MathLang: Final Report

Sophie Lucy (sjl2185)
Ravie Lakshmanan (rl2857)

Contents

Introduction

Initially, as a group of four, we had intended on building a financial language that would focus on the programmer being able to implement complex algorithms for financial analysis. As a result, when we split as a team, we decided to still be in the scope of a mathematical language but narrow our functions.

MathLang is a simple mathematical programming language that compiles to LLVM IR. Our goal in designing this language was to be able to help a programmer easily run through various mathematical operations.

Language Tutorial

Environment Setup

MathLang runs on a UNIX environment and has been tested on macOS Sierra (10.12.2). To compile and run the program, follow the below steps -

The ‘make’ command reads the .mlang files and generates the corresponding .ll files corresponding to LLVM IR.

```
$ make clean
$ make
$ ./mathlang <filename>.mlang
$ ./mathlang.sh input
```

Any program in MathLang starts with a main() function within which variables and statements can be defined to perform mathematical operations. Variables must be declared and initialized separately.

For example, for the sample program example.mathlang -

```
int  main(){
     printf("This is a sample program");
}
```

Compiling the above file, `./mathlang example.mlang` would generate the .ll file which can then be used by the shell script to generate the output.

Run and Test

Once in the MathLang directory, you can run the program with the Makefile and then test the files in the test suite in the folder `tests` that can be executed with the command "./testMathLang.sh"

```
$ make
ocamlfind ocamlopt -c -package llvm ast.ml
ocamlfind ocamlopt -c -package llvm codegen.ml
ocamlyacc parser.mly
ocamlc -c ast.ml
ocamlc -c parser.mli
ocamlfind ocamlopt -c -package llvm parser.ml
ocamllex scanner.mll
90 states, 4125 transitions, table size 17040 bytes
ocamlfind ocamlopt -c -package llvm scanner.ml
ocamlfind ocamlopt -c -package llvm semant.ml
$ ./testMathLang.sh
```

Hello World

To test a "Hello World!" program

```
$ make mathlang
$ ./mathlang < hello.mlang > hello.ll
```

Input the following into hello.mlang file:

```
int main(){
     prints("hello!");
     return 0;
}

$ /usr/local/opt/llvm38/bin/lli-3.8 hello.ll
"Hello!"
```

Language Reference Manual

Primitive Types

All primitives are first declared and separately assigned. MathLang supports the following primitive types -

`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

`string`

- a finite sequence of ASCII characters, enclosed in double quotes
- e.g. “we are stockx”

Tokens

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

Identifiers

Identifiers indicate function or variable name. MathLang identifiers are case-sensitive.

Comments

Both single and multi-line comments are supported in MathLang

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

```
/*this is a
multi-line comment/*
```

Whitespace

Whitespace is ignored in MathLang.

Separators

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

Reserved Words and Symbols

Data Types

`int`
`float`
`bool`
`string`
`true`
`false`

Boolean Logic Operators

`and`
`or`
`not`

Branch Control and Loops

`if`
`else`
`for`
`while`
`return`

Expressions

Declaration and Assignment

The general syntax is as follows:

- `<type> <identifier>;`
- e.g. `int x; float number; string name;`

Arithmetic Operators

`+` addition
`-` subtraction
`*` multiplication
`/` division

Relational Operators

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

Logic Operators

`and`
- logical intersection of two expressions

e.g. 0 and 1 evaluates to 0(false)

`or`
- logical union of two expressions

e.g. 1 or 0 evaluates to 1(true)

`not`
- logical negation of an expression

e.g. not 1 evaluates to 0

Branch Control

`if`

- conditional if statement are followed by a boolean expression

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

`else`

- An `else` statement may follow an if statement
- A statement list then follows
- e.g.
```
if(x==2){
  return true
}
else{
  return false
}
```

`for`

- The for statement allows for looping over a range of values. The format is as follows:
- e.g. `for (initialization; termination; upda te) { 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.

`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 (x>4) { x+y; }`

`return`

- The return keyword is used both in function declarations, and inside of functions to return a value. The syntax is as follows: `return expr;`
- In the `main()` function, the return keyword is used to exit the program.

Functions

`function`
- Establishes a user-defined function that will return a value

`void`
- A function can be set as `void` when it does not return a value

`return`
- Caller of the function

Built-in Functions

`print`
- Prints `int` values

`printb`
- Prints `bool` values

`printf`
- Prints `float` values

`prints`
- Prints `string` values

Project Outline

The idea of a financial language was proposed by Jesse who had had some interest in financial trading. While Ravie and Jesse outlined the specific structs and functions required in a financial language for the LRM, Sophie outlined the more common functionalities. As a full group, we had initially begun coding together but because of the dynamic within the team, we started to divide the work amongst ourselves each week and would work independently. Admittedly, this was extremely difficult to coordinate as there were several occasions when members of the group were overlapping on the work they were doing.

Development Process

By December 4, we had the basics of the scanner, parser, AST and code gen working so that we could run the "hello world" program. At this time, one half of the team decided to split. Consequently, we started to work on the project as a pair, trying to figure out how much we could achieve in the given time. Unfortunately, we spent a few days trying to implement both structs and arrays with the intention to then be able to implement a more complex mathematical function such as regression. However, as we moved closer to the deadline, we decided to prioritize having a language that would compile and therefore simplified the idea and started to build our language on top of the MicroC example. In the Appendix we included the code snippets of files where we started to implement arrays, structs and then float operations.

Testing

We came with various use cases that could be handled by our language and identified areas where there were errors and ensured they were fixed. We also tested for both positive and negative test cases so that we capture all possible inputs. In our Tests folder negative test cases were named "fail-<name>" and positive test cases called "test<name>".

Project Timeline

Milestone	***Date***
Initial Idea Formulated	September 22
Proposal Submitted	September 28
Language Reference Manual submitted	October 26
Scanner, Parser and AST completed	December 4
Final Project Completed	December 19

Team Responsibilities

The responsibilities were equally distributed between the two of us. As a pair, there was no strict division of work based on the role. We worked better towards the end of our project when we pair programmed to be more efficient with our time.

Team Member	***Responsibility***
Sophie Lucy	Semantic Analyzer, Testing, AST, code cleanup
Ravie Lakshmanan	Parser, Scanner, AST, code generation, code cleanup

Architecture

The MathLang compiler constitutes the following -

parser.mly - Scans the tokens passed from the scanner to produce an AST representation of the program based on the definitions provided

scanner.mll - Reads a source file and tokenizes it to the corresponding token output

ast.ml - The abstract syntax tree representation of the program

semant.ml - Semantically checks incoming AST representation to make sure the expressions are properly type-checked

codegen.ml - Converts a semantically checked AST into a executable LLVM code by producing LLVM IR

mathlang.ml - The main module that calls on all the other modules depending on compiler flags passed to it

Lexical Analysis (Scanner)

Implemented using ocamllex, the scanner takes a .mlang program and produces a stream of tokens, providing basic lexical analysis. Whitespace and comments are discarded, and that programs with invalid tokens are caught.

Syntactic Analysis (Parser and AST)

Implemented using ocamlyacc, the parser takes the stream of tokens read by the scanner, and then uses them to generate an abstract syntax tree that's defined in the OCaml ast file. Errors like invalid syntax (i.e. when tokens are in an invalid order) are caught in this stage.

Semantic Analysis (Semantic Checker)

Implemented using OCaml, the semantic checker looks for type mismatches in the program, duplicated variables, duplicated function names

Code Generation

Implemented using OCaml, the compiled file takes in the AST and parses it to generate corresponding LLVM IR code that can be readily executed to generate output.

Testing Strategy

Unit Testing

During development, we ran unit tests as we added each layer from the scanner, parser, AST, semantic checker and codegen to check that the files could be compiled in LLVM. We also used unit tests when adding new features to the language to check for edge cases and uncaught errors.

Regression Testing

We adapted the automated regression testing script from MicroC so as to cover both positive and negative test inputs. The tests can be found in the `tests` folder of the project.

Sample Test Scripts

test.mlang

```
int main()
{
   print(1);
   printb(true);
   printf(1.0);
   prints("Hello World!");
   return 0;
}
```

test.out

```
1
1
1.000000
"Hello World!"
```

Conclusions

Sophie Lucy

As my first class doing a group project at Columbia, the class has helped me gain a greater appreciation for the creators of programming languages both from a technical and non-technical standpoint. With regard to the non-technical aspect of the project, I now understand the importance of establishing a group that I can work well with especially as the project required that each person had a thorough understanding of every aspect of the language and therefore it was necessary for the group to sit down together to code. Things also took longer than I expected they would particularly as we were overly ambitious in the beginning. A little OCaml goes a long way and small tweak can cause an entire program to crash.

Ravie Lakshmanan

Although I had learnt the basics of NFA, DFA, Parsing and various other aspects associated with programming languages during my undergrad, this was my first hands on experience creating one. The project gave me an opportunity to understand how each of these stages work behind the scene, and also helped me immensely in improving my expertise with OCaml. We started as a team of four, but due to various time constraints we couldn’t sit together very often the way I would have otherwise preferred. This led to a lot of scaling back with respect to design. But I would also acknowledge that this project made me appreciate time management a lot more. It also highlighted the importance of a project as a collaborative effort.

Appendix

Scanner

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "returns"{ RETURNS }
| "int"    { INT }
| "float"  { FLOAT }
| "bool"   { BOOL }
| "void"   { VOID }
```

```
| "string" { STRING }
| "true"   { TRUE }
| "false"  { FALSE }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['0'-'9']*['.']['0'-'9']+ as lxm { FLOATLIT( float_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '"'('\\'_|[^'"'])*'"' as str { STRINGLIT(str) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped
char)) }

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

Parser

```
%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN RETURNS IF ELSE FOR WHILE INT FLOAT BOOL VOID STRING
%token <int> LITERAL
%token <float> FLOATLIT
%token <string> STRINGLIT
%token <string> ID
%token FUNCTION
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
```

```
%right NOT NEG

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }

decls:
                        {[],[]}
  | decls var_decl      {($2 :: fst $1), snd $1}
  | decls fdecl         {fst $1, ($2 :: snd $1)}

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

var_decl:
  typ ID SEMI           { ($1, $2) }

var_decl_list:
                        {[]}
 | var_decl_list var_decl{$2 :: $1}

fdecl:
    typ ID LPAREN formals_opt RPAREN LBRACE var_decl_list stmt_list
RBRACE
     { {
       ftyp = $1;
       fname = $2;
       formals = $4;
       locals = List.rev $7;
       body = List.rev $8
    } }

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

formal_list:
    typ ID                     { [($1, $2)] }
```

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

typ:
    INT { Int }
  | FLOAT { Float }
  | BOOL { Bool }
  | VOID { Void }
  | STRING { String }

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) }

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

expr:
    LITERAL          { Literal($1) }
  | FLOATLIT         { FloatLit($1) }
  | STRINGLIT        { StringLit($1) }
  | TRUE             { BoolLit(true) }
  | FALSE            { BoolLit(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 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) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($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

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

type uop = Neg | Not

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

type bind = typ * string

type expr =
    Literal of int
  | FloatLit of float
  | BoolLit of bool
  | StringLit of string
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of string * 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

type func_decl = {
    ftyp : typ;
    fname : string;
    formals : bind list;
    locals: bind list;
    body : stmt list;
  }

type program =  bind list * func_decl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | 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"
  | Void -> "void"
  | String -> "string"

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

```
let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | FloatLit(l) -> string_of_float l
  | StringLit(str) -> str
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | 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(id, e) ->  id ^ "=" ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^
"}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | 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

let string_of_fdecl fdecl =
  string_of_typ fdecl.ftyp ^" "^
  fdecl.fname ^
  "(" ^ String.concat ", " (List.map snd fdecl.formals) ^ ")" ^
"\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals)^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"
```

```
let string_of_program (vdecls, fdecls) =
  String.concat "" (List.map string_of_vdecl vdecls) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl fdecls)
```

Semantic Checker

```
open Ast

module StringMap = Map.Make(String)

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

   Check each global variable, then check each function *)

let check (globals, functions) =

  (* 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
       (Void, n) -> raise (Failure (exceptf n))
     | _ -> ()
  in

  (* Raise an exception of 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
```

```
  (* Perform binary operation semantic checks depending on the data
type passed *)
  let checkBinaryOp e1 op e2 err =
     let getequal type1 type2 op =
        if (type1 = Float || type2 = Float)
          then raise (Failure ("illegal binary operator used for
float types"))
        else
          match type1, type2 with
            Int, Int -> Bool
          | Bool, Bool -> Bool
          | _  -> raise (Failure ("Invalid equality operator " ^
string_of_op op ^ "for types " ^
                      (string_of_typ type1) ^ " and " ^
(string_of_typ type2)))
     in

     let getlogic type1 type2 op =
        match type1, type2 with
          Bool, Bool -> Bool
        | _          -> raise (Failure ("invalid type for logical
operator " ^
                      string_of_op op ^ " for types " ^
(string_of_typ type1) ^ " and " ^ (string_of_typ type2)))
     in

     let getcomp type1 type2 op =
        match type1, type2 with
          Int, Int     -> Bool
      | Float, Float -> Bool
        | _            -> raise (Failure ("invalid type for
comparison operator " ^
                      string_of_op op ^ " for types " ^
(string_of_typ type1) ^ " and " ^ (string_of_typ type2)))
     in

     let getarith type1 type2 op =
        match type1, type2 with
          Int, Float
        | Float, Int
        | Float, Float -> Float
        | Int, Int     -> Int
```

```
        | _            -> raise (Failure ("invalid type for
arithmetic operator " ^
                     string_of_op op ^ " for types " ^
(string_of_typ type1) ^ " and " ^ (string_of_typ type2)))
     in

     match op with
        Equal | Neq                  -> getequal e1 e2 op
      | Less | Leq | Greater | Geq   -> getcomp e1 e2 op
      | Add | Mult | Sub | Div       -> getarith e1 e2 op
      | And | Or                     -> getlogic e1 e2 op
      | _                            -> raise err
  in

  (**** Checking Global Variables ****)

  List.iter (check_not_void (fun n -> "illegal void global " ^ n))
globals;

  report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd
globals);

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

  if List.mem "print" (List.map (fun fd -> fd.fname) functions)
  then raise (Failure ("function print 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 built_in_decls = StringMap.empty in

  let built_in_decls = StringMap.add "print"
     { ftyp = Void; fname = "print"; formals = [(Int, "x")];
     locals = []; body = [] } built_in_decls in
  let built_in_decls = StringMap.add "printb"
     { ftyp = Void; fname = "printb"; formals = [(Bool, "x")];
     locals = []; body = [] } built_in_decls  in
  let built_in_decls = StringMap.add "printf"
     { ftyp = Void; fname = "printf"; formals = [(Float, "x")];
     locals = []; body = [] } built_in_decls in
  let built_in_decls = StringMap.add "prints"
```

```
     { ftyp = Void; fname = "prints"; formals = [(String, "x")];
     locals = []; body = [] } built_in_decls 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 _ = function_decl "main" in (* Ensure "main" is defined *)

  let check_function func =

    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);

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

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

    (* Type of each variable (global, formal, or local *)
    let symbols = List.fold_left (fun m (t, n) -> StringMap.add n t
m)
      StringMap.empty (globals @ func.formals @ func.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
      Literal _ -> Int
       | BoolLit _ -> Bool
       | StringLit _ -> String
       | FloatLit _ -> Float
       | Id s -> type_of_identifier s
       | Binop(e1, op, e2) as ex -> let t1 = expr e1
                                    and t2 = expr e2 in
        checkBinaryOp t1 op t2 (Failure ("illegal binary operator " ^
              string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
              string_of_typ t2 ^ " in " ^ string_of_expr ex))
       | 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 operator " ^
string_of_uop op ^
                   string_of_typ t ^ " in " ^ string_of_expr ex)))
       | Noexpr -> Void
       | 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_typ lt ^
                           " = " ^ string_of_typ rt ^ " in " ^
                           string_of_expr ex))
       | Call(fname, actuals) as call -> 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 et = expr e in
              ignore (check_assign ft et
                (Failure ("illegal actual argument found " ^
string_of_typ et ^
                " expected " ^ string_of_typ ft ^ " in " ^
string_of_expr e))))
             fd.formals actuals;
           fd.ftyp
    in
```

```
    let check_bool_expr e = if expr e != Bool
     then raise (Failure ("expected Boolean expression in " ^
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 = func.ftyp then () else
         raise (Failure ("return gives " ^ string_of_typ t ^ "
expected " ^
                        string_of_typ func.ftyp ^ " in " ^
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
    in

    stmt (Block func.body)

  in
  List.iter check_function functions
```

Code Generator

```
module L = Llvm
module A = Ast

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

```
let context = L.global_context ()
let the_module = L.create_module context "MathLang"

let i32_t    = L.i32_type  context
let i8_t     = L.i8_type   context
let i1_t     = L.i1_type   context
let double_t = L.double_type context
let str_t    = L.pointer_type i8_t
let void_t   = L.void_type context

let translate (globals, functions) =

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

  (* Declare each global variable; remember its value in a map *)
  let global_vars =
    let global_var m (t, n) =
      let init = L.const_int (ltype_of_typ t) 0
      in StringMap.add n (L.define_global n init the_module) m in
    List.fold_left global_var StringMap.empty globals 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

  (* Define each function (arguments and return type) so we can call
it *)
  let function_decls =
    let function_decl m fdecl =
      let name = fdecl.A.fname
      and formal_types =
     Array.of_list (List.map (fun (t,_) -> ltype_of_typ t)
fdecl.A.formals)
      in let ftype = L.function_type (ltype_of_typ fdecl.A.ftyp)
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 *)
  let build_function_body fdecl =
    let (the_function, _) = StringMap.find fdecl.A.fname
function_decls in
    let builder = L.builder_at_end context (L.entry_block
the_function) in

    let int_format_str   = L.build_global_stringptr "%d\n" "fmt"
builder in
    let str_format_str   = L.build_global_stringptr "%s\n" "fmt"
builder in
    let float_format_str = L.build_global_stringptr "%f\n" "fmt"
builder 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 builder in
      ignore (L.build_store p local builder);
      StringMap.add n local m in

      let add_local m (t, n) =
      let local_var = L.build_alloca (ltype_of_typ t) n builder
      in StringMap.add n local_var m in

      let formals = List.fold_left2 add_formal StringMap.empty
fdecl.A.formals
          (Array.to_list (L.params the_function)) in
      List.fold_left add_local formals fdecl.A.locals in

    (* Return the value for a variable or formal argument *)
    let lookup n = try StringMap.find n local_vars
                   with Not_found -> StringMap.find n global_vars
    in
```

```
    (* Construct code for an expression; return its value *)
    let rec expr builder = function
      A.Literal i -> L.const_int i32_t i
      | A.FloatLit f -> L.const_float double_t f
      | A.StringLit str -> L.build_global_stringptr str "tmp" builder
      | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
      | A.Noexpr -> L.const_int i32_t 0
      | A.Id s -> L.build_load (lookup s) s builder
      | A.Binop (e1, op, e2) ->
           let e1' = expr builder e1
                and e2' = expr builder e2 in
           (match op with
        | A.Add     -> L.build_add
        | A.Sub     -> L.build_sub
        | A.Mult    -> L.build_mul
        | A.Div     -> L.build_sdiv
        | A.And     -> L.build_and
        | A.Or      -> L.build_or
        | A.Equal   -> L.build_icmp L.Icmp.Eq
        | A.Neq     -> L.build_icmp L.Icmp.Ne
        | A.Less    -> L.build_icmp L.Icmp.Slt
        | A.Leq     -> L.build_icmp L.Icmp.Sle
        | A.Greater -> L.build_icmp L.Icmp.Sgt
        | A.Geq     -> L.build_icmp L.Icmp.Sge
           ) e1' e2' "tmp" builder

      | A.Unop(op, e) ->
        let e' = expr builder e in
        (match op with
          A.Neg     -> L.build_neg
           | A.Not     -> L.build_not) e' "tmp" builder
      | A.Assign (s, e) -> let e' = expr builder e in
                         ignore (L.build_store e' (lookup s)
builder); e'
      | A.Call ("print", [e])  ->
        L.build_call printf_func [| int_format_str ; (expr builder e)
|]
          "printf" builder
      | A.Call ("printb", [e]) ->
        L.build_call printf_func [| int_format_str ; (expr builder e)
|]
             "printf" builder
```

```
      | A.Call ("printf", [e] ) ->
       L.build_call printf_func [| float_format_str ; (expr builder
e) |]
            "printf" builder
      | A.Call ("prints", [e]) ->
          L.build_call printf_func [| str_format_str; (expr builder
e) |]
            "printf" builder
      | A.Call (f, act) ->
         let (fdef, fdecl) = StringMap.find f function_decls in
      let actuals = List.rev (List.map (expr builder) (List.rev
act)) in
      let result = (match fdecl.A.ftyp with A.Void -> ""
                                         | _ -> f ^ "_result") in
         L.build_call fdef (Array.of_list actuals) result builder
    in

    (* Invoke "f builder" if the current block doesn't already
       have a terminal (e.g., a branch). *)
    let add_terminal builder f =
      match L.block_terminator (L.insertion_block builder) with
     Some _ -> ()
      | None -> ignore (f builder) in

    (* Build the code for the given statement; return the builder for
       the statement's successor *)
    let rec stmt builder = function
     A.Block sl -> List.fold_left stmt builder sl
      | A.Expr e -> ignore (expr builder e); builder
      | A.Return e -> ignore (match fdecl.A.ftyp with
       A.Void -> L.build_ret_void builder
     | _ -> L.build_ret (expr builder e) builder); builder
      | A.If (predicate, then_stmt, else_stmt) ->
         let bool_val = expr builder 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 (stmt (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 (stmt (L.builder_at_end context else_bb)
else_stmt)
        (L.build_br merge_bb);

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

      | A.While (predicate, body) ->
       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 (stmt (L.builder_at_end context body_bb) body)
         (L.build_br pred_bb);

       let pred_builder = L.builder_at_end context pred_bb in
       let bool_val = expr 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

      | A.For (e1, e2, e3, body) -> stmt builder
         ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ; A.Expr
e3]) ] )
    in

    (* Build the code for each statement in the function *)
    let builder = stmt builder (A.Block fdecl.A.body) in

    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.A.ftyp with
        A.Void -> L.build_ret_void
      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in

  List.iter build_function_body functions;
  the_module
```

Top Level

```
type action = Ast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);  (* Print the AST only *)
                        ("-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
  Semant.check ast;
  match action with
    Ast -> print_string (Ast.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)
```

Scripts

Makefile

```
.PHONY : mathlang.native

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

# "make clean" removes all generated files

.PHONY : clean
clean :
	ocamlbuild -clean
```

```
	rm -rf testall.log *.diff mathlang scanner.ml parser.ml
parser.mli
	rm -rf *.cmx *.cmi *.cmo *.cmx *.o

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

OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx
mathlang.cmx

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

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 :

codegen.cmo : ast.cmo
codegen.cmx : ast.cmx

mathlang.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
mathlang.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx

parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
```

```
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx

semant.cmo : ast.cmo
semant.cmx : ast.cmx

parser.cmi : ast.cmo
```