types
    in
    StringMap.add name (L.define_function name ftype the_module, fdecl) m
  in
  List.fold_left function_decl StringMap.empty functions
in

(* Fill in the body of the given function
 * 't' indicates whether we are working on
 * a user-defined function, or the "main"
 * function that encompasses all of the
 * statements outside of any functions.
 * t=0 -> functions, t=1 -> statements
 *)
let build_function_body fdecl =

  let type_of_expr e = U.type_of_expr fdecl functions e in
  let the_function =
    if fdecl.fname = main_func_name then
      let fty = L.function_type i32_t [| |] in
      L.define_function "main" fty the_module
    else
      let get_1_2 (a, _) = a in
      get_1_2 (StringMap.find fdecl.fname function_decls)
  in

  let builder = L.builder_at_end context (L.entry_block the_function) in

  (* print formatters *)
  let str_format_str   = L.build_global_stringptr "%s\n" "fmt" builder in
```

```
let int_format_str   = L.build_global_stringptr "%d\n" "fmt" builder in
let float_format_str = L.build_global_stringptr "%f\n" "fmt" builder in

let rec get_ptr_type datatype = match datatype with
  | Arraytype(t, 0)   -> type_of_datatype (Primitive(t))
  | Arraytype(t, 1)   -> L.pointer_type (type_of_datatype (Primitive(t)))
  | Arraytype(t, i)   -> L.pointer_type (get_ptr_type (Arraytype(t, (i-1))))
  | _                 -> raise (Failure ("Invalid Array Pointer Type"))
in

let get_type datatype = match datatype with
  | Arraytype(t, i) -> get_ptr_type (Arraytype(t, (i)))
  | Primitive(p)    -> type_of_datatype datatype
  | d -> raise (Failure ("Unable to match array type"))
in

(* construct the function's "locals" *)
let local_vars =

  (* get formals first, if this is a function decl *)
  let formals =
    if fdecl.fname = main_func_name then
      StringMap.empty
    else
      let add_formal m (t, n) p = L.set_value_name n p;
        let t = match t with
          | Primitive(_) -> type_of_datatype t
          | Arraytype(_, _) -> get_type t
        in
        let local = L.build_alloca t n builder in
        ignore (L.build_store p local builder);
        StringMap.add n local m
      in
      List.fold_left2 add_formal StringMap.empty fdecl.formals
        (Array.to_list (L.params the_function))
  in

  let add_local m (t, n) =
    let t = match t with
```

```
      | Primitive(_) -> type_of_datatype t
      | Arraytype(_, _) -> get_type t
    in
    let local_var = L.build_alloca t n builder
    in StringMap.add n local_var m
  in

  let rec get_locals mylocals = function
    | [] -> mylocals
    | [Local (t, s, _)] -> get_locals [(t, s)] []
    | Local (t, s, _) :: r -> get_locals ((t, s) :: mylocals) r
    | _ :: r -> get_locals mylocals r
  in
  List.fold_left add_local formals
    (get_locals [] (if fdecl.fname = main_func_name then stmts else fdecl.body))
in

(* return the value for a variable *)
let lookup n = StringMap.find n local_vars in

(* Construct code for expressions in a function *)
let rec exprgen builder = function
  | IntLiteral i -> L.const_int i32_t i
  | FloatLiteral f -> L.const_float double_t f
  | StringLiteral str -> L.build_global_stringptr str "" builder
  | BoolLiteral b -> L.const_int i1_t (if b then 1 else 0)
  | Id s -> L.build_load (lookup s) s builder
  | Binop (e1, op, e2) ->
    let e1' = exprgen builder e1
    and e2' = exprgen builder e2
    and t1 = type_of_expr e1
    and t2 = type_of_expr e2 in
    let e1' = if t1 = Int && t2 = Float then
                (L.const_sitofp e1' double_t)
              else e1'
    and e2' = if t1 = Float && t2 = Int then
                (L.const_sitofp e2' double_t)
              else e2'
    in
```

```
      (match op with
        | Add when t1 = Float || t2 = Float -> L.build_fadd e1' e2' "tmp" builder
        | Add when t1 = Int && t2 = Int -> L.build_add e1' e2' "tmp" builder

        | Sub when t1 = Float || t2 = Float -> L.build_fsub e1' e2' "tmp" builder
        | Sub when t1 = Int && t2 = Int -> L.build_sub e1' e2' "tmp" builder

        | Mult when t1 = Float || t2 = Float -> L.build_fmul e1' e2' "tmp" builder
        | Mult when t1 = Int && t2 = Int -> L.build_mul e1' e2' "tmp" builder

        | Div when t1 = Float || t2 = Float -> L.build_fdiv e1' e2' "tmp" builder
        | Div when t1 = Int && t2 = Int -> L.build_sdiv e1' e2' "tmp" builder

        | Mod when t1 = Float || t2 = Float -> L.build_frem e1' e2' "tmp" builder
        | Mod when t1 = Int && t2 = Int -> L.build_srem e1' e2' "tmp" builder

        | And     -> L.build_and    e1' e2' "tmp" builder
        | Or      -> L.build_or     e1' e2' "tmp" builder

        | Equal when t1 = Float || t2 = Float -> L.build_fcmp L.Fcmp.Ueq e1' e2' "tmp"
builder
        | Equal   -> L.build_icmp L.Icmp.Eq     e1' e2' "tmp" builder

        | Neq when t1 = Float || t2 = Float -> L.build_fcmp L.Fcmp.Une e1' e2' "tmp"
builder
        | Neq     -> L.build_icmp L.Icmp.Ne     e1' e2' "tmp" builder

        | Less when t1 = Float || t2 = Float -> L.build_fcmp L.Fcmp.Ult e1' e2' "tmp"
builder
        | Less when t1 = Int && t2 = Int -> L.build_icmp L.Icmp.Slt e1' e2' "tmp" builder

        | Leq when t1 = Float || t2 = Float -> L.build_fcmp L.Fcmp.Ule e1' e2' "tmp"
builder
        | Leq when t1 = Int && t2 = Int -> L.build_icmp L.Icmp.Sle e1' e2' "tmp" builder

        | Greater when t1 = Float || t2 = Float -> L.build_fcmp L.Fcmp.Ugt e1' e2'
"tmp" builder
        | Greater when t1 = Int && t2 = Int -> L.build_icmp L.Icmp.Sgt e1' e2' "tmp"
builder
```

```
      | Geq when t1 = Float || t2 = Float -> L.build_fcmp L.Fcmp.Uge e1' e2' "tmp"
builder
      | Geq when t1 = Int && t2 = Int -> L.build_icmp L.Icmp.Sge e1' e2' "tmp" builder

      | Addeq   -> exprgen builder (Assign((U.string_of_id e1), Binop(e1, Add, e2)))
      | Subeq   -> exprgen builder (Assign((U.string_of_id e1), Binop(e1, Sub, e2)))
      | Multeq  -> exprgen builder (Assign((U.string_of_id e1), Binop(e1, Mult, e2)))
      | Diveq   -> exprgen builder (Assign((U.string_of_id e1), Binop(e1, Div, e2)))
      | Modeq   -> exprgen builder (Assign((U.string_of_id e1), Binop(e1, Mod, e2)))

      | _ -> exprgen builder Noexpr
    )
  | Unop (uop, e) -> let e' = exprgen builder e
                     and t = type_of_expr e in
    (match uop with
      | Neg when t = Float -> L.build_fneg
      | Neg   -> L.build_neg
      | Not   -> L.build_not
    ) e' "tmp" builder
  | Assign (s, e) -> let e' = exprgen builder e in
                     ignore (L.build_store e' (lookup s) builder); e'
  | ArrayAssign(e1, e2) ->
    let codegen_array_access isAssign e el builder =
      let index = exprgen builder (List.hd el) in
      let arr = exprgen builder e in
      let val_t = L.build_gep arr [| index |] "tmp" builder in
      if isAssign then val_t else L.build_load val_t "tmp" builder
    in

    let rhs = exprgen builder e2 in

    let lhs = match e1 with
      | ArrayAccess(e, el) -> codegen_array_access true e el builder
    in

    ignore (L.build_store rhs lhs builder);
    rhs
```

```
    | Call ("print", [e]) ->
       (match (type_of_expr e) with
         | Int -> L.build_call printf_func [| int_format_str;
                     (exprgen builder e) |] "printf" builder
         | Float -> L.build_call printf_func [| float_format_str;
                     (exprgen builder e) |] "printf" builder
         | String -> L.build_call printf_func [| str_format_str;
                     (exprgen builder e) |] "printf" builder
         | Bool -> L.build_call printf_func [| int_format_str;
                     (exprgen builder e) |] "printf" builder
         | _ -> L.build_call printf_func [| int_format_str;
                   (exprgen builder e) |] "printf" builder
       )

    | Call (f, act) ->
       let (fdef, fdecl) = StringMap.find f function_decls in
       let actuals = List.rev (List.map (exprgen builder) (List.rev act)) in
       let result = (match (ast_type_of_datatype fdecl.ftype) with Void -> ""
                                    | _ -> f ^ "_result")
       in L.build_call fdef (Array.of_list actuals) result builder
    | ArrayCreate (t, el) ->
       (* initialize array from DICE *)
       let initialize_array arr arr_len init_val start_pos builder =
         let new_block label =
           let f = L.block_parent (L.insertion_block builder) in
           L.append_block (L.global_context ()) label f
         in

         let bbcurr = L.insertion_block builder in
         let bbcond = new_block "array.cond" in
         let bbbody = new_block "array.init" in
         let bbdone = new_block "array.done" in
         ignore (L.build_br bbcond builder);
         L.position_at_end bbcond builder;

         let counter = L.build_phi [L.const_int i32_t start_pos, bbcurr] "counter"
builder in
         L.add_incoming ((L.build_add counter (L.const_int i32_t 1) "tmp" builder),
bbbody) counter;
```

```
      let cmp = L.build_icmp L.Icmp.Slt counter arr_len "tmp" builder in
      ignore (L.build_cond_br cmp bbbody bbdone builder);
      L.position_at_end bbbody builder;

      let arr_ptr = L.build_gep arr [| counter |] "tmp" builder in
      ignore (L.build_store init_val arr_ptr builder);
      ignore (L.build_br bbcond builder);
      L.position_at_end bbdone builder;
    in

    let e = List.hd el in
    let t = get_type t in

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

    let arr = L.build_array_malloc t size_real "tmp" builder in
    let arr = L.build_pointercast arr (L.pointer_type t) "tmp" builder in
    let arr_len_ptr = L.build_pointercast arr (L.pointer_type i32_t) "tmp" builder in

    ignore (L.build_store size_real arr_len_ptr builder);
    initialize_array arr_len_ptr size_real (L.const_int i32_t 0) 0 builder;
    arr

  | ArrayAccess(e, el) ->
    let codegen_array_access isAssign e el builder =
      let index = exprgen builder (List.hd el) in
      let arr = exprgen builder e in
      let val_t = L.build_gep arr [| index |] "tmp" builder in
      if isAssign then val_t else L.build_load val_t "tmp" builder
    in
    codegen_array_access false e el builder
  | Noexpr -> L.const_int i32_t 0
in

let add_terminal builder f =
  match L.block_terminator (L.insertion_block builder) with
```

```
    | Some _ -> ()
    | None -> ignore (f builder)
in

(* build the statements in a function *)
let rec stmtgen builder = function
  | Block sl -> List.fold_left stmtgen builder sl
  | Expr e -> ignore(exprgen builder e); builder
  | Return e -> ignore (match (ast_type_of_datatype fdecl.ftype) with
    | Void -> L.build_ret_void builder
    | _ -> L.build_ret (exprgen builder e) builder); builder

  | If (e, s1, s2) ->
    let bool_val = exprgen builder e 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 (stmtgen (L.builder_at_end context then_bb) s1)
    (L.build_br merge_bb);

    let else_bb = L.append_block context "else" the_function in
    add_terminal (stmtgen (L.builder_at_end context else_bb) s2)
    (L.build_br merge_bb);

    ignore (L.build_cond_br bool_val then_bb else_bb builder);
    L.builder_at_end context merge_bb

  | For (e1, e2, e3, s) -> stmtgen builder
      ( Block [Expr e1 ; While (e2, Block [s ; Expr e3]) ] )
  | While (e, s) ->
    let pred_bb = L.append_block context "while" the_function
    in
    ignore (L.build_br pred_bb builder);

    let body_bb = L.append_block context "while_body" the_function
    in
    add_terminal (stmtgen (L.builder_at_end context body_bb) s)
    (L.build_br pred_bb);
```

```
      let pred_builder = L.builder_at_end context pred_bb
      in
      let bool_val = exprgen pred_builder e
      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

    | Local (_, s, e) -> (match e with
      | Noexpr -> builder
      | _ -> ignore(exprgen builder (Assign(s, e))); builder)
  in

  if fdecl.fname = main_func_name then
    let builder = List.fold_left stmtgen builder stmts in
    ignore(L.build_ret (L.const_int i32_t 0) builder);
  else
    let builder = List.fold_left stmtgen builder fdecl.body in
    add_terminal builder (match (ast_type_of_datatype fdecl.ftype) with
      | Void -> L.build_ret_void
      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0));
 in

 List.iter build_function_body functions;
 let mainfdecl = { ftype = Primitive(Int); fname = main_func_name; formals = []; body
= stmts } in
 build_function_body mainfdecl;
 The_module
```

Stockq.ml

```
type action = Tokenize | Parserize | Ast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-t", Tokenize);
                              ("-p", Parserize);
                              ("-c", Compile);
```

```
                        ("-a",Ast);
                        ("-l",LLVM_IR)] (* Generate LLVM, don't check *)
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  match action with
    | Tokenize ->
       let rec print_tokens = function
         | Parser.EOF -> print_endline ""
         | token ->
            print_endline (Utils.string_of_token token);
            print_tokens (Scanner.token lexbuf)
       in
       print_tokens (Scanner.token lexbuf)
    | Parserize ->
       let ast = Parser.program Scanner.token lexbuf in
       let result = Utils.string_of_program [] ast in
       print_endline result
    | _ ->
       let ast = Parser.program Scanner.token lexbuf in
       Semant.check ast;
       match action with
         | Ast -> print_string (Utils.string_of_program [] ast)
         | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate ast))
         | Compile -> let m = Codegen.translate ast in
          Llvm_analysis.assert_valid_module m;
          print_string (Llvm.string_of_llmodule m)
         | _ -> ()
```

Stockq.sh

```
#!/bin/bash

if [ "$#" -ne 3 ]; then
   echo "Usage: $0 [stockq_file] [flag] [output_file]" 1>&2
   exit 1
fi

MYDIR="$(dirname "$(which "$0")")"
STOCKQ_FILE="$MYDIR/stockq"
```

```
cat $1 | $STOCKQ_FILE $2 > $3

exit 0
```

Makefile

```
# StockQ: compiler Makefile
#  - builds and manages all compiler components


default: stockq

.PHONY: stockq.native
stockq.native:
	ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags \
		-w,+a-4 stockq.native
	mv stockq.native stockq

.PHONY: test
test:
	@echo Running the test suite!
	@cd ../test; make > /dev/null
	@echo All tests passed, directory clean!

.PHONY: clean
clean:
	ocamlbuild -clean
	rm -rf testall.log *.diff stockq scanner.ml parser.ml parser.mli
	rm -rf *.cmx *.cmi *.cmo *.cmx *.o temp.ll parser.output


# More detailed: build using ocamlc/ocamlopt + ocamlfind to locate LLVM

OBJS = ast.cmx utils.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx
stockq.cmx

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

```
scanner.ml: scanner.mll
	ocamllex scanner.mll

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

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

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

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

### Generated by "ocamldep *.ml *.mli" after building scanner.ml and parser.ml
ast.cmo:
ast.cmx:
utils.cmo: parser.cmo ast.cmo
utils.cmx: parser.cmx ast.cmx
codegen.cmo: ast.cmo
codegen.cmx: ast.cmx
stockq.cmo: semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo utils.cmo
stockq.cmx: semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx utils.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
semant.cmo: ast.cmo
semant.cmx: ast.cmx
parser.cmi: ast.cmo
```

Scanner_test.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'
```

```
INPUT_FILES="scanner/*.in"
printf "${CYAN}Running scanner tests...\n${NC}"

for input_file in $INPUT_FILES; do
   output_file=${input_file/.in/.out}
   ../compiler/stockq -t < $input_file | cmp -s $output_file -
   if [ "$?" -eq 0 ]; then
      printf "%-65s ${GREEN}SUCCESS\n${NC}" "  - checking $input_file..."
   else
      printf "%-65s ${RED}ERROR\n${NC}" "  - checking $input_file..." 1>&2
      exit 1
   fi
done

exit 0
```

Parser_test.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'

INPUT_FILES="parser/*.in"
printf "${CYAN}Running parser tests...\n${NC}"

for input_file in $INPUT_FILES; do
   output_file=${input_file/.in/.out}
   ../compiler/stockq -p < $input_file | cmp -s $output_file -
   if [ "$?" -eq 0 ]; then
      printf "%-65s ${GREEN}SUCCESS\n${NC}" "  - checking $input_file..."
   else
      printf "%-65s ${RED}ERROR\n${NC}" "  - checking $input_file..." 1>&2
      exit 1
   fi
done
```

```
exit 0
```

Compiler_test.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'

INPUT_FILES="compiler/*.sq"
ERROR_FILES="compiler/fail-*.sq"
printf "${CYAN}Running compiler tests...\n${NC}"

for input_file in $INPUT_FILES; do
   llfile=${input_file/.sq/.ll}
   tmpfile=${input_file/.sq/.tempout}
   output_file=${input_file/.sq/.out}

   # compile stockq program to llvm file
   ../compiler/stockq.sh $input_file -c $llfile

   lli "$llfile" > "$tmpfile"
   echo " >> "$tmpfile"

   # if test output file exists, compare compiled output to it
   if [ -e "$output_file" ]; then
      cmp -s $tmpfile $output_file
      if [ "$?" -ne 0 ]; then
         printf "%-65s ${RED}ERROR\n${NC}" "  - checking $output_file..." 1>&2
         rm -f $llfile $tmpfile
         exit 1
      fi
   fi

   printf "%-65s ${GREEN}SUCCESS\n${NC}" "  - checking $input_file..."
   rm -f $llfile $tmpfile
done
```

```
for input_file in $ERROR_FILES; do
   tmpfile=${input_file/.err/.tempout}
   output_file=${input_file/.err/.out}

   # compile stockq program to llvm file
   ../compiler/stockq -c &> $tmpfile < $input_file

   # if test output file exists, compare compiled output to it
   if [ -e "$output_file" ]; then
      cmp -s $tmpfile $output_file
      if [ "$?" -ne 0 ]; then
         printf "%-65s ${RED}ERROR\n${NC}" "  - checking $output_file..." 1>&2
         rm -f $llfile $tmpfile
         exit 1
      fi
   fi

   printf "%-65s ${GREEN}SUCCESS\n${NC}" "  - checking $input_file..."
   rm -f $llfile $tmpfile
done

exit 0
```

Makefile (* test Makefile *)

```
# StockQ: test Makefile
#  - builds all files needed for testing, then runs tests

default: test clean

test: scanner parser compiler

scanner: build
	@./scanner_test.sh

parser: build
	@./parser_test.sh

compiler: build
```

```
	@./compiler_test.sh

build:
	@echo Building tests and necessary files.
	@cd ../compiler; make > /dev/null

.PHONY: clean
clean:
	@echo Cleaning up all directories.
	@cd ../compiler; make clean > /dev/null
```

Compiler tests

```
Fatal error: exception Failure("illegal assignment int = float in FloatLiteral(6.)")
int a = 1.0;
Fatal error: exception Failure("illegal binary operator int Add string in Binop(Id(x),
Add, StringLiteral(hello))")
int x = 1;

print(x + "hello");
Fatal error: exception Failure("illegal binary operator bool Sub int in
Binop(BoolLiteral(true), Sub, IntLiteral(1))")
print(true - 1);
Fatal error: exception Failure("undeclared identifier a")
print(a + 1);
Fatal error: exception Failure("nothing may follow a return")
def int myfunc () {
   return 5;
   print("great");
}

print(myfunc());
Fatal error: exception Failure("undeclared identifier x")
for (x=0; x<5; x+=1) {
   print("why");
}
Fatal error: exception Parsing.Parse_error
for (;;) {
   print("needs expr");
}
Fatal error: exception Parsing.Parse_error
```

```
for(x = 9; x < 10; x+=1) {
   print("won't work");
}
Fatal error: exception Parsing.Parse_error
def int myfunc {
   return 0;
}
Fatal error: exception Failure("return gives float expected int in FloatLiteral(5.5)")
def int myfunc () {
   return 10.0;
}
Fatal error: exception Failure("expected Boolean expression in IntLiteral(1)")
if (1) print("hello world");
Fatal error: exception Failure("illegal binary operator bool Sub int in
Binop(BoolLiteral(true), Sub, IntLiteral(5))")
if (false) {
   print("don't print me");
} else if (true - 5) {
   print("try printing me?");
}
Fatal error: exception Failure("undeclared identifier a")
print(a);
Fatal error: exception Failure("return gives float expected int in FloatLiteral(5.5)")
def float myfunc () {
   return 3;
}
Fatal error: exception Failure("expected Boolean expression in IntLiteral(4)")
while(4) {
   print("print me 4 times?");
}
31

def int add(int x, int y) {
   return x + y;
}

print( add(18, 13) );
9.500000
5.100000
```

```
print(5.5 + 4);
print(2 + 3.1);
```
```
26

print(23 + 3);
```
```
11

print(1 + 2 * 3 + 4);
```
```
47

def int foo(int a) {
   return a;
}

int a;
a = 42;
a = a + 5;
print(a);
```
```
11.400000
16.000000
20.800000
1.454545
0.400000

print(4.4 + 7);
print(19 - 3.0);
print(4 * 5.2);
print(8/5.5);
print(7%2.2);
```
```
0
0
0
0
0
0
0
```

```
0
0
0
0
1
2
3
4
5
6
7
8
9

int[] a = int[10];
int x;
for(x=0;x<10;x+=1){
   print(a[x]);
}
for(x=0;x<10;x+=1){
   a[x] = x;
   print(a[x]);
}
5.500000
2.000000

float[] f = float[2];

f[0] = 5.5;
f[1] = 2.0;

print( f[0] );
print( f[1] );
1
1
2
3
5
8
```

```
def int fib (int x) {
   if (x < 2) return 1;
   return fib(x - 1) + fib(x-2);
}

print( fib(0) );
print( fib(1) );
print( fib(2) );
print( fib(3) );
print( fib(4) );
print( fib(5) );
0
1
2
3
4
42

int i;
for (i = 0; i < 5; i = i + 1) {
   print(i);
}
print(42);
0
1
2
3
4
42

int i;
i = 0;
for ( ; i < 5; ) {
   print(i);
   i = i + 1;
}
print(42);
42
```

```
def int add (int a, int b) {
   return a + b;
}

int a;
a = add(39, 3);
print(a);
2

def int fun(int x, int y) {
   return 0;
}

int i;
i = 1;

fun(i = 2, i = i + 1);

print(i);
42
17
192
8

def int printem(int a, int b, int c, int d) {
   print(a);
   print(b);
   print(c);
   print(d);
}

printem(42, 17, 192, 8);
62

def int add (int a, int b) {
   int c;
   c = a + b;
   return c;
```

```
}

int d;
d = add(52, 10);
print(d);
42

def int bar (int a, bool b, int c) { return a + c; }

print(bar(17, false, 25));
43

def int foo (int a) {
   print (a + 3);
}

foo(40);
7
4
11

def int gcd (int a, int b) {
   while ( a != b ) {
      if ( a > b ) {
         a = a - b;
      } else {
         b = b - a;
      }
   }
   return a;
}

print( gcd(14, 21) );
print( gcd(8, 36) );
print( gcd(99, 121) );
42
71
1
```

```
print(42);
print(71);
print(1);
42
17

if ( true ) print(42);
print(17);
42
17

if ( true ) print(42); else print(8);
print(17);
17

if (false) print(42);
print(17);
8
17

if (false) print(42); else print(8);
print(17);
42
17

def int cond (bool b) {
   int x;
   if (b)
      x = 42;
   else
      x = 17;
   return x;
}

print( cond(true) );
print( cond(false) );
43

def int foo( bool i ) {
```

```
    int i; /* should hide the formal i */

    i = 42;
    print( i + 1 );
}

foo ( true );
47

def int foo(int a, bool b) {
    int c;
    bool d;
    c = a;
    return c + 10;
}

print( foo(37, false) );
3
-1
2
50
99
0
1
99
1
0
99
1
0
99
1
1
0
99
0
1
99
0
```

```
1
1

print( 1 + 2 );
print( 1 - 2 );
print( 1 * 2 );
print( 100 / 2 );
print(99);
print(1 == 2);
print(1 == 1);
print(99);
print(1 != 2);
print(1 != 1);
print(99);
print(1 < 2);
print(2 < 1);
print(99);
print(1 <= 2);
print(1 <= 1);
print(2 <= 1);
print(99);
print(1 > 2);
print(2 > 1);
print(99);
print( 1 >= 2 );
print( 1 >= 1 );
print( 2 >= 1 );
1
0
1
0
0
0
1
1
1
0
1
0
```

```
-10
42

print(true);
print(false);
print(true and true);
print(true and false);
print(false and true);
print(false and false);
print(true or true);
print(true or false);
print(false or true);
print(false or false);
print(not false);
print(not true);
print(-10);
print(--42);
1
0

print(true);
print(false);
4.300000

print(4.3);
10.920000
34.320000
0.098700

print(10.92);
print(34.32);
print(0.0987);
5

print(5);
21
72
56
```

```
print(21);
print(72);
print(56);
```

```
1
2
3
5
8
```

```
def int myRec(int x){
   if(x<=1){
      return 1;
   }
   else{
      return (myRec(x-1)+myRec(x-2));
   }
}

int[] a = int[5];
int x;
for(x=0;x<5;x+=1){
   a[x] = myRec(x+1);
   print(a[x]);
}
```

```
42
```

```
int a;
a = 42;
print(a);
```

```
5
4
3
2
1
42
```

```
int i;
i = 5;
```

```
while (i > 0) {
   print(i);
   i = i - 1;
}
print(42);
14

def int foo(int a) {
   int j;
   j = 0;
   while (a > 0) {
      j = j + 2;
      a = a - 1;
   }
   return j;
}

print( foo(7) );
```

Parser tests

```
1+2;
4+5.6;

3-2;

2*3;

10/5;

9%4;

-100;

4 * (3-1);
Binop(IntLiteral(1), Add, IntLiteral(2))
Binop(IntLiteral(4), Add, FloatLiteral(5.6))
Binop(IntLiteral(3), Sub, IntLiteral(2))
Binop(IntLiteral(2), Mult, IntLiteral(3))
Binop(IntLiteral(10), Div, IntLiteral(5))
```

```
Binop(IntLiteral(9), Mod, IntLiteral(4))
Unop(Neg, IntLiteral(100))
Binop(IntLiteral(4), Mult, Binop(IntLiteral(3), Sub, IntLiteral(1)))

myvar;
studentAge;
print("hello");
myfunc(myvar);
get_string(my_int);
Id(myvar)
Id(studentAge)
Call(print, StringLiteral(hello))
Call(myfunc, Id(myvar))
Call(get_string, Id(my_int))

for (x = 3; x < 10; ) {
   print("why do meteors always land in craters?");
}
For(Assign(x, IntLiteral(3)); Binop(Id(x), Less, IntLiteral(10)); Noexpr) {
Block(Call(print, StringLiteral(why do meteors always land in craters?))) }

def int myfunc (int x, int y) {
   3+4;
}
def int myfunc (x, y) {
        Binop(IntLiteral(3), Add, IntLiteral(4))
}

if (x < 4) {
   print("hello world");
} else {
   print("x is greater than 4");
}
If(Binop(Id(x), Less, IntLiteral(4))) { Block(Call(print, StringLiteral(hello world))) } Else
{ Block(Call(print, StringLiteral(x is greater than 4))) }

5.5556;
42;
"hello world";
```

```
true;
false;
FloatLiteral(5.5556)
IntLiteral(42)
StringLiteral(hello world)
BoolLiteral(true)
BoolLiteral(false)
```

```
true;
false;
not true;
not false;
```

```
1 == 2;
3 < 4;
5 <= 6;
7 > 0;
1 >= -5;
not (1 < 5);
BoolLiteral(true)
BoolLiteral(false)
Unop(Not, BoolLiteral(true))
Unop(Not, BoolLiteral(false))
Binop(IntLiteral(1), Equal, IntLiteral(2))
Binop(IntLiteral(3), Less, IntLiteral(4))
Binop(IntLiteral(5), Leq, IntLiteral(6))
Binop(IntLiteral(7), Greater, IntLiteral(0))
Binop(IntLiteral(1), Geq, Unop(Neg, IntLiteral(5)))
Unop(Not, Binop(IntLiteral(1), Less, IntLiteral(5)))
```

```
while ( x < 42 ) {
   print("42 is my favorite number");
}
While(Binop(Id(x), Less, IntLiteral(42))) { Block(Call(print, StringLiteral(42 is my
favorite number))) }
```

Scanner tests

```
[
]
```

LBRACKET
RBRACKET

if
else
for
while
return
IF
ELSE
FOR
WHILE
RETURN

+
// ignore this
/* ignore
   everything
   inside
   this comment
*/
-
PLUS
MINUS

int
float
bool
void
true
false
struct
array
string
INT
FLOAT
BOOL
VOID
TRUE

FALSE
STRUCT
ARRAY
STRING

5.556
FLOAT_LITERAL

def def
DEF
DEF

34
34.5
myfunc
thisisanid
first_student
"stringlit"
"hello world!"
INT_LITERAL
FLOAT_LITERAL
ID
ID
ID
STRING_LITERAL
STRING_LITERAL

== !=

< <=
> >=

and or not
EQ
NEQ
LT
LEQ
GT
GEQ

AND
OR
NOT

+ - * / %

+= -= *= /= %=

=
PLUS
MINUS
TIMES
DIVIDE
MODULO
PLUSEQ
MINUSEQ
TIMESEQ
DIVIDEEQ
MODULOEQ
ASSIGN

())(
{}}{
;
;;
,
,,
LPAREN
RPAREN
RPAREN
LPAREN
LBRACE
RBRACE
RBRACE
LBRACE
SEMI
SEMI
SEMI
COMMA

COMMA
COMMA