miniMap

COMS 4115 Programming Languages and Translators Fall 2017

Ryan DeCosmo (rd2680)
Olesya Medvedeva (oam2113)
Jyhyun Song (js4390)
Charis Lam (cl3257)

Table of Contents

Title 1

Table of Contents 2

Introduction 4

Language Tutorial 5
- MiniMap interface 5
- Simple Hello, World program 6
- miniMap Hello, World program(s) 6

Language Reference Manual 8
- Lexical conventions 8
- Data types 8
- Variables 9
- Expressions 9
- Statements 9
- Functions 10
- Scope 10
- miniMap Standard Library 10

Project Process 12
- Members and roles 12
- Process and Timeline 12
- Software development environment 12
- Programming style guide 13
- Project Log 13

Architectural Design 13

Testers 13
- Test 1 13
- Test 2 13
- Test 3 14
- Test 4 14

Lessons Learned 15

Appendix 16

Introduction

miniMap (MM) is a multithreaded text processing language modeled on the concept of MapReduce. The purpose of miniMap is provide the user with a simple interface for efficiently manipulating text files in parallel.

MapReduce is a computational model that processes large datasets by first scaling down the data into smaller chunks over which the computations are carried out in parallel. Afterwards, all of the results get aggregated back into a single result. miniMap works in the same way: The user inputs a text file, specifies how they would like to split that file into smaller pieces and then inputs the function that they want to apply to the files and a function for aggregating the result. On the backend, miniMap takes all of those user inputs and processes the text over multiple threads and writes the final result into a file.

The inspiration for miniMap came from some Apache projects that are centered around Big Data. The first was (obviously) Hadoop. The second was Apache Spark, which uses Resilient Distributed Datasets (RDD) and runs large computation jobs IN memory. Hadoop, on the other hand, serializes the computations onto the actual file system. For our project we attempted to do a blend of serialization and in-memory computation.

The goal of our project was to be a useful tool for small-medium data. An example of this would be lowering the barrier of entry into parallel processing for researchers and students. Perhaps the ideal use-case would be a student who wants to work on NLP assignments and experiments--but doesn't have access to a Hadoop cluster. miniMap was designed not to be a replacement, but to allow someone to take greater advantage of their existing computer by providing a friendly introduction to MapReduce.

Language Tutorial

A. MiniMap interface

```
miniMap(File* inputFile, void* splitter(), void* mapper(),
File* context, void* reducer())
```
[1]

1. First, specify the path to the `inputFile` of interest.

```
inputFile = open("bigbang.txt", "r");
```

2. Next, use one of miniMap's built-in `splitter()` functions to scale down the text file into smaller chunks to be processed. `miniMap` provides three ways to split a file:

```
split_by_size(myfile, 500);
```

which takes the size of each chunk to be created in terms of characters (for example, to split the file into chunks of 500 characters)

```
split_by_quant(myfile, 10)
```

which takes in the number of chunks to be created (for example, to split the file into 10 chunks); and

```
split_by_regex(File* f, String s),
```

which takes in the file path and a regular expression on which the file will get splits (For example, to split the file after every number:

```
split_by_regex(user/desktop/inputFile.txt, “[0-9]”) )
```

3. Next, use the function header, `void mapper(File* f, File* g)`, to write the function to compute over the newly split text file.

[1] This is the goal, right now the pieces are in separate parts.

4. The context file is then used as a intermediary buffer to serialize data between the mapper and the reducer. This also allows the data from after the mapper to be cached and used/viewed later by the user.

5. Finally, use the function header, `void reducer(File* f)`, to write a function that aggregates all the results returned by the `mapper()` function. (For example

```
void reducer(File* f){

}
```

6. The `miniMap()`function itself returns `void` but the result of the computation will get written to a file.

B. Simple *Hello, World* program

```
int main()
{
      string s;
      s = "Hello World";
      printstring(s);
}
```

C. miniMap Hello, World program(s)

```
int main()
{
      file myfile;
      string test;

      myfile = open("bigbang.txt", "r");
      split_by_size(myfile, 500);

      test = "hello world";
      printstring(test);
      return 0;
}

int main()
{
```

```
        file inputFile;
        file chunkArrayPtr;
        file[9] chunkArr;
        string position;
        file tmp;
        int i;
        string line;


        inputFile = open("bigbang.txt", "r");
        chunkArrayPtr = split_by_size(inputFile, 1000);

        chunkArr[0] = open("smallFileName_0.txt", "r");
        chunkArr[1] = open("smallFileName_1.txt", "r");
        chunkArr[2] = open("smallFileName_2.txt", "r");
        chunkArr[3] = open("smallFileName_3.txt", "r");
        chunkArr[4] = open("smallFileName_4.txt", "r");
        chunkArr[5] = open("smallFileName_5.txt", "r");
        chunkArr[6] = open("smallFileName_6.txt", "r");
        chunkArr[7] = open("smallFileName_7.txt", "r");
        chunkArr[8] = open("smallFileName_8.txt", "r");

        for (i=0; i<8; i=i+1) {
              tmp = chunkArr[i];
              while(!isFileEnd(tmp))
              {
                          line = readFile(tmp, 100);
              }
              printstring(line);
        }

        printstring("hello");

}
```

Language Reference Manual

A. Lexical conventions

a. Identifiers

Identifiers consist of a string of case-sensitive ASCII letters, digits, underscores. The first character must be a letter.

b. Keywords

`if, else, for, while, return, true, false, file, %, #, len`

c. Delimiters/Separators and Whitespace:

   i. Tokens may be separated by whitespace characters and/or comments.
   ii. Single comment is supported:
      1. // my comment
   iii. Other delimiters:
      1. Delimiter characters
         a. Terminate statements: `;`
         b. Separating elements: `,`
         c. Strings: “”
      2. Parentheses
         a. Arrays: `[]`
         b. Blocks: `{}`
         c. Functions/Statements: `()`
      3. Newline characters:
         a. `\n {nl}`

B. Data types

miniMap supports the following data types:

a. Integers: `int x`
b. Booleans: `bool b`
c. Floats: `float f`
d. Strings: `String s`
e. Void types: `void y`
f. Files: `File* f`
g. Arrays: `a[]`
   i. Arrays can be of integer, floats, and files types

One thing to notice is that miniMap does not provide a ‘char’ data type. In miniMap, String is a pointer type of integer (i8 type in llvm) .

C. Variables

In miniMap, all variables need to be declared at the start and then defined on separate lines afterwards.

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

Variables in miniMap are mutable, except strings, they are immutable.

D. Expressions

a. Assignment syntax:

```
x = e
```

If x is mutable, then the assignment changes the current value of x to be the result of evaluating the expression e. The type of e is expected to conform to the type of x.

b. Operators, listed in the order of ascending precedence:
   Postfix () []
   Additive + -
   Multiplicative * /
   Assignment =
   Unary !
   Relational > >= < <=
   Equality == !=
   Bitwise AND &&
   Bitwise OR ||

E. Statements

a. Control Flow Statements

i. If statements

If the expression in if-clause evaluates to true, execute statements. If expression evaluates to false, execute statements in else-clause. The syntax is:

```
if (boolean expression) {
      /* List of statements */
} else {
      /* List of statements */
}
```

b. Loop Statements
   i. While loops

The list of statements in the block continue to be as long as the boolean expression evaluates to true. The syntax is:

```
while (boolean expression) {
      /* List of statements */
}
```

   ii. For loops

The list of statements in the loop are executed until condition in expression is no longer true. The syntax is:

```
for (expression) {
      /*List of statements */
}
```

F. Functions

When writing functions, functions declarations the the form:

```
return_type functionName(parameter_type
parameter_name)
{
      /* function body */
       return a;
}
```

Functions may take in a list of zero or more variables separated by commas, and a return type needs to be defined. Calling a function takes the form:

```
functionName(list of parameters);
```

G. Scope

Scope defines whether a variable or function is accessible at a given point in the program. In miniMap, there are no globally accessible variables. Functions have global scope and must be declared with unique identifiers and the scope for blocks (ie. body of if statements and loops) are defined by curly braces: `{}` Any identifiers declared within curly braces { } have a local scope and are not accessible from outside.

H. miniMap Standard Library

a. Print
   i. `print(x)`
      Prints x, integer type value
   ii. `printstring(x)`
      Prints x, string type value, defined in our language as pointer to integer pointer

b. File I/O (linked from C-library)
   i. a. `open(string "file_name.txt", string "r")`
      Takes a file path and file mode, and returns pointer to the file. Throws an IOException if file is not found.
   ii. b. `readFile(file file_name, int buffer_size)`
      Reads `file_name` of miniMap file type up to the size of buffer_size.
   iii. c. `isFileEnd(file file_name)`
      Returns true if end of file has been reached, and false otherwise.
   iv. d. `close(file file_name, int buffer_size)`
      Safely closes a file by freeing memory from file pointer and the buffer used. Throws an IOException if close() fails.

c. String (linked from C-library)
   i. `strstr(string x, string y)`
      Takes in an input string x, searches for the first occurrence of y in x and returns the rest of x starting from and including y.
   ii. `strcat(string x, string y)`

Appends string y to string x.[2]

iii. `strcpy(string x, string y)`
Copies string y into string x. [3]

d. Splitter functions
   i. `split_by_size(file myfile, int 500)`
   Splits a file into chunks by taking in the size of each chunk to be created in terms of characters
   ii. `split_by_quant(file myfile, int 10)`
   Splits a file into chunks by taking in the number of chunks to be created
   iii. `split_by_regex(File* f, String s)`
   Splits a file based on the input regular expression

e. Mapper functions
   i. `map(file inFile, file contextFile)`
   Function header for the user defined mapper function.

f. Reducer function
   i. `reduce(file contextFile, Array[] array )`
   Function header for the user defined reducer function.

g. MiniMap function
   i. `miniMap(file context, file[] splits, function mapper, function reducer)`
   Function header for the miniMap function.

Project Process

Members and roles

These were our initial, nominal roles of our members, but by the end, we had ended up all doing a bit of everything:

Charis Lam - Manager
Olesya Medvedeva - Language Guru
Ryan DeCosmo - Language Architect
Jyhyun Song - Tester

[2] This is what we would have liked to implement, but we had difficulties with the way we structured our string data types.
[3] “ ”

Process and Timeline

Right after the first class, we began discussing possible language ideas and by the proposal deadline we had what we thought was a pretty small and solid concept-- a MapReduce language that would harness a graphics card. That turned out to be way more complicated than we thought-- a recurring theme as we progressed through the language development. Structurally, we consistent set apart time to work together a few times each week--mostly in person, but occasionally through video call. After meeting the two in-class deadlines, we proceeded to set internal goals about what to work on each week, beginning with hello,world to implementing our own print functions and string data types to writing our built-in functions and linking to the C-library. Overall, we followed a rather iterative development structure. As we implemented and we figured out what we could and could not do, we repetetively went back to the drawing board to clarify and refine our understanding of our own language.

Software development environment

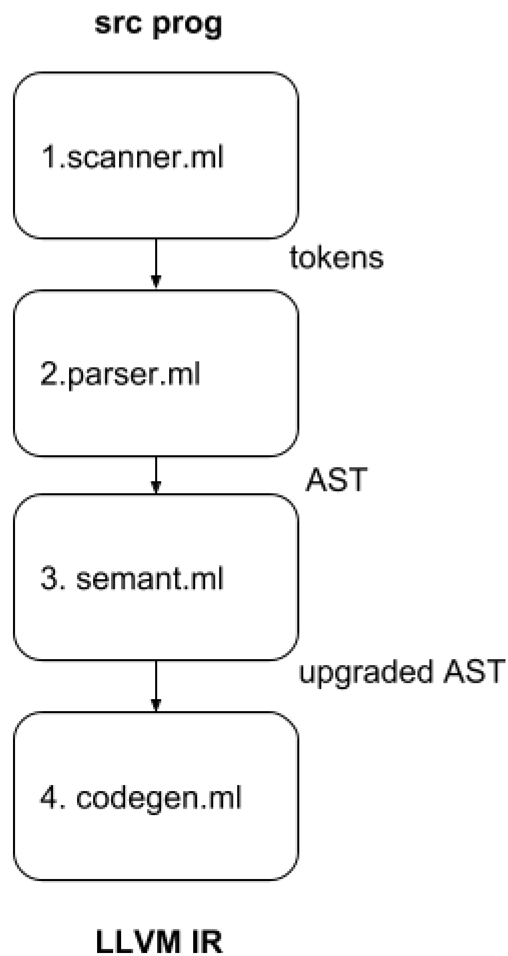
For text editing, we mainly used Atom. For version control of our code, we used GitHub: one master branch and then many sub branches for testing new developments to prevent major mishaps. And for other documents such as the concept proposal, language reference manual, useful readings, concept documents and other non-code related work, we used Google drive. For communication outside of in-person meetings, we mostly used GroupMe and occasionally Skype, but for the most part we met pretty consistently. All of us ran OSX and compiled code using the OSX terminal. Between our members we used a mix of LLVM 3.7 and 3.8, but for the most part we didn't run into any issues with switching between versions.

Programming style guide

For the most part we followed C-style conventions. In particular, we used camelCase for naming variables and functions, a one-tab indent for each substructure and we did not have any specific line length cut off as none of our lines became too unwieldy.

Project Log

Architectural Design



We have the all the components of a regular translator: a scanner, parser, semantic checker, a linker to the C-library for the File/IO functions and String functions and a code generator.

Scanner
Our scanner takes in a stream of characters as outlined in class and turns them into tokens that are passed into the parser.

Parser
The parser takes the tokens returned by the scanner and processes them into an Abstract Syntax Tree.

Semantic Analyser

The semantic checker goes through the functions and variable declarations to double check the duplications, names, list of formal arguments. It also checks the scoping of the variables to verify. Some built in functions were allocated special space in the analyzer to ensure that the user will not use the same names in their functions. And in these situations we alerted the user with warnings to update.

Code Generator

The code gen takes the AST that we generated earlier and translates it to LLVM IR code. We map the types and expressions between the two and loops / other control flow statements. This is all accomplished in OCaml with the LLVM module. Code gen is where we probably spent the most time trying to understand it.

Testers

Test 1

This is our Hello World tester that tests the printstring() function:

```
int main()
{
      string s;
      s = "Hello World";
      printstring(s);

      return 0;
}
```

Test 2

This is a File I/O tester that tests the open() File function as well as passing a File into the split_by_size function:

```
int main()
{
      file myfile;
      file outputfilelist;
      string test;

      myfile = open("bigbang.txt", "r");
      split_by_size(myfile, 500);

      /*outputfilelist = split_by_quant(myfile, 10);*/

      test = "hello world";
      printstring(test);

      return 0;
}
```

Test 3

This is a tester for accessing files in an array:

```
int addjamie(int a, int b)
{
      return a+b;
```

```
}

int main()
{
      file myfile;
      string line;

      myfile = open("text.txt", "r");

      while(!isFileEnd(myfile))
      {
            line = readFile(myfile,200);
            printstring(line);
      }

      close(myfile,line);
}
```

Test 4

This is a tester for getting the chunks as a pointer to the array of file pointers:

```
int main()
{
      int [2] a;
      file [2] s;
      string line;
      file myfile;

      string p;
      p = "Hello";

      myfile = open("text.txt", "r");
      s[1] = myfile;

      while(!isFileEnd(s[1]))
      {
            line = readFile(s[1],200);
            printstring(line);
      }

      close(s[1],line);
```

```
        a[0] = 1;

        print(a[0]);
        return 0;
}
```

Lessons Learned

Ryan

“It’s better to be overly ambitious with a simple idea than it is to be simple with an overly ambitious idea” - me

The most important learning outcome was understanding how programming languages work behind the scenes, what goes into a compiler, and what sort of trade-offs affect language design choices. I am glad I had the opportunity to use OCaml and see why it’s great for compilers. Lastly, witnessing the application of the theory we learned in lecture was encouraging to me to learn more theory.

Olesya

I think I got the grasp on the structure of the compiler. It definitely pays to go over the structure multiple times with the TAs and read the tutorials that were given in the beginning of the microc’s codegen file. Also realizing that there is no one to one translation of the data types in your language and LLVM helps to look for suitable alternatives. I think learning Ocaml as early as possible will pay off for sure. Also, deciding on the grammar of your language and sticking to the plan will save a lot of time.

Jamie

I finally understood the structure of the compiler and what each stage does after getting helloworld program to work. It took us over a week to get helloworld program working but the whole process accelerated when we discovered a way to debug. “./miniMap.native -a tests/test-helloworld.mm”. By using different tags, “-a”, “-l”, and “-c”, I could now know where our error was coming from, i.e. scanner, parser, or codegen.

I think the learning curve for this project was very steep and if I were to do it again, I would try to learn OCaml before starting the project. Also, studying previous semesters’ projects are very helpful and could reduce the learning time.

Charis

At the beginning I kept getting confused between what a programming language is and does and what a computer program is and does. Now I’m very clear on both, especially all the

parts and the function of each part of a programming language. Looking back it would have been good to dedicate more time to developing a solid idea of what our language needed and then laying out a project plan instead of jumping into the implementation first and developing goals as we went.

Appendix

AST:

```
(* Abstract Syntax Tree and functions for printing it *)

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

type uop = Neg | Not

(*We added a string here*)
type typ = Int | Bool | Void | String | File | Float | ArrayType of typ * int | ArrayPointer of
typ



type bind = typ * string

type expr =
    Literal of int
  | FloatLiteral of float
  | BoolLit of bool
  | StringSeq of string
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of expr * expr
  | PointerIncrement of string
  | Len of string
  | ArrayLiteral of expr list
  | ArrayAccess of string * expr
  | Call of string * expr list
  | ArrayReference of string
  | Dereference of string
```

```
  | 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 = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
  }

type program = bind list * func_decl list

(* Pretty-printing functions *)
(*Olessya: this is not the part of the AST itself, but the function to print it out!!!!!!*)

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_array m =
  let rec string_of_array_lit = function
```

```
      [] -> "]"
    | [hd] -> (match hd with
              Literal(i) -> string_of_int i
            | FloatLiteral(i) -> string_of_float i
            | BoolLit(i) -> string_of_bool i
            | Id(s) -> s
            | _ -> raise( Failure("Illegal expression in array literal") )) ^ string_of_array_lit []
    | hd::tl -> (match hd with
                  Literal(i) -> string_of_int i ^ ", "
                | FloatLiteral(i) -> string_of_float i ^ ", "
                | BoolLit(i) -> string_of_bool i ^ ", "
                | Id(s) -> s
                | _ -> raise( Failure("Illegal expression in array literal") )) ^
string_of_array_lit tl
  in
  "[" ^ string_of_array_lit m

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | FloatLiteral(i) -> string_of_float i
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | StringSeq(s) -> s
  | 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(v, e) -> v ^ " = " ^ string_of_expr e *)
  | Assign(r1, r2) -> (string_of_expr r1) ^ " =  " ^ (string_of_expr r2)
  | PointerIncrement(s) -> "++" ^ s
  | ArrayLiteral(m) -> string_of_array m
  | ArrayAccess(s, r1) -> s ^ "[" ^ (string_of_expr r1) ^ "]"
  | Len(s) -> "len(" ^ s ^ ")"
  | ArrayReference(s) -> "%" ^ s
  | Dereference(s) -> "#" ^ s
  | 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 rec string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Void -> "void"
  | Float -> "float"
  | String -> "string"
  | File -> "file"
  | ArrayType(t, i1) -> string_of_typ t ^ "[" ^ string_of_int i1 ^ "]"
  | ArrayPointer(t) -> string_of_typ t ^ "[]"


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

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  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 (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

CODEGEN:

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

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

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

Detailed documentation on the OCaml LLVM library:

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

```
*)

module L = Llvm
module A = Ast
open Exceptions

module StringMap = Map.Make(String)

let translate (globals, functions) =
  let context = L.global_context () in
  let the_module = L.create_module context "miniMap"
  and i8_t   = L.i8_type   context
  and i32_t  = L.i32_type  context
  and str_t  = L.pointer_type (L.i8_type context)
  and i1_t   = L.i1_type   context
  and float_t = L.double_type context
  and pointer_t = L.pointer_type
  and void_t = L.void_type context
  and array_t   = L.array_type
  and void_ptr =  L.pointer_type (L.i8_type context) in

  let ltype_of_typ = function
      A.Int -> i32_t
    | A.Bool -> i1_t
    | A.Float -> float_t
    | A.String -> str_t
    | A.Void -> void_t
    | A.File -> void_ptr
    | A.ArrayType(typ, size) -> (match typ with
                                     A.Int -> array_t i32_t size
                                   | A.Float -> array_t float_t size
                                   | A.Bool -> array_t i1_t size
                                   | A.File -> array_t void_ptr size
                                   | _ -> raise ( UnsupportedArrayType )
    )
    | A.ArrayPointer(t) -> (match t with
                                A.Int -> pointer_t i32_t
```

```
                                  | A.Float -> pointer_t float_t
                                  | _ -> raise (IllegalPointerType))
  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

  (* Declare the built-in printbig() function *)
  let printbig_t = L.function_type i32_t [| i32_t |] in
  let printbig_func = L.declare_function "printbig" printbig_t the_module in

  (* Declare the built-in split_by_size() function *)
  let split_by_size_t = L.function_type void_ptr [| str_t ; i32_t |] in
  let split_by_size_func = L.declare_function "split_by_size" split_by_size_t the_module
in

  (* Declare the built-in split_by_quant() function *)
  let split_by_quant_t = L.function_type void_ptr [| str_t ; i32_t |] in
  let split_by_quant_func = L.declare_function "split_by_quant" split_by_quant_t
the_module in

  let open_t = L.function_type void_ptr [| str_t; str_t|] in
  let open_func = L.declare_function "open" open_t the_module in

  let readFile_t = L.function_type str_t [| void_ptr; i32_t|] in
  let readFile_func = L.declare_function "readFile" readFile_t the_module in

  let isFileEnd_t = L.function_type i1_t [| void_ptr |] in
  let isFileEnd_func = L.declare_function "isFileEnd" isFileEnd_t the_module in

  let close_t = L.function_type i32_t [| void_ptr; void_ptr|] in
  let close_func = L.declare_function "close" close_t the_module in

  let strstr_t = L.function_type str_t [| str_t; str_t |] in
  let strstr_func = L.declare_function "strstr" strstr_t the_module in
```

```
  let miniMap_t = L.function_type i32_t [| void_ptr; L.pointer_type (L.function_type
(i32_t) [| L.i32_type  context; L.i32_type  context |] )|] in
  let miniMap_func = L.declare_function "miniMap" miniMap_t the_module in

  let miniMapNonThreaded_t = L.var_arg_function_type i32_t  [| void_ptr; void_ptr;
L.i32_type  context; L.pointer_type (L.function_type (i32_t) [| void_ptr|] );L.pointer_type
(L.function_type (i32_t) [| void_ptr|] )|] in
  let miniMapNonThreaded_func = L.declare_function "miniMapNonThreaded"
miniMapNonThreaded_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.typ) 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" "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

  let check_function =
     List.fold_left (fun m (t, n) -> StringMap.add n t m)
     StringMap.empty (globals @ fdecl.A.formals @ fdecl.A.locals)
  in

  let type_of_identifier s =
    let symbols = check_function in
    StringMap.find s symbols
  in

  let build_array_argument s builder =
    L.build_in_bounds_gep (lookup s) [| L.const_int i32_t 0; L.const_int i32_t 0 |] s
builder
  in

  let build_array_access s i1 i2 builder isAssign =
  if isAssign
    then L.build_gep (lookup s) [| i1; i2 |] s builder
  else
     L.build_load (L.build_gep (lookup s) [| i1; i2 |] s builder) s builder
  in
  let build_pointer_dereference s builder isAssign =
  if isAssign
    then L.build_load (lookup s) s builder
  else
    L.build_load (L.build_load (lookup s) s builder) s builder
  in
  let build_pointer_increment s builder isAssign =
  if isAssign
    then L.build_load (L.build_in_bounds_gep (lookup s) [| L.const_int i32_t 1 |] s
builder) s builder
  else
    L.build_in_bounds_gep (L.build_load (L.build_in_bounds_gep (lookup s) [|
L.const_int i32_t 0 |] s builder) s builder) [| L.const_int i32_t 1 |] s builder
 in
```

```
let rec array_expression e =
  match e with
  | A.Literal i -> i
  | A.Binop (e1, op, e2) -> (match op with
       A.Add     -> (array_expression e1) + (array_expression e2)
     | A.Sub     -> (array_expression e1) - (array_expression e2)
     | A.Mult    -> (array_expression e1) * (array_expression e2)
     | A.Div     -> (array_expression e1) / (array_expression e2)
     | _ -> 0)
  | _ -> 0
in

let find_array_type arr =
  match (List.hd arr) with
    A.Literal _ -> ltype_of_typ (A.Int)
  | A.FloatLiteral _ -> ltype_of_typ (A.Float)
  | A.BoolLit _ -> ltype_of_typ (A.Bool)
  | _ -> raise (UnsupportedArrayType) in

  (* Construct code for an expression; return its value *)
  let rec expr builder = function
      A.Literal i -> L.const_int i32_t i
    | A.FloatLiteral f -> L.const_float float_t f
    | A.StringSeq 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.ArrayLiteral s -> L.const_array (find_array_type s) (Array.of_list (List.map (expr
builder) s))
    | A.ArrayReference (s) -> build_array_argument s builder
    | A.Len s -> (match (type_of_identifier s) with A.ArrayType(_, l) -> L.const_int i32_t
l (* NEED TO CHANGE THIS!!!!!*)
                                   | _ -> L.const_int i32_t 0 )
    | A.Binop (e1, op, e2) ->
                    let e1' = expr builder e1
                    and e2' = expr builder e2 in
                    let float_bop operator =
                           (match operator with
                                A.Add     -> L.build_fadd
                              | A.Sub     -> L.build_fsub
                              | A.Mult    -> L.build_fmul
```

```
                    | A.Div     -> L.build_fdiv
                    | A.And     -> L.build_and
                    | A.Or      -> L.build_or
                    | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
                    | A.Neq     -> L.build_fcmp L.Fcmp.One
                    | A.Less    -> L.build_fcmp L.Fcmp.Olt
                    | A.Leq     -> L.build_fcmp L.Fcmp.Ole
                    | A.Greater -> L.build_fcmp L.Fcmp.Ogt
                    | A.Geq     -> L.build_fcmp L.Fcmp.Oge
                ) e1' e2' "tmp" builder
                  in

                         let int_bop operator =
                           (match operator 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
                         in

                        let string_of_e1'_llvalue = L.string_of_llvalue e1'
                        and string_of_e2'_llvalue = L.string_of_llvalue e2' in

                        let space = Str.regexp " " in

                        let list_of_e1'_llvalue = Str.split space
string_of_e1'_llvalue
                        and list_of_e2'_llvalue = Str.split space
string_of_e2'_llvalue in

                        let i32_re = Str.regexp "i32\\|i32*\\|i8\\|i8*\\|i1\\|i1*"
                        and float_re = Str.regexp "double\\|double*" in

                        let rec match_string regexp str_list i =
```

```
let length = List.length str_list in
match (Str.string_match regexp (List.nth str_list i)
0) with
  true -> true
| false -> if (i > length - 2) then false else
match_string regexp str_list (succ i) in

let get_type llvalue =
  match (match_string i32_re llvalue 0) with
    true  -> "int"
  | false -> (match (match_string float_re llvalue 0)
with
              true -> "float"
            | false -> "") in

let e1'_type = get_type list_of_e1'_llvalue
and e2'_type = get_type list_of_e2'_llvalue in

let build_ops_with_types typ1 typ2 =
  match (typ1, typ2) with
    "int", "int" -> int_bop op
  | "float" , "float" -> float_bop op
  | _, _ -> raise(IllegalAssignment)
in
build_ops_with_types e1'_type e2'_type
| A.Unop(op, e) ->
let e' = expr builder e in

let float_uops operator =
  match operator with
    A.Neg -> L.build_fneg e' "tmp" builder
  | A.Not -> raise(IllegalUnop)  in

let int_uops operator =
  match operator with
    A.Neg -> L.build_neg e' "tmp" builder
  | A.Not -> L.build_not e' "tmp" builder in

let bool_uops operator =
  match operator with
  A.Neg -> L.build_neg e' "tmp" builder
  | A.Not -> L.build_not e' "tmp" builder in
```

```
                    let string_of_e'_llvalue = L.string_of_llvalue e' in

                    let space = Str.regexp " " in

                    let list_of_e'_llvalue = Str.split space
string_of_e'_llvalue in

                    let i32_re = Str.regexp "i32\\|i32*"
                    and float_re = Str.regexp "double\\|double*"
                    and bool_re = Str.regexp "i1\\|i1*" in

                    let rec match_string regexp str_list i =
                      let length = List.length str_list in
                      match (Str.string_match regexp (List.nth str_list i)
0) with
                        true -> true
                      | false -> if (i > length - 2) then false else
match_string regexp str_list (succ i) in

                    let get_type llvalue =
                      match (match_string i32_re llvalue 0) with
                        true  -> "int"
                      | false -> (match (match_string float_re llvalue 0)
with
                                   true -> "float"
                                 | false -> (match (match_string bool_re
llvalue 0) with
                                              true -> "bool"
                                            | false -> "")) in

                    let e'_type = get_type list_of_e'_llvalue  in

                    let build_ops_with_type typ =
                     match typ with
                       "int" -> int_uops op
                     | "float" -> float_uops op
                     | "bool" -> bool_uops op
                     | _ -> raise(IllegalAssignment)
                    in

                    build_ops_with_type e'_type
  | A.Assign (e1, e2)  -> let e1' = (match e1 with
                          A.Id s -> lookup s
```

```
                              | A.ArrayAccess (s, e1) -> let i1 = expr builder e1 in (match
(type_of_identifier s) with
                                      A.ArrayType(_, l) -> (
                                        if (array_expression e1) >= l then
raise(ArrayOutOfBounds)
                                        else build_array_access s (L.const_int i32_t 0) i1
builder true)
                                      | _ -> build_array_access s (L.const_int i32_t 0) i1
builder true )
                              | A.PointerIncrement(s) -> build_pointer_increment s builder
true
                              | A.Dereference(s) -> build_pointer_dereference s builder
true
                              | _ -> raise (IllegalAssignment)
                              )
                    and e2' = expr builder e2 in
                ignore (L.build_store e2' e1' builder); e2'
    | A.ArrayAccess (s, e1) -> let i1 = expr builder e1 in (match (type_of_identifier s)
with
                                                    A.ArrayType(_, l) -> (
                                                      if (array_expression e1) >= l then
raise(ArrayOutOfBounds)
                                                      else build_array_access s (L.const_int
i32_t 0) i1 builder false)
                                                      | _ -> build_array_access s
(L.const_int i32_t 0) i1 builder false )
    | A.PointerIncrement (s) ->  build_pointer_increment s builder false
    | A.Dereference (s) -> build_pointer_dereference s builder false
    | A.Call ("print", [e]) | A.Call ("printb", [e]) ->
         L.build_call printf_func [| int_format_str ; (expr builder e) |]
          "printf" builder
    | A.Call ("printbig", [e]) ->
      L.build_call printbig_func [| (expr builder e) |] "printbig" builder
    | A.Call ("split_by_size", [e;f]) ->
      L.build_call split_by_size_func [| (expr builder e); (expr builder f)|] "split_by_size"
builder
    | A.Call ("split_by_quant", [e;f]) ->
      L.build_call split_by_quant_func [| (expr builder e); (expr builder f)|]
"split_by_quant" builder
    |  A.Call ("printstring", [e]) ->
      L.build_call printf_func [| str_format_str; (expr builder e) |]
      "printf" builder
    | A.Call ("open", [e1;e2]) ->
```

```
    L.build_call open_func [| (expr builder e1);(expr builder e2)|] "open" builder
  | A.Call ("readFile", [e1;e2]) ->
     L.build_call readFile_func [| (expr builder e1); (expr builder e2)|] "readFile"
builder
  | A.Call ("isFileEnd", [e1]) ->
     L.build_call isFileEnd_func [| (expr builder e1)|] "isFileEnd" builder
  | A.Call ("close", [e1;e2]) ->
     L.build_call close_func [| (expr builder e1); (expr builder e2)|] "close" builder
  | A.Call ("strstr", [e1;e2]) ->
    L.build_call strstr_func [| (expr builder e1); (expr builder e2)|]
     "strstr" builder
  |A.Call ("miniMap", [e1; A.Id(e2)]) ->
      let fileptr = expr builder e1 in

      let (fdef,_) = StringMap.find e2 function_decls in

      L.build_call miniMap_func [|fileptr; fdef |] "miniMap" builder

  |A.Call ("miniMapNonThreaded", [e1; e2; e3; A.Id(e4);A.Id(e5)]) ->
          let fileptr1 = expr builder e1 in
          let fileptr2 = expr builder e2 in
          let numFiles = expr builder e3 in

          let (fdef1,_) = StringMap.find e4 function_decls in
          let (fdef2,_) = StringMap.find e5 function_decls in

          L.build_call miniMapNonThreaded_func [|fileptr1; fileptr2; numFiles; fdef1
;fdef2 |] "miniMapNonThreaded" 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.typ 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.typ 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.typ 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
```

EXCEPTIONS

```
exception UnsupportedArrayType

exception IllegalAssignment

exception IllegalPointerType

exception ArrayOutOfBounds

exception IllegalUnop

exception WrongReturn
```

FILE LIBRARY C

```
/*
* Files I/O library
*
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>


void* open(void* filename1, char* mode)
{
   char *filename = (char *)filename1;
   FILE *mfile = fopen (filename , mode);
```

```
  //  int result = addjamie(1,3);
  //  printf("THis is printjamie: %i\n", result);
  //  split(mfile);

   return (void *)mfile;
}

void* readFile(void * fp_void, int size_buf)
 {
   FILE * fp = (FILE *) fp_void;
   char *buffer = malloc(size_buf+1);
   if ( fp != NULL )
   {
      fgets (buffer, sizeof buffer, fp);
   }
   else
   {
      return NULL;
   }
   return (void *) buffer;
 }

bool isFileEnd (void * fp_void)
{
  FILE * fp = (FILE *) fp_void;
   if (!feof(fp))
     return false;
   else
      return true;
}

//free memory from file pointer and the buffer used
 void closeFileBuffer(void * fp_void, void * buffer)
 {
   FILE * fp = (FILE *) fp_void;
   free(buffer);
   fclose(fp);
 }

 void freeBuffer(void* buffer)
 {
    free(buffer);
 }
```

```
 void close(void * fp_void)
 {
   FILE * fp = (FILE *) fp_void;
   fclose(fp);
 }


#ifdef BUILD_TEST
int main(void)
{
void * temp = open("text.txt", "r");
 FILE * fp = (FILE *) temp;

 char * buf;

 while(! isFileEnd(fp))
 {
   buf = readFile(fp, 200);
   printf("%s\n", buf );
 }


 close(fp,buf);
}
#endif


/*
@Ryan DeCosmo
@Olessya Medvedeva
*/

#include <stdio.h>
#include "NonThreaded.h"
#include <stdlib.h>


int main() {

    int numFiles = 4;
    FILE** files = malloc(numFiles * sizeof(FILE *));
    FILE *fp =  fopen("sample.txt", "r+");
```

```
    FILE *fpOut =  fopen("out.txt", "w+");

    miniMapNonThreaded(fpOut,fp, 1, map,reduce );

    return 0;
}
```

MAKEFILE:

```
# Make sure ocamlbuild can find opam-managed packages: first run
#
# eval `opam config env`


#@Charis Lam
#@Jamie Song
#@Ryan DeCosmo
#@Olessya Medvedeva


# Easiest way to build: using ocamlbuild, which in turn uses ocamlfind

.PHONY : all
all : miniMap.native printbig.o split_by_size.o fileLibrary.o miniMap.o

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

# "make clean" removes all generated files

.PHONY : clean
clean :
	ocamlbuild -clean
	rm -rf testall.log *.diff miniMap scanner.ml parser.ml parser.mli
	rm -rf printbig
	rm -rf split_by_size
	rm -rf fileLibrary
	rm -rf *.cmx *.cmi *.cmo *.cmx *.o *.s *.ll *.out *.exe
	find . -name "smallFileName_*" -print0 | xargs -0 rm
```

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

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

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

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 $<

# Testing the "printbig" example

printbig : printbig.c
	cc -o printbig -DBUILD_TEST printbig.c

split_by_size : split_by_size.c
	cc -o split_by_size -DBUILD_TEST split_by_size.c

fileLibrary : fileLibrary.c
	cc -o fileLibrary -DBUILD_TEST fileLibrary.c

miniMap : miniMap.c
	cc -o miniMap -DBUILD_TEST miniMap.c


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

```
miniMap.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
miniMap.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

# Building the tarball

TESTS = add1 arith1 arith2 arith3 fib for1 for2 func1 func2 func3    \
   func4 func5 func6 func7 func8 gcd2 gcd global1 global2 global3 \
   hello helloworld if1 if2 if3 if4 if5 local1 local2 ops1 ops2 var1 var2                    \
   while1 while2 split_by_size  fileLibrary miniMap1 miniMap2

FAILS = assign1 assign2 assign3 dead1 dead2 expr1 expr2 for1 for2        \
   for3 for4 for5 func1 func2 func3 func4 func5 func6 func7 func8   \
   func9 global1 global2 if1 if2 if3 nomain return1 return2 while1     \
   while2

TESTFILES = $(TESTS:%=test-%.mm) $(TESTS:%=test-%.out) \
            $(FAILS:%=fail-%.mm) $(FAILS:%=fail-%.err)

TARFILES = ast.ml codegen.ml Makefile _tags miniMap.ml parser.mly README \
       scanner.mll semant.ml testall.sh printbig.c fileLibrary.c miniMap.c split_by_size.c
arcade-font.pbm font2c \
          $(TESTFILES:%=tests/%)

miniMap-llvm.tar.gz : $(TARFILES)
          cd .. && tar czf miniMap-llvm/miniMap-llvm.tar.gz \
                    $(TARFILES:%=miniMap-llvm/%)
```

MINIMAP.c

```
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>
#include <stdbool.h>



void miniMap(FILE * inputFile, int (*func_ptr)(int,int))
{
   printf("%d\n", 100);

   func_ptr(2,3);

}
```

MINIMAP.ML

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

module StringMap = Map.Make(String)

type action = Ast | LLVM_IR | Compile

let _ =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./miniMap.native [-a|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
  let lexbuf = Lexing.from_channel !channel in
  let ast = Parser.program Scanner.token lexbuf in
  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)
```

NONTHREADED.C

```
/*
@Ryan DeCosmo
@Olessya Medvedeva
*/

#include "NonThreaded.h"


//compare for string functions
static int compare (const void * a, const void * b)
{
    return strcmp (*(const char **) a, *(const char **) b);
}


void tokenizeFile(FILE* splits, FILE* context){
    char delimit[]=" \t\r\n\v\f-,:;."; //const char s = ' ';
    char *token;

    if (splits != NULL) {
        char line [1000];
        while(fgets(line,sizeof line,splits)!= NULL)  {
            //  fprintf(stdout,"%s",line);
            //  char* one = "1";
            token = strtok(line, delimit);
            while( token  ) {
                fprintf(context,"%s \n", token ); //, one); //
                token = strtok(NULL, delimit);
            }
        }
    }
    else {
        perror("Error in mapper file");
    }
    rewind(context);

}
```

```
void countUniqueTokens(FILE* context, char** array){
    rewind(context);
    int totalstrings = 300;
    int stringsize = 50;
    char** words ;
    int freq[300];

    for (int i = 0; i < totalstrings; i++) {
        array[i] = (char *)malloc(stringsize);
    }

    char delimit[]=",1"; //const char s = ' ';
    char *token;

    if (context != NULL) {
        char line [1000];
        int j = 0;
        while(fgets(line,sizeof line,context)!= NULL)  {
            token = strtok(line, delimit);
            strcpy(array[j],token);
            token = strtok(NULL, delimit);
            j++;
        }
    }
    else {
        perror("Error");
    }
    //sort the array
    qsort (array, totalstrings, sizeof (const char *), compare);

    words = malloc(sizeof(char*)*totalstrings);

    for (int i = 0; i < totalstrings; i++) {
        words[i] = (char *)malloc(stringsize);
        freq[i] = 0;
    }

    //accumulator
    char** ptr = array;
    int counts =0;
    int i = 0;
    words[0] = *ptr;
```

```
    while(*ptr != 0) {
        // printf("%s \n", *ptr);
        if (strcmp(words[i], *ptr) == 0) {
            words[i] = *ptr;
            ++ptr;
            freq[i] = freq[i] + 1;
        } else {
            i++;
            words[i] = *ptr;
            freq[i] = freq[i] + 1;
            ++ptr;
        }
    }

    for (int i = 0; i < sizeof(freq) / sizeof(freq[0]); i++) {
        printf("Word: %s , Freq  :%i \n", words[i], freq[i]);
    }
}




void map(FILE* splits, FILE* context){
    //built in function to tokenize mini files
    tokenizeFile(splits,context);
}



void reduce(FILE* context, char** array){
    //function to accumulate values of the context file
    countUniqueTokens(context,array);
}




void miniMapNonThreaded(FILE* context, FILE* splits, int numberOfFiles, void
(*mapper)(), void (*reducer)()){


    char **array = malloc(300 * sizeof(char *));
```

```
    map(splits,context);
    reduce(context,array);

    fclose(splits);
    fclose(context);

}
```

NONTHREADED.H

```
//
// Created by Ryan DeCosmo on 12/19/17.
//

#ifndef UNTITLED_NONTHREADED_H
#define UNTITLED_NONTHREADED_H

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

void miniMapNonThreaded(FILE* context, FILE* splits, int numberOfFiles, void
(*mapper)(), void (*reducer)());
void map( FILE* file,FILE* context  );
void reduce(FILE* context, char** array);

#endif //UNTITLED_NONTHREADED_H
```

PARSER:

```
/* Ocamlyacc parser for miniMap */
/*
@Charis Lam
@Jamie Song
@Ryan DeCosmo
@Olessya Medvedeva
```

```
*/

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA
COLON
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token FILE INT BOOL STRING FLOAT VOID

%token RETURN IF ELSE FOR WHILE

/* Array related: Reference and Dereference and len */
%token OCTOTHORP PERCENT LEN

%token <float> FLOATLITERAL
%token <int> LITERAL
%token <string> STRING_SEQ
%token <string> ID
%token EOF

/* the presedence of the tokens, bottom is the highest */
/* associativity of the token, left, right or no assoc */
%nonassoc NOELSE
%nonassoc ELSE
%nonassoc NOLBRACK
%nonassoc LBRACK

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

/* specifies where your program starts */
%start program

/* defines what the program is, token program with the value of <Ast.program>*/
```

```
%type <Ast.program> program

%%


/*the syntax : symbol … symbol { semantic-action }*/

program:
  decls EOF { $1 }

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

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { typ = $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 }
  | BOOL { Bool }
  | FLOAT { Float }
  | VOID { Void }
  | STRING { String }
  | array_typ { $1 }
  | array_pointer_typ { $1 }
  | FILE { File }

array_typ:
    typ LBRACK LITERAL RBRACK %prec NOLBRACK  { ArrayType($1, $3) }
```

```
array_pointer_typ:
    typ LBRACK RBRACK %prec NOLBRACK { ArrayPointer($1)}

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

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

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

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:
   primitives      { $1 }
  | STRING_SEQ      { StringSeq($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) }
 | expr ASSIGN expr                    { Assign($1, $3)  }
 | LPAREN expr RPAREN { $2 }
 | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
 | LBRACK array_literal RBRACK  { ArrayLiteral(List.rev $2) }
 | LEN LPAREN ID RPAREN                   { Len($3) }
 | ID LBRACK expr  RBRACK %prec NOLBRACK        { ArrayAccess($1, $3)}
 | PERCENT ID                         { ArrayReference($2)}
 | OCTOTHORP ID                        { Dereference($2)}
 | PLUS PLUS ID                       { PointerIncrement($3) }

primitives:
   LITERAL                         { Literal($1)   }
 | FLOATLITERAL                         { FloatLiteral($1) }

array_literal:
   primitives                 { [$1] }
 | array_literal COMMA primitives { $3 :: $1 }


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

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

SCANNER.MLL

```
(* Ocamllex scanner for miniMap *)
(* @Charis Lam
 @Olessya Medvedeva
 @Jamie Song
```

```
  @Ryan DeCosmo
*)

{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)

(* Operators and separators *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '['      { LBRACK }
|  ']'     { RBRACK }
| '{'      { LBRACE }
| '}'      { RBRACE }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| ';'      { SEMI }
| ':'      { COLON }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }

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

(* Types *)
| "int"    { INT }
```

```
| "bool"   { BOOL }
| "float" { FLOAT }
| "void"   { VOID }
| "string" { STRING }
| "true"   { TRUE }
| "false"  { FALSE }
| "file"   { FILE }

(* Array related: Reference Dereference Len *)
| '%'     { PERCENT }
| '#'     { OCTOTHORP }
|  "len"  { LEN }

| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| '"'(('\\'_|[^'"'])* as lxm )'"'  { STRING_SEQ(lxm) } (* We added this, a regex for char /
num*... add special chars -ryan*)
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| ['0'-'9']* ['.'] ['0'-'9']+ | ['0'-'9']+ ['.'] ['0'-'9']* as lxm  { FLOATLITERAL(float_of_string lxm)
}
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

SEMANT.ML

```
(* Semantic checking for the miniMap compiler *)
(*
  @Olessya Medvedeva
  @Jamie Song
*)

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

  (**** 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"
   { typ = Void; fname = "print"; formals = [(Int, "x")];
     locals = []; body = [] } (StringMap.add "printb"
   { typ = Void; fname = "printb"; formals = [(Bool, "x")];
     locals = []; body = [] } (StringMap.add "printbig"
   { typ = Void; fname = "printbig"; formals = [(Int, "x")];
     locals = []; body = []} (StringMap.add "printstring"
   { typ = Void; fname = "printstring"; formals = [(String,"x")];
     locals = []; body = []})))
in *)

let built_in_decls = StringMap.add "print"
        { typ = Void; fname = "print"; formals = [(Int, "x")];
 locals = []; body = [] } built_in_decls in

let built_in_decls = StringMap.add "printb"
        { typ = Void; fname = "printb"; formals = [(Bool, "x")];
 locals = []; body = [] } built_in_decls  in

let built_in_decls = StringMap.add "printbig"
        { typ = Void; fname = "printbig"; formals = [(Int, "x")];
 locals = []; body = [] } built_in_decls in

let built_in_decls = StringMap.add "printstring"
        { typ = Void; fname = "printstring"; formals = [(String, "x")];
 locals = []; body = [] } built_in_decls in

let built_in_decls = StringMap.add "split_by_size"
{ typ = File; fname = "split_by_size"; formals = [(File, "x");(Int, "y")];
  locals = []; body = [] } built_in_decls in

let built_in_decls = StringMap.add "split_by_quant"
{ typ = File; fname = "split_by_quant"; formals = [(File, "x");(Int, "y")];
  locals = []; body = [] } built_in_decls in

let built_in_decls = StringMap.add "open"
        { typ =  File; fname = "open"; formals = [(String, "x"); (String, "y")];
        locals = []; body = [] } built_in_decls in

let built_in_decls = StringMap.add "readFile"
        { typ = String; fname = "readFile"; formals = [(File, "x"); (Int, "y")];
    locals = []; body = [] } built_in_decls in
```

```
 let built_in_decls = StringMap.add "isFileEnd"
 { typ = Bool; fname = "isFileEnd"; formals = [(File, "x")];
   locals = []; body = [] } built_in_decls in

 let built_in_decls = StringMap.add "close"
 { typ = Void; fname = "close"; formals = [(File, "x"); (String, "y")];
 locals = []; body = [] } built_in_decls in

  let built_in_decls = StringMap.add "strstr"
  { typ = String; fname = "strstr"; formals = [(String, "x"); (String, "y")];
  locals = []; body = [] } built_in_decls in

(*miniMap will be checked -> implement *)
  let built_in_decls = StringMap.add "miniMap"
  { typ = Void; fname = "miniMap"; formals = [(File, "x")];
  locals = []; body = [] } built_in_decls in

 (*will not be checked still*)
  let built_in_decls = StringMap.add "miniMapNonThreaded"
  { typ = Void; fname = "miniMapNonThreaded"; formals = [(File, "x");(File, "y");(Int,
"z")];
  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
  let array_access_type = function
  ArrayType(t, _) -> t
  | _ -> raise (Failure ("illegal array access") )
in

let check_pointer_type = function
    ArrayPointer(t) -> ArrayPointer(t)
  | _ -> raise ( Failure ("cannot increment a non-pointer type") )
in

let check_array_pointer_type = function
  ArrayType(p, _) -> ArrayPointer(p)
  | _ -> raise ( Failure ("cannont reference non-array pointer type"))
in


let pointer_type = function
  | ArrayPointer(t) -> t
  | _ -> raise ( Failure ("cannot dereference a non-pointer type")) in

let array_type s = match (List.hd s) with
  | Literal _ -> ArrayType(Int, List.length s)
  | FloatLiteral _ -> ArrayType(Float, List.length s)
  | BoolLit _ -> ArrayType(Bool, List.length s)
  | _ -> raise ( Failure ("Cannot instantiate a array of that type")) in

let rec check_all_array_literal m ty idx =
  let length = List.length m in
```

```
    match (ty, List.nth m idx) with
      (ArrayType(Int, _), Literal _) -> if idx == length - 1 then ArrayType(Int, length) else
check_all_array_literal m (ArrayType(Int, length)) (succ idx)
    | (ArrayType(Float, _), FloatLiteral _) -> if idx == length - 1 then ArrayType(Float,
length) else check_all_array_literal m (ArrayType(Float, length)) (succ idx)
    | (ArrayType(Bool, _), BoolLit _) -> if idx == length - 1 then ArrayType(Bool, length)
else check_all_array_literal m (ArrayType(Bool, length)) (succ idx)
    | _ -> raise (Failure ("illegal array literal"))
  in

    (* Return the type of an expression or throw an exception *)
    let rec expr = function
              Literal _ -> Int
      | BoolLit _ -> Bool
      | FloatLiteral _ -> Float
      | ArrayLiteral s -> check_all_array_literal s (array_type s) 0
      | StringSeq _ -> String
      | Id s -> type_of_identifier s
      | PointerIncrement(s) -> check_pointer_type (type_of_identifier s)
      | ArrayAccess(s, e1) -> let _ = (match (expr e1) with
                          Int -> Int
        | _ -> raise (Failure ("attempting to access with a non-integer type"))) in
        array_access_type (type_of_identifier s)
      | Len(s) -> (match (type_of_identifier s) with
          ArrayType(_, _) -> Int
        | _ -> raise(Failure ("cannot get the length of non-array")))
      | Dereference(s) -> pointer_type (type_of_identifier s)
      | ArrayReference(s) -> check_array_pointer_type( 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 when t1 = Int && t2 = Int -> Int
          | Equal | Neq when t1 = t2 -> Bool
          | Less | Leq | Greater | Geq when t1 = Int && t2 = Int -> Bool
          | And | Or when t1 = Bool && t2 = Bool -> Bool
        | _ -> raise (Failure ("illegal binary operator " ^
            string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
            string_of_typ t2 ^ " in " ^ string_of_expr e))
        )
      | Unop(op, e) as ex -> let t = expr e in
           (match op with
             Neg when t = Int -> Int
           | 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(e1, e2) as ex -> let lt = ( match e1 with
                                               | ArrayAccess(s, _) -> (match (type_of_identifier s)
with
                                                          ArrayType(t, _) -> (match t with
                                                                    Int -> Int
                                                                   | Float -> Float
                                                                   | File -> File
                                                                   | _ -> raise ( Failure
("illegal array") )
                                                                  )
                                                         | _ -> raise ( Failure ("cannot access
a primitive") )
                                                        )
                                               | _ -> expr e1)
              and rt = expr e2 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 fname <> "miniMap" then
      if fname <> "miniMapNonThreaded" then
        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.typ
  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.typ then () else
        raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                    string_of_typ func.typ ^ " 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

#include <stdio.h>
#include <string.h>
#include <regex.h>

/*
        Ryan DeCosmo
*/
//appropriated from: https://gist.github.com/ianmackinnon/3294587
//

void* split_by_regex(FILE *inputFile, char* argv[])
{

  size_t sizeFile = file_size(inputFile);

  char * regexString =  argv[1];
  char * source =       argv[2];
  size_t maxMatches = 10;
  size_t maxGroups = 3;

  regex_t regexCompiled;
  regmatch_t groupArray[maxGroups];
```

```
  unsigned int m;
  char * cursor;

  if (regcomp(&regexCompiled, regexString, REG_EXTENDED))
    {
      printf("Could not compile regular expression.\n");
      return 1;
    };

  m = 0;
  cursor = source;
  for (m = 0; m < maxMatches; m ++)
    {
      if (regexec(&regexCompiled, cursor, maxGroups, groupArray, 0))
        break;  // No more matches

      unsigned int g = 0;
      unsigned int offset = 0;
      for (g = 0; g < maxGroups; g++)
        {
          if (groupArray[g].rm_so == (size_t)-1)
            break;  // No more groups

          if (g == 0)
            offset = groupArray[g].rm_eo;

          char cursorCopy[strlen(cursor) + 1];
          strcpy(cursorCopy, cursor);
          cursorCopy[groupArray[g].rm_eo] = 0;
          printf("Match %u, Group %u: [%2u-%2u]: %s\n",
                 m, g, groupArray[g].rm_so, groupArray[g].rm_eo,
                 cursorCopy + groupArray[g].rm_so);
        }
      cursor += offset;
    }

  regfree(&regexCompiled);

  return 0;
}
```

SPLITBYSIZE

```
/*
  @Jamie Song
  @Ryan DeCosmo
*/
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

/*
Splitting large file into smaller files by file size
https://stackoverflow.com/questions/30222271/split-file-into-n-smaller-files-in-c
*/

// #define SEGMENT long 10 //approximate target size of small file

size_t file_size(FILE *inputFile);//function definition below

void* split_by_quant(FILE *inputFile, int quant)
{
  int j, len, accum;

  FILE *fp2;

  size_t sizeFile = file_size(inputFile);

  double segment = sizeFile / quant;

  char filename[260]={"smallFileName_"};//base name for small files.
  char smallFileName[260];
  char line[1080];

  FILE *outputFileList[quant];

  for (j=0; j<quant; j++)
  {
    accum = 0;
    sprintf(smallFileName, "%s%d.txt", filename, j);
    fp2 = fopen(smallFileName, "w");
    outputFileList[j] = fp2;
    if(fp2)
```

```
    {
      while(fgets(line, 1080, inputFile) && accum <= segment)
      {
        accum += strlen(line);//track size of growing file
        fputs(line, fp2);
      }
      // fclose(fp2);
    }

  }
  rewind(inputFile);

  // for (int i=0;i<quant; i++)
  // {
  //   printf("address of file is: %d\n", outputFileList[i]);
  // }
  // FILE ** res = &outputFileList;
  void * res ;
  return (void *) res;
}


FILE* split_by_size(FILE *inputFile, int size)
{
  double segment = (double)size;
  double segments = 0;
  int i, len, accum, quant;

  FILE *fp2;

  size_t sizeFile = file_size(inputFile);

  printf("sizeFile is %lu\n", sizeFile);
  // printf("segment is %f\n", segment);
  // printf("file_size / segment is: %f\n", sizeFile / segment);
  // printf("file_size / segment is: %i\n", sizeFile / segment);

  segments = sizeFile/segment ;//ensure end of file
  quant = (int) segments;
  printf("Number of chunks is %i\n", quant);

  char filename[260]={"smallFileName_"};//base name for small files.
  char smallFileName[260];
```

```
  char line[1080];

  // FILE *outputFileList[quant];

  if(inputFile)
  {
    for(i=0;i<quant;i++)
    {
      accum = 0;
      sprintf(smallFileName, "%s%d.txt", filename, i);
      fp2 = fopen(smallFileName, "w");
      if(fp2)
      {
        while(fgets(line, 1080, inputFile) && accum <= segment)
        {
          accum += strlen(line);//track size of growing file
          fputs(line, fp2);
        }
        // fclose(fp2);
      }
    }
  }
  rewind(inputFile);

  //need to change this to fit miniMap!
  FILE* outputFileList;

  return outputFileList;

}

size_t file_size(FILE* inputFile)
{
  size_t count = -1;  /* number of characters seen */

  while (inputFile)
  {
    /* character or EOF flag from input */
    int ch;
    ch = fgetc(inputFile);
    if (ch == EOF)
    {
      break;
```

```
    }
    ++count;
  }

  printf("Number of characters is %zu\n", count);
  rewind(inputFile);
  return count;
}
```

TESTALL:

```
#!/bin/sh

# Edited by: Jamie, Charis, Olessya, and Ryan

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

# Path to the LLVM interpreter
#LLI="lli"
#LLI="/usr/local/opt/llvm@3.8/bin/lli"
LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
#LLC="/usr/local/opt/llvm@3.8/bin/llc"
LLC="/usr/local/opt/llvm/bin/llc"

# Path to the C compiler
CC="cc"

# Path to the miniMap compiler.  Usually "./miniMap.native"
# Try "_build/miniMap.native" if ocamlbuild was unable to create a symbolic link.
MINIMAP="./miniMap.native"
#MINIMAP="_build/miniMap.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
```

```
error=0
globalerror=0

keep=0

Usage() {
   echo "Usage: testall.sh [options] [.mm files]"
   echo "-k    Keep intermediate files"
   echo "-h    Print this help"
   exit 1
}

SignalError() {
   if [ $error -eq 0 ] ; then
        echo "FAILED"
        error=1
   fi
   echo "  $1"
}

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

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

# RunFail <args>
# Report the command, run it, and expect an error
```

```
RunFail() {
   echo $* 1>&2
   eval $* && {
         SignalError "failed: $* did not report an error"
         return 1
   }
   return 0
}

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

   echo -n "$basename..."

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

   generatedfiles=""

   generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe
${basename}.out" &&
   Run "$MINIMAP" "$1" ">" "${basename}.ll" &&
   Run "$LLC" "${basename}.ll" ">" "${basename}.s" &&
   Run "$CC" "-o" "${basename}.exe" "${basename}.s" "split_by_size.o printbig.o
fileLibrary.o miniMap.o" &&
   Run "./${basename}.exe" > "${basename}.out" &&
   Compare ${basename}.out ${reffile}.out ${basename}.diff

   # Report the status and clean up the generated files

   if [ $error -eq 0 ] ; then
         if [ $keep -eq 0 ] ; then
            rm -f $generatedfiles
         fi
         echo "OK"
         echo "###### SUCCESS" 1>&2
   else
         echo "###### FAILED" 1>&2
         globalerror=$error
```

```
    fi
}

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

    echo -n "$basename..."

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

    generatedfiles=""

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

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
        if [ $keep -eq 0 ] ; then
            rm -f $generatedfiles
        fi
        echo "OK"
        echo "###### SUCCESS" 1>&2
    else
        echo "###### FAILED" 1>&2
        globalerror=$error
    fi
}

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

```
    esac
done

shift `expr $OPTIND - 1`

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

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

if [ ! -f printbig.o ]
then
    echo "Could not find printbig.o"
    echo "Try \"make printbig.o\""
    exit 1
fi

if [ ! -f split_by_size.o ]
then
    echo "Could not find split_by_size.o"
    echo "Try \"make split_by_size.o\""
    exit 1
fi

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

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

```
      *)
        echo "unknown file type $file"
        globalerror=1
        ;;
  esac
done

exit $globalerror
```