Democritus Language Final Report

Amy Xu
xx2152
Project Manager

Emily Pakulski
enp2111
Language Guru

Amarto Rajaram
aar2160
System Arhictect

Kyle Lee
kpl2111
Tester

May 11, 2016

Contents

1. Introduction

Democritus is a programming language with a static type system and native support for concurrent programming via its `atomic` keyword, with facilities for both imperative and functional programming. Democritus is compiled to the LLVM (Low Level Virtual Machine) intermediate form, which can then be optimized to machine-specific assembly code. Democritus' syntax draws inspiration from contemporary languages, aspiring to emulate Go and Python in terms of focusing on use cases familiar to the modern software engineer, emphasizing readability, and having "one – and preferably only one – obvious way to do it"[1].

1.1 Motivation

The main motivation behind Democritus was to create a lower level imperative language that supported concurrency 'out-of-the-box'. The race condition arising from threading would be solved by the language's `atomic` keyword, which provides a native locking system for variables and data. Users would then be able to write and run simple multi-threaded applications relatively quickly.

1.2 Product Goals

Native Concurrency and Atomicity

Users should be able to easily and quickly thread their program with minimal worry about race conditions. Their development process should not be hindered by the use of multithreading, nor should they have to define special threading classes as is common in some other languages.

Portability

Developed under the LLVM IR, code written in Democritus can be compiled and run on any machine that LLVM can run on. As an industrial-level compiler, LLVM offers robustness and portability as the compiler back-end of Democritus.

Flexibility

Though Democritus is not an object-oriented language, it seeks to grant users flexibility in functionality, supporting structures, standard primitive data types, and native string support.

[1]http://c2.com/cgi/wiki?PythonPhilosophy

2. Language Tutorial

Democritus is a statically-typed, imperative language with standard methods for conditional blocks, iteration, variable assignment, and expression evaluation. In this chapter, we will cover environment configuration as well as utilizing bothDemocritus' basic and advanced features.

2.1 Setup and Installation

To set up the Democritus compiler, OCaml and LLVM must be installed. Testing and development was done in both native Ubuntu 15.04 and Ubuntu 14.04 running on a virtual machine.

```
sudo apt-get install m4 clang-3.7 clang clang-3.7-doc libclang-common-3.7-dev libclang
    -3.7-dev libclang1-3.7 libclang1-3.7-dbg libllvm-3.7-ocaml-dev libllvm3.7 libllvm3
    .7-dbg lldb-3.7 llvm-3.7 llvm-3.7-dev llvm-3.7-doc llvm-3.7-examples llvm-3.7-
    runtime clang-modernize-3.7 clang-format-3.7 python-clang-3.7 lldb-3.7-dev liblldb
    -3.7-dbg opam llvm-runtime
```

For Ubuntu 15.04, we need the matching LLVM 3.6 OCaml Library.

```
sudo apt-get install -y ocaml m4 llvm opam
opam init
opam install llvm.3.6 ocamlfind
eval `opam config env`
```

For Ubuntu 14.04:

```
sudo apt-get install m4 llvm software-properties-common

sudo add-apt-repository --yes ppa:avsm/ppa
sudo apt-get update -qq
sudo apt-get install -y opam
opam init

eval `opam config env`

opam install llvm.3.4 ocamlfind
```

After setting up the environment, clone the git repository into your desired installation directory:

```
git clone https://github.com/DemocritusLang/Democritus.git
```

2.2 Compiling Your Code

To build the compiler, `cd` into the Democritus repository, and run `make`.

If building fails, try running `eval ‘opam config env‘`, which should update your local environment use OPAM packages and compilers. It's recommended to add the above command to your shell's configuration file if you plan on developing with Democritus.

To compile code, simply run

```
./Democritus < filename.dem > outfile.lli
```

To run compiled code, call `lli` on the output:

```
lli outfile.lli
```

2.3 Writing Code

Code can be written in any text file, but Democritus source files should have the `.dem` extension by convention. Democritus programs consist of global function, struct, and variable declarations. Only the code inside main() will be exectued at runtime. At this time, linking is not included in the Democritus compiler; all code should be written and compiled from a single `.dem` source file.

2.4 Getting Started

Declarations

Functions are declared with the `<function func_name(a type, b type) return_type>` syntax. Variables are declared with the `<let var_name var_type;>` syntax. Statements are terminated with the semicolon `;`. Note that all variable declarations must happen before statements (including assignments) in any given function.

```
function triangle_area(base int, height int) int{
  return base*height/2;
}
```

Types

Primitives

Primitive types in Democritus include booleans and integers. The void type is also used for functions.

Strings

Strings are built-in to Democritus. String literals are added to global static memory at runtime, and string variables point to the literals. These literals are automatically null-terminated.

```
function main() int{
  let s string;
  let foo int;
  let bar bool;

  bar = true;
  s = "Hello, World!"
  foo = 55;
  bar = false;

  return 0;
}
```

Structs

Structs are declared at the global level with the `<struct struct_type { named fields }>` syntax. Struct declarations may also be nested.

```
struct Person{
  let name string;
  let age int;
  let info struct Info;
}

struct Info{
  let education string;
  let salary int;
}


function main() int{
  let p struct Person;

  p.name = "Joe";
  p.age = 30;
  p.info.education = "Bachelor’s";
  p.info.salary = 99999;

  print(p.name);
  print(" earns: ");
  print_int(p.info.salary);

  return 0;
}
```

Operators

Democritus includes the ‘standard’ set of operators, defined as follows:

Binary Operators:

- artithmetic: `+, -, *, /, %`
- logical: `==, !=, <, <=, >, >=, && (and), || (or)`

Unary Operators:

- artihmetic: `-`
- logical: `! (not)`
- addressing: `& (reference), * (reference)`

Logical expressions return a boolean value.

The expressions on each side of a binary operation must be of the same type. The `&&, ||,` and `!` operators must be called on boolean expressions.

References can only be called on addressable fields (such as variables, or struct fields). Dereferences can only be called on pointer types.

2.5 Pointers

2.6 Control Flow

As an imperative language, Democritus executes statements sequentially from the top of any given function to the bottom. Branching and iteration is done similarly to many other imperative languages.

Conditional Branching

Conditional branching is done with:

```
if(boolean expression)
{
  /* do something here */
}

else
{
  /* do alternative here */
}
```

Here is an example of conditional branching in Democritus:

```
struct Person{
  let education string;
  let name string;
  let age int;
  let working bool;
}


function main() int{
  let p Struct person;
  p.name = "Joe"
  p.education = "Bachelor's";
  p.age = 25;
  p.working = false;

  if(p.working){
    print(p.name);
    print(" works.\n");
  }else{
    print(p.name);
    print(" is looking for work.\n");
  }

  return 0;
}
```

This program prints "Joe is looking for work."

Loops and Iteration

Iteration can be done either via a `for` loop. A `for(e1; e2; e3)` loop may take three expressions; `e1` is called prior to entering the loop, `e2` is a boolean conditional statement for the loop, and `e3` is called after each iteration. Both `e1` and `e3` are optional; omitting both converts the `for` loop into the conditional `while` used in other languages.

```
function main() int{
  let i int;
  for(i = 0; i<42; i=i+1){
    i = i+1;
  }

  for(;false;){
    // This block will never be reached
  }

  for(true){
    print_int(i);
  }
  return 0;
}
```

This program will print 42 forever.

2.7 Multithreading and Atomicity

Democritus supports threading with the `thread()` function call, which then calls the underlying `pthread` function in C. Any defined function can be called with multiple threads. The calling syntax is as follows:

```
thread("functionname", <comma separated args>, #threads);
```

Multithreaded functions must take a `*void` type as input and return a starvoid to conform with C's calling convention. An example of a multithreaded program:

```
function multiprint(noop *void) *void{
  let x starvoid;
  print("Hello, World!\n");
  return x;
}

function main() int{
  thread("multiprint", 0, 6); /* "Hello, World!" will be printed six times. */
  return 0;
}
```

2.8 Miscellaneous

Besides threading, a couple of other functions from C have been bound to Democritus.

Malloc

`malloc(size)` may be called, returning a pointer to a newly heap-allocated block of *size* bytes. These pointers may be bound to strings, which themselves are pointers to string literals. An example utilizing malloc will be included with file I/O.

File I/O

Files may be opened with `open()`. This call returns an integer, which may be then bound as a file descriptor. C functions such as `write()`, `read()`, or `lseek()` may then be called on the file descriptor.

- `open(filename, fd, fd2)`: opens a file. `filename` is a string referring to a file to be opened. `Fd` and `fd2` are file descriptors used for `open`.
- `write(fd, text, length)`: writes to a file. `fd` refers to the file descriptor of an open file. `text` is a string representing text to be written. `length` is an integer specifying the number of bytes to be written.
- `read(fd, buf, length)`: reads from a file. `fd` refers to the file descriptor of an open file. `buf` is a pointer to malloc'd or allocated space. `length` is an integer representing amount of data to be read. Buffers should be `malloc`'d before reading.
- `lseek(fd, offset, whence)`: sets a file descriptors cursor position. `fd` is the file descriptor to an open file. `offset` is an integer describing how many bytes the cursor should be offset by, and `whence` is an integer describing how `offset` should be applied: as an absolute location, relative location to the current cursor, or relative location to the end of the file.

For more detailed information on these calls, run `man function_name`.

```
function main() int{

  let fd int;
  let malloced string;

  fd = open("tests/HELLOOOOO.txt", 66, 384);   /* Open this file */
  write(fd, "hellooo!\n", 10);                  /* Write these 10 bytes */

  malloced = malloc(10);      /* Allocate space for the data and null terminator */
  lseek(fd, 0, 0);            /* Jump to the front of the file */
  read(fd, malloced, 10);     /* Read the data we just wrote into the buffer */

  print(malloced);            /* Prints "hellooo!\n" */
  return 0;
}
```

Sockets API

Democritus provides support for networking functionality through use of the C sockets API. A bound C function, `request_from_server`, allows a user to retrieve the contents of a webpage, written to a file, as follows:

```
function main() int
{
    let fd int;  //the file descriptor
    request_from_server("www.xkcd.com/index.html"); //write the content of the page to
         "index.html"
    exec_prog("/bin/cat", "cat", "tests/index.html"); //cat the file to stdout, used
        for testing
    return 0;
}
```

3. Language Reference Manual

3.1 Introduction

In this language reference manual, Democritus, its syntax, and underlying operating mechanisms will be documented. In the grammars shown in this reference manual, all terminals are expressed in uppercase and all nonterminals are kept lowercase. The Lexical Conventions section will detail terminals (also known as tokens).

3.2 Structure of a Democritus Program

A basic Democritus program reduces to a list of global variable, struct, and function declarations. Code 'to be executed' should be written in functions. These declarations are accessible and usable from any scope in a Democritus program. At runtime, the function `main()` will be executed.

The full grammar of a program is as follows:

```
program:
  decls EOF

decls:
   /* nothing */
 | decls vdecl
 | decls fdecl
 | decls sdecl

fdecl:
   FUNCTION ID LPAREN formals_opt RPAREN typ LBRACE vdecl_list stmt_list RBRACE

formals_opt:
    /* nothing */
  | formal_list

formal_list:
    ID typ
  | formal_list COMMA ID typ

typ:
    INT
  | FLOAT
  | BOOL
  | VOID
  | STRTYPE
  | STRUCT ID
  | VOIDSTAR
  | STAR %prec POINTER typ

vdecl_list:
    /* nothing */
  | vdecl_list vdecl

vdecl:
   LET ID typ SEMI

sdecl:
    STRUCT ID LBRACE vdecl_list RBRACE

stmt_list:
    /* nothing */
  | stmt_list stmt

stmt:
    expr SEMI
  | RETURN SEMI
  | RETURN expr SEMI
  | LBRACE stmt_list RBRACE
  | IF LPAREN expr RPAREN stmt %prec NOELSE
  | IF LPAREN expr RPAREN stmt ELSE stmt
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
  | FOR LPAREN expr RPAREN stmt

expr_opt:
    /* nothing */
  | expr

expr:
    LITERAL
  | FLOATLITERAL
  | TRUE
  | FALSE
  | ID
  | STRING
  | expr PLUS   expr
  | expr MINUS  expr
  | expr STAR   expr
  | expr DIVIDE expr
  | expr MOD    expr
  | expr EQ     expr
  | expr NEQ    expr
  | expr LT     expr
  | expr LEQ    expr
  | expr GT     expr
  | expr GEQ    expr
  | expr AND    expr
  | expr OR     expr
  | expr DOT    ID
  | expr DOT    ID ASSIGN expr
  | MINUS expr  %prec NEG
  | STAR expr   %prec DEREF
  | REF expr
  | NOT expr
  | ID ASSIGN expr
  | ID LPAREN actuals_opt RPAREN
  | LPAREN expr RPAREN

actuals_opt:
    /* nothing */
  | actuals_list

actuals_list:
    expr
  | actuals_list COMMA expr
```

3.3 Data types

Primitive Types

int

A standard 32-bit two's-complement signed integer. It can take any value in the inclusive range (-2147483648, 2147483647).

float

A 64-bit floating precision number, represnted in the IEEE 754 format.

boolean

A 1-bit true or false value.

pointer

A 64-bit pointer that holds the value to a location in memory; pointers may be passed and dereferenced.

Complex Types

string

An immutable array of characters, implemented as a native data type in Democritus. Pointers variables are 8-bit pointers to the location of the string literal in the global static memory.

struct

A struct is a simple user-defined data structure that holds various data types, such as primitives, other structs, or pointers.

3.4 Lexical Conventions

In this subsection, we will cover the standard lexical conventions for Democritus. Lexical elements are scanned as 'tokens,' which are then parsed into a valid Democritus program. Democritus is a free-format language, discarding all whitespace characters such as `' '`, `\t`, and `\n`.

Identifiers

Identifiers for Democritus will be defined as follows: any sequence of letters and numbers without whitespaces and not a keyword will be parsed as an identifier. Identifiers must start with a letter, but they may contain any lowercase or uppercase ASCII letter, numbers, and the underscore '_'. Identifiers are case-sensitive, so 'var1' and 'Var1' would be deemed separate and unique. Identifiers are used to identify named elements, such as variables, struct fields, and functions. Note that identifiers cannot begin with a number. The following is a regular expression for identifiers:

```
ID = "['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*"
```

Reserved Keywords

The following is a list of reserved Democritus keywords:

```
if          else        for         return      int
bool        void        true        false       string
struct      *void       function    let
```

Literals

Literals are used to represent various values or constants within the language.

Integer Literals

Integer literals are simply a sequence of ASCII digits, represented in decimal.

```
INT = "['0'-'9']+"
```

Boolean Literals

Boolean literals represent the two possible values that boolean variables can take, `true` or `false`. These literals are represented in lowercase.

```
BOOLEAN = "true|false"
```

String Literals

String literals represent strings of characters, including escaped characters. String literals are automatically null-terminated. Strings are opened and closed with double quotations. A special OCaml `lexbuf` was used to parse string literals.

```
(Taken from http://realworldocaml.org/v1/en/html/parsing-with-ocamllex-and-menhir.html)

read_string buf = parse
```

```
  | '"'              { STRING (Buffer.contents buf) }
  | '\\' '/'         { Buffer.add_char buf '/'; read_string buf lexbuf }
  | '\\' '\\'        { Buffer.add_char buf '\\'; read_string buf lexbuf }
  | '\\' 'b'         { Buffer.add_char buf '\b'; read_string buf lexbuf }
  | '\\' 'f'         { Buffer.add_char buf '\012'; read_string buf lexbuf }
  | '\\' 'n'         { Buffer.add_char buf '\n'; read_string buf lexbuf }
  | '\\' 'r'         { Buffer.add_char buf '\r'; read_string buf lexbuf }
  | '\\' 't'         { Buffer.add_char buf '\t'; read_string buf lexbuf }
  | [^ '"' '\\']+    { Buffer.add_string buf (Lexing.lexeme lexbuf);
                         read_string buf lexbuf
                     }
```

All Democritus Tokens

The list of tokens used in Democritus are as follows:

```
| "//"      { comment lexbuf }
| "/*"      { multicomment lexbuf }
| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
| '}'       { RBRACE }
| ';'       { SEMI }
| ','       { COMMA }
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { STAR }
| '&'       { REF }
| '.'       { DOT }
| '/'       { DIVIDE }
| '='       { ASSIGN }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "&&"      { AND }
| "||"      { OR }
| "!"       { NOT }
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "return"  { RETURN }
| "int"     { INT }
| "bool"    { BOOL }
| "void"    { VOID }
| "true"    { TRUE }
| "string"  { STRTYPE }
| "struct"  { STRUCT }
| "*void"   { VOIDSTAR }
| "false"   { FALSE }
| "function"  { FUNCTION }
| "let"       { LET }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '"'    { read_string (Buffer.create 17) lexbuf } (* refer to read_string above *)
| eof { EOF }
```

Punctuation

Semicolon

The semicolon ';' is required to terminate any statement in Democritus.

```
statement SEMI
```

Curly Brackets

In order to keep the language free-format, curly braces are used to delineate separate and nested blocks. These braces are required even for single-statement conditional and iteration loops.

```
LBRACE statements RBRACE
```

Parentheses

To assert precedence, expressions may be encapsulated within parentheses to guarantee order of operations.

```
LPAREN expression RPAREN
```

Comments

Comments may either be single-line, intialized with two backslashes, or multi-line, enclosed by `\* and *\`.

```
COMMENT = ("// [^'\n']* \n") | ("/*" [^ "*/"]* "*/")
```

3.5 Variable Declarations

In Democritus, local variables must be declared at the top of each function, before being later assigned.

Variable Declaration

Democritus requires all named variables to be declared with its type at the top of each function. Named variables are declared with the `let [ID] type` syntax. Assignment to these variables may then be done with `=`.

The grammar for variable declarations is as follows:

```
vdecl:
  LET ID typ SEMI

typ:
    INT
  | BOOL
  | VOID
  | STRTYPE
  | STRUCT ID
  | VOIDSTAR
  | STAR %prec POINTER typ
```

Struct Declaration

Structs are defined at the global scope, and can then be declared as variables. The global definitions are as follows:

```
sdecl:
    STRUCT ID LBRACE vdecl_list RBRACE

vdecl_list:
  | vdecl_list vdecl
```

3.6 Expressions and Operators

Expressions

Expressions may be any of the following:

Literal

A literal of any type, as detailed in the lexical conventions section.

Identifier

An identifier for a variable.

Binary Operation

A binary operation between an expression and another expression.

Unary Operation

A unary operation acting on the expression appearing on the immediate right of the operator.

Struct Access

An expression of a `struct` type accessing an identifier field with the `dot (.)` operator.

Struct Assignment

An expression of a `struct` type assigning a value to one of its fields (accessed with the `dot (.)` operator) using the `=` operator.

Function Call

A call to a function along with its formal arguments.

Variable Assignment

An identifier being assigned a value with the = operator.

Parenthisization

Another expression nested within parentheses.

The grammar for expressions is as follows:

```
expr:
    LITERAL
  | FLOATLITERAL
  | TRUE
  | FALSE
  | ID
  | STRING
  | expr PLUS   expr              (* expr TERMINAL expr are binary operations *)
  | expr MINUS  expr
  | expr STAR   expr
  | expr DIVIDE expr
  | expr MOD    expr
  | expr EQ     expr
  | expr NEQ    expr
  | expr LT     expr
  | expr LEQ    expr
  | expr GT     expr
  | expr GEQ    expr
  | expr AND    expr
  | expr OR     expr
  | expr DOT    ID                (* struct acccess     *)
  | expr DOT    ID ASSIGN expr    (* struct assign      *)
  | MINUS expr  %prec NEG         (* unary arith negate *)
  | STAR expr   %prec DEREF       (* unary deref        *)
  | REF expr                      (* unary ref          *)
  | NOT expr                      (* unary log negate   *)
  | ID ASSIGN expr
  | ID LPAREN actuals_opt RPAREN  (* function call      *)
  | LPAREN expr RPAREN            (* paren'd expr       *)
```

Binary and Unary Operations

A binary operation operates on the two expressions on the left and right side of the operator. Binary operations may be:

- an addition, subtraction, mult., division, or modulo on two arithmetic expressions (+,-,*,/,%). Modulo only works on integer types.
- equality or inequality expression between boolean expressions (==,!=,<,<=,>,>=,&&,||)

A unary operation operates on the expression on the operator's right side:

- a negation of an arithmetic expression (-)

- a dereference of a pointer type (`*`)
- an address reference of a variable or field within a struct (`&`)
- a negation of a boolean expression (`!`)

Arithmetic Operations

Democritus supports all the arithmetic operations standard to most general-purpose languages, documented below. Automatic casting has not been included in the language, and the compiler will throw an error in the case that arithmetic operations are performed between the same types of expressions.

Addition and Subtraction

Addition works with the `+` character, behaving as expected. Subtraction is called with `-`.

Multiplication and Division

Multiplication is called with `*`, and division with `/`. Division between integers discards the fractional part of the division.

Modulo

The remainder of an integer division operation can be computed via the modulo `%` operator.

Boolean Expressions

Democritus features all standard logical operators, utilizing `!` for negation, and `&&` and `||` for `and` and `or`, respectively. Each expression will return a boolean value of true or false.

Equality

Equality is tested with the `==` operator. Inequality is tested with `!=`. Equality may be tested on both boolean and arithmetic expressions.

Negation

Negation is done with `!`, a unary operation for boolean expressions.

Comparison

Democritus also features the `<`, `<=`, `>`, and `>=` operators. These represent less than, less than or equal to, greater than, and greater than or equal to, respectively. These operators are called on arithmetic expressions and return a boolean value.

Chained Expressions

Boolean expressions can be chained with `&&` and `||`, representing `and` and `or`. These operators have lower precedence than any of the other boolean operators described above. The `and` operator has a higher precedence than `or`.

Parentheses

Parentheses are used to group expressions together, since they have the highest order of precedence. Using parentheses will ensure that whatever is encapsulated within will be evaluated first.

Function Calls

Function calls are treated as expressions with a type equal to their return type. As an applicative-order language, Democritus evaluates function arguments first before passing them to the function. The grammar for function calls is as follows:

```
expr:
  .
  .
  .
  | ID LPAREN actuals_opt RPAREN   (* Function call *)

actuals_opt:
  /* nothing *?
  |  actuals_list

actuals_list:
     expr
  |  actuals_list COMMA expr
```

Pointers and References

Referencing and dereferencing operations are used to manage memory and addressing in Democritus. The unary operator `&` gives a variable or struct field's address in memory, and the operator `*` dereferences a pointer type.

Operator Precedence and Associativity

Precedence	Operator	Description	Associativity
1	`()`	Parenthesis	Left-to-right
2	`()`	Function call	Left-to-right
3	`*`	Dereference	Right-to-left
	`&`	Address-of	
	`!`	Negation	
	`-`	Unary minus	
4	`*`	Multiplication	Left-to-right
	`/`	Division	
	`%`	Modulo	
5	`+`	Addition	Left-to-right
	`-`	Subtraction	
6	`>> =`	For relational $>$ and $\geq$ respectively	Left-to-right
	`<<=`	For relational $<$ and $\leq$ respectively	
7	`== !=`	For relational $=$ and $\neq$ respectively	Left-to-right
8	`&&`	Logical `and`	Left-to-right
9	`||`	Logical `or`	Left-to-right
10	`=`	Assignment	Right-to-left

3.7 Statements

Statements written in a Democritus program are run from the top to bottom, sequentially. Statements can reduce to the following:

Expressions

An expression statement consists of an expression followed by a semicolon. Expressions in expression statements will be evaluated, with their values calculated.

Return Statements

A return statement is either a `RETURN SEMI` or `RETURN expr SEMI`. They are used as endpoints of a function, and control from a function returns to the original caller when a return statement is executed. Returns may be empty or return a type, though non-void functions must return an expression of their type.

Nested Blocks

A nested block is another statement list encapsulated within braces `{}`.

Conditional Statements

Conditional statements follow the `IF (boolean expr) stmt1 ELSE stmt2` format. When the `expr` evaluates to true, `stmt1` is run. Otherwise, if an `ELSE` and `stmt2` have been specified, `stmt2` is run.

Conditional Loops

A conditional loop is similar to a conditional statement, except in that it will loop or run repeatedly until its given boolean expression evaluates to `false`. In the case that the expression never evaluates to `false`, an infinite loop will occur.

Democritus eliminates the `while` keyword sometimes used in conditional iteration. Conditional loops follow the `FOR (expr1;boolean expr2;expr3) stmt` format where `expr1` and `expr3` are optional expressions to be evaluated prior to entering the `for` loop and upon each loop completion, respectively. Prior to entering or re-entering the loop, `expr2` is evaluated; control only transfers to `stmt` if this evaluation returns true. If both `expr1` and `expr3` are omitted, a simpler `for` loop can be written of the form `FOR (boolean expr) stmt`.

The full gammar for statements is as follows:

```
stmt_list:
    /* nothing */
  | stmt_list stmt

stmt:
    expr SEMI
  | RETURN SEMI
  | RETURN expr SEMI
  | LBRACE stmt_list RBRACE
  | IF LPAREN expr RPAREN stmt %prec NOELSE
  | IF LPAREN expr RPAREN stmt ELSE stmt
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
  | FOR LPAREN expr RPAREN stmt
```

3.8 Functions

Overview and Grammar

Functions can be defined in Democritus to return one or no data type. Functions are evaluated via eager (applicative-order) evaluation and the function implementation must directly follow the function header. The grammar for function declarations is as follows:

```
fdecl:
   FUNCTION ID LPAREN formals_opt RPAREN typ LBRACE vdecl_list stmt_list RBRACE

formals_opt:
    /* nothing */
  | formal_list

formal_list:
    ID typ
  | formal_list COMMA ID typ
```

All functions require `return` statements at the end, and must return an expression of the same type as the function. `void` functions may simply terminate with an empty `return` statement.

```
    }
function function_name([formal_arg type, ... ]) type_r {

    [function implementation]
    return [variable of type type_r]
}
```

Calling and Recursion

Functions may be recursive and call themselves:

```
function recursive_func(i int) void {

    if (i < 0) {
        return;
    } else {
        print(   h i   );
        recursive_func(i—1);    // Call ourselves again.
    }
}
```

Functions may be called within other functions:

```

function main() void{
    recursive_func(3);
    return;                     // Return nothing for void.
}
```

3.9 Concurrency

Overview

Democritus intends to cater to modern software engineering use cases. Developments in the field are steering us more and more towards highly concurrent programming as the scale at which software is used trends upward.

Spawning Threads

To spawn threads, Democritus uses a wrapper around the C-language `pthread` family of functions.

The $thread_t$ data type wraps $pthread_t$.

To spawn a thread, the thread function takes a variable number of arguments where the first argument is a function and the remaining optional arguments are the arguments for that function. It returns an error code.

The `detach` boolean determines whether or not the parent thread will be able to join on the thread or not.

```
      {
  function thread(f function, arg *void, arg, nthreads int) *void;
}
```

4. Project Plan

4.1 Planning

Much of the planning was facilitated by our weekly meetings with David Watkins, our TA. He very clearly explained what the requirements of each milestone entailed, and helped keep expectations transparent. Since Professor Edwards had emphasized the need for vertical development of features instead of horizontal building of each compiler layer, we quickly identified the key features that would be required to enable the key functionality of our language. Two of the most important components were structs and threads.

Our initial plan, which we largely followed, was to complete the Language Reference Manual, experiment with the layers of the compiler and get `Hello, World!` working, and then use what we had learned to begin implementing the more crucial aspects of the language.

4.2 Workflow

Workflow was facilitated by Git and GitHub, which allowed for the team to easily work on multiple features simultaneously and (usually) merge together features without overlapping conflicts. The Git workflow reached an optimal point by the conclusion of the project; new features would be developed, tested, and finalized in separate branches, and the commits for that feature would be squashed down until a single commit representing the new feature would be merged into the master branch.

Features were developed individually or through paired programming, depending on the scope and complexity of the feature. GitHub and division of labor allowed for many team members to work independently, at different times of the day per their own schedule. Branching allowed for one person to quickly deploy buggy code to another in hopes of resolving the issue, without any modification or bad commits to the master branch. GroupMe was used extensively for inter-team communication.

4.3 Team Member Responsibilities

Team Member	Responsibilities	GitHub Handle
Amy Xu	structs, nested structs, pointers, malloc	axxu
Emily Pakulski	C-bindings, reformatting tests, atomicity	ohEmily
Amarto Rajaram	C-bindings, pthread, file I/O, malloc	Amarto
Kyle Lee	structs, LRM, Final Report, debug and assist Amy	kyle–lee

4.4 Git Logs

5. Architecture Overview

Democritus' compiler is built off of Professor Stephen Edwards' `MicroC` compiler.

5.1 Compiler Overview

Several files make up the source code of the compiler. These include:

- `scanner.mll`: the OCamllex scanner.
- `ast.ml`: the abstract syntax tree, summarizing the overall structure of a Democritus program.
- `parser.mly`: the Ocamlyacc parser. Tokens from the scanner are parsed into the abstract syntax tree in the parser.
- `semant.ml`: the semantic analyzer.
- `codegen.ml`: the LLVM IR code generator.
- `democritus.ml`: the overarching OCaml program that calls the four main steps of the compiler.
- `bindings.c`: a C file that provides facilitates low-level operations that interact with the OS through C functions, such as for threads, which is then compiled to LLVM bytecode.

The Scanner

The scanner is simply a text scanner that parses text into various tokens, to then be interpreted by the parser. The regular expressions used by the scanner are listed in the language reference chapter.

The Parser

The parser is a token scanner that converts the tokens read into a valid abstract syntax tree of the program. If the program follows valid syntax, it will be parsed accordingly. Otherwise, compilation of code will yield a parse error. The structure of the program is as follows:

The Semantic Analyzer

The semantic analyzer checks the consistency and correctness of user programs. For example, it will check whether variables are defined within a scope, whether types of expressions match their uses in definitions and function calls, and whether structs are used correctly (a large modification we made was semantic checking for circular struct definitions).

The Code Generator

The code generator then takes in a definition of a program and builds the equivalent LLVM IR.

6. Testing

As with any software project, extensive testing was required to verify that all the features being implemented were working properly.

6.1 Integration Testing

Development and Testing Process

Development of new features required making them pass through the scanner, parser, semantic analyzer, and then code generation, in that order. When envisioning or developing a new feature, the testing process would proceed as follows:

1. Write example code implementing and utilizing the desired feature. (E.g. writing a struct definition in a new test file).
2. Modify the scanner (if needed) to read new tokens required by the new feature.
3. Modify the parser (usually needed) to change the grammar of the program to accept the new feature and pass necessary information (E.g. struct field names) to the semantic analyzer.
4. Modify the example code and test it so that only the 'correct' implementation of the feature passes the parser. Modify the scanner and parser until this step passes.
5. Modify the semantic analyzer so that it detects possible semantic issues that could arise from utilization of the new feature (E.g. accessing an undefined field in a struct or an undefined struct).
6. Modify the example code and test it so that only the 'correct' implementation passes the semantic analyzer; try testing multiple cases that should cause the analyzer to raise an error. Modify the semantic analyzer until this step passes.
7. Modify the code generator so that it generates the appropriate LLVM IR representing your new feature (E.g., allocating the correct amount of memory for new structs, building a map of struct field indexes, calling `LLVM.build_struct_gep`, etc.).
8. Modify your example code to utilize your feature and produce some visible effect or output (E.g. assigning a struct field, doing arithmetic on it, then printing it).
9. Test the code and ensure that running the program produces the expected output or effect; continue working on code generation until it does.

The process of writing test code, compiling it, and observing its output after being run as LLVM IR was the integration testing method that the Democritus team utilized throughout development. It helped ensure that whole features were working properly, and that the language, built up from multiple features, was still functioning correctly. Integration testing was done on all new features added to the language, as well as the existing ones from MicroC (such as basic variable assignment, conditional iteration, etc).

Aside: Unit Testing

Unit testing was not overly utilized in this development process, besides for testing to ensure that new features could pass certain layers of the compiler while working towards a passing integration test. This is because unit tests can still pass, while whole features lose vertical integration in the process of building up a compiler. This is because new features may often conflict with each other and the successful introduction of one feature could very well mean the breaking of another. This leads us to the test suite and automated regression testing.

6.2 The Test Suite and Automated Regression Testing

Democritus' test suite was built upon MicroC's automated regression testing package. Within the `tests` directory, there are dozens of integration test files for various language features as well as their expected `stdout` output. Additionally, there are several 'fail' tests used for showing invalid Democritus code as well as their expected error outputs.

The automated regression testing suite was used to quickly test all major language features by compiling each test, writing the error thrown by compilation (if it was a failure) or output of running the LLVM file (if compilation was a success) to a temporary file, and comparing that output to the expected output of each test with `diff`. The automated test was a shell script, invoked with `./testall.sh` in the Democritus root directory.

The test suite was used frequently throughout development; while developing new features, team members would utilize the test suite to ensure that all major features of the language were still working. If a certain test in the suite failed, more verbose information about the test's failure could be accessed in the `testall.log` file generated by the testing suite. The automated regression testing was crucial in ensuring that the language stayed consistent and working, and that our master branch remained 'updated' and error-free.

7. Lessons Learned

7.1 Amy

Trying to force new code to match legacy code can be more effort than it?s worth. It?s always okay to branch and attempt a larger rewrite if it will make everyone?s lives easier. Also, be sure to understand your own syntax when writing tests.

7.2 Emily

Remote teamwork can be tough. Writing tests that guarantee no regressions is surprisingly difficult, especially when testing against remote files.

7.3 Amarto

Debugging a compiler is like playing whack-a-mole – it?s much easier to write a script to isolate the action you?re trying to debug, and then gradually build it back into the compiler.

7.4 Kyle

In a team, try to play your strengths and figure out where you can help most effectively. If you think you can do something well or more efficiently than someone else, try to do it and save time - same thing works the other way (if pressed for time, let someone who knows how to do it manage it)

8. Code Listing

8.1 democritus.ml

```
(* Democritus, adapted from MicroC by Stephen Edwards Columbia University *)
(* Top-level of the MicroC compiler: scan & parse the input,
   check the resulting AST, generate LLVM IR, and dump the module *)

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)
```

8.2 scanner.mll

```
(* Democritus, adapted from MicroC by Stephen Edwards Columbia University *)
(* Ocamllex scanner *)

{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "//"     { comment lexbuf }              (* Comments *)
| "/*"     { multicomment lexbuf }            (* Multiline comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ':'      { COLON }
| ','      { COMMA }
| '+'      { PLUS }
| ’–’      { MINUS }
| ’*’      { STAR }
| ’%’    { MOD }
| ’&’    { REF }
| ’.’      { DOT }
| ’/’      { DIVIDE }
| ’=’      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| ’<’      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
|  "||"    { OR }
| "!"      { NOT }
| "["      { LEFTBR }
| "]"      { RIGHTBR }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "return" { RETURN }
| "int"    { INT }
| "float"  { FLOAT }
| "bool"   { BOOL }
| "void"   { VOID }
| "true"   { TRUE }
| "string" { STRTYPE }
| "struct" { STRUCT }
| "*void"  {VOIDSTAR }
| "false"  { FALSE }
| "function" { FUNCTION }
| "cast"   { CAST }
| "to"     { TO }
| "set"    { SET }
| "let"      { LET }
| [’0’–’9’]+[’.’][’0’–’9’]+ as lxm { FLOATLITERAL(float\_of\_string lxm) }
| [’0’–’9’]+ as lxm { LITERAL(int\_of\_string lxm) }
| [’a’–’z’ ’A’–’Z’][’a’–’z’ ’A’–’Z’ ’0’–’9’ ’\_’]* as lxm { ID(lxm) }
| ’"’      { read\_string (Buffer.create 17) lexbuf }
| eof { EOF }
| \_ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

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

(* From: realworldocaml.org/v1/en/html/parsing–with–ocamllex–and–menhir.html *)
and read\_string buf =
  parse
  | ’"’       { STRING (Buffer.contents buf) }
  | ’\\’ ’/’  { Buffer.add\_char buf ’/’; read\_string buf lexbuf }
  | '\\' '\\' { Buffer.add\_char buf '\\'; read\_string buf lexbuf }
  | '\\' 'b'  { Buffer.add\_char buf '\b'; read\_string buf lexbuf }
  | '\\' 'f'  { Buffer.add\_char buf '\012'; read\_string buf lexbuf }
  | '\\' 'n'  { Buffer.add\_char buf '\n'; read\_string buf lexbuf }
  | '\\' 'r'  { Buffer.add\_char buf '\r'; read\_string buf lexbuf }
  | '\\' 't'  { Buffer.add\_char buf '\t'; read\_string buf lexbuf }
  | [^ '"' '\\']+
    { Buffer.add\_string buf (Lexing.lexeme lexbuf);
      read\_string buf lexbuf
    }
  | \_ { raise (Failure("Illegal string character: " ^ Lexing.lexeme lexbuf)) }
  | eof { raise (Failure("String is not terminated")) }
```

8.3 parser.mly

```
/* Democritus, adapted from MicroC by Stephen Edwards Columbia University */
/* Ocamlyacc parser */

%{
open Ast;;

let first (a,\_,\_) = a;;
let second (\_,b,\_) = b;;
let third (\_,\_,c) = c;;
%}

%token LEFTBR RIGHTBR
%token COLON SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS STAR DIVIDE MOD ASSIGN NOT DOT DEREF REF
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token LET RETURN IF ELSE FOR INT FLOAT BOOL VOID STRTYPE FUNCTION STRUCT VOIDSTAR
    CAST TO SET
%token <string> STRING
%token <float> FLOATLITERAL
%token <int> LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc POINTER
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left STAR DIVIDE MOD
%right NOT NEG DEREF REF
%left DOT

%start program
%type <Ast.program> program

%%
```

```
program:
  decls EOF { $1 }

decls:
   /* nothing */ { [], [], [] }
 | decls vdecl { ($2 :: first $1), second $1, third $1 }
 | decls fdecl { first $1, ($2 :: second $1), third $1 }
 | decls sdecl { first $1, second $1, ($2 :: third $1) }

fdecl:
   FUNCTION ID LPAREN formals\_opt RPAREN typ LBRACE vdecl\_list stmt\_list RBRACE
     { { typ = $6;
   fname = $2;
   formals = $4;
   locals = List.rev $8;
   body = List.rev $9 } }

formals\_opt:
    /* nothing */ { [] }
  | formal\_list   { List.rev $1 }

formal\_list:
    ID typ                   { [($2,$1)] }
  | formal\_list COMMA ID typ { ($4,$3) :: $1 }

typ:
    INT { Int }
  | FLOAT { Float }
  | BOOL { Bool }
  | VOID { Void }
  | STRTYPE { MyString }
  | STRUCT ID { StructType ($2) }
  | VOIDSTAR { Voidstar }
  | STAR %prec POINTER typ { PointerType ($2) }

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

vdecl:
   LET ID typ SEMI { ($3, $2) }

sdecl:
    STRUCT ID LBRACE vdecl\_list RBRACE
      { { sname = $2;
      sformals = $4;
      } }

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) }
  | FOR LPAREN expr RPAREN stmt { While($3, $5) }

expr\_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    LITERAL          { Literal($1) }
  | FLOATLITERAL     { FloatLiteral($1) }
  | TRUE             { BoolLit(true) }
  | FALSE            { BoolLit(false) }
  | ID               { Id($1) }
  | STRING       { MyStringLit($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr STAR  expr  { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr MOD expr    { Binop($1, Mod,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | expr DOT    ID   { Dotop($1, $3) }
  | expr LEFTBR LITERAL RIGHTBR { ArrayRef($1, $3) }
  | CAST expr TO typ { Castop($4, $2) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | STAR expr %prec DEREF { Unop(Deref, $2) }
  | REF expr { Unop(Ref, $2) }
  | NOT expr         { Unop(Not, $2) }
  | expr 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 }
```

8.4 semant.ml

```
(* Democritus, adapted from MicroC by Stephen Edwards Columbia University *)
(* Semantic checking for compiler *)
```

```
open Ast

module StringMap = Map.Make(String)
module StringSet = Set.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, structs) =

(* 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 there is a recursive struct dependency*)

  let find\_sdecl\_from\_sname struct\_type\_name =
    try List.find (fun s-> s.sname= struct\_type\_name) structs
      with Not\_found -> raise (Failure("Struct of name " ^ struct\_type\_name ^ "not
          found."))
  in
  let rec check\_recursive\_struct\_helper sdecl seen\_set =
    let check\_if\_repeat struct\_type\_name =
      let found = StringSet.mem struct\_type\_name seen\_set in
      if found then raise (Failure ("recursive struct definition"))
      else check\_recursive\_struct\_helper (find\_sdecl\_from\_sname struct\_type\
          _name)  (StringSet.add struct\_type\_name seen\_set)
    in
    let is\_struct\_field = function
      (StructType s, \_) -> check\_if\_repeat s
     | \_ -> ()
    in
    List.iter (is\_struct\_field) sdecl.sformals
  in
  let check\_recursive\_struct sdecl =
     check\_recursive\_struct\_helper sdecl StringSet.empty
  in
  let \_ = List.map check\_recursive\_struct structs
  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 (String.compare (string\_of\_typ lvaluet) (string\_of\_typ rvaluet)) == 0
  then lvaluet
  else raise err
     (*if lvaluet == rvaluet then lvaluet else raise err*)
  in

  let match\_struct\_to\_accessor a b =
    let  s1 = try List.find (fun s-> s.sname=a) structs
      with Not\_found -> raise (Failure("Struct of name " ^ a ^ "not found.")) in
    try fst( List.find (fun s-> snd(s)=b) s1.sformals) with
  Not\_found -> raise (Failure("Struct " ^ a ^ " does not have field " ^ b))
  in

  let check\_access lvaluet rvalues =
     match lvaluet with
       StructType s -> match\_struct\_to\_accessor s rvalues
       | \_ -> raise (Failure(string\_of\_typ lvaluet ^ " is not a struct"))

  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 "append\_strings" (List.map (fun fd -> fd.fname) functions)
  then raise (Failure ("function append\_strings may not be defined")) else ();

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

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

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

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

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

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

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

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

  if List.mem "read" (List.map (fun fd -> fd.fname) functions)
```

```
 then raise (Failure ("function read may not be defined")) else ();

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

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

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

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

 if List.mem "memset" (List.map (fun fd -> fd.fname) functions)
 then raise (Failure ("function memset 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\_funcs = [
     { typ = Void; fname = "print\_int"; formals = [(Int, "x")];
     locals = []; body = [] };

     { typ = Void; fname = "printb"; formals = [(Bool, "x")];
     locals = []; body = [] };

     { typ = Void; fname = "print\_float"; formals = [(Float, "x")];
     locals = []; body = [] };

     { typ = Void; fname = "thread"; formals = [(MyString, "func"); (MyString, "arg")
         ; (Int, "nthreads")]; locals = []; body = [] };

     { typ = MyString; fname = "malloc"; formals = [(Int, "size")]; locals = []; body
          = [] };

    (* { typ = DerefAndSet; fname = "malloc"; formals = [(Int, "size")]; locals = [];
         body = [] }; *)

     { typ = Int; fname = "open"; formals = [(MyString, "name"); (Int, "flags"); (Int
         , "mode")]; locals = []; body = [] };

     { typ = Int; fname = "close"; formals = [(Int, "fd")]; locals = []; body = [] };

     { typ = Int; fname = "read"; formals = [(Int, "fd"); (MyString, "buf"); (Int, "
         count")]; locals = []; body = [] };

     { typ = Int; fname = "write"; formals =  [(Int, "fd"); (MyString, "buf"); (Int,
         "count")]; locals = []; body = [] };

     { typ = Int; fname = "lseek"; formals =  [(Int, "fd"); (Int, "offset"); (Int, "
         whence")]; locals = []; body = [] };

     { typ = Int; fname = "sleep"; formals =  [(Int, "seconds")]; locals = []; body =
          [] };
```

```
      { typ = Int; fname = "memset"; formals =  [(MyString, "s"); (Int, "val"); (Int,
          "size")]; locals = []; body = [] };

      { typ = MyString; fname = "request\_from\_server"; formals = [(MyString, "link")
          ]; locals = []; body = [] }
;

      { typ = Int; fname = "exec\_prog"; formals = [(MyString, "arg1"); (MyString, "
          arg2"); (MyString, "arg3") ]; locals = []; body = [] };

      { typ = Void; fname = "free"; formals = [(MyString, "tofree")]; locals = [];
          body = [] }
;

      { typ = Void; fname = "append\_strings"; formals = [(MyString, "str1"); (
          MyString, "str2")]; locals = []; body = [] };


      { typ = Void; fname = "int\_to\_string"; formals = [(Int, "n"); (MyString, "buf
          ")]; locals = []; body = [] }
]

  in

 let built\_in\_decls\_names = [ "print\_int"; "printb"; "print\_float"; "thread"; "
    malloc"; "open"; "close"; "read"; "write"; "lseek"; "sleep"; "memset"; "request\
    _from\_server"; "exec\_prog"; "free"; "append\_strings"; "int\_to\_string" ]

  in

  let built\_in\_decls = List.fold\_right2 (StringMap.add)
                        built\_in\_decls\_names
                        built\_in\_decls\_funcs
                        (StringMap.singleton "print"
                                { typ = Void; fname = "print"; formals = [(MyString, "
                                    x")];
                                locals = []; body = [] })

  in

  let function\_decls = List.fold\_left (fun m fd -> StringMap.add fd.fname fd m)
                         built\_in\_decls functions

  in

  let function\_decl s = try StringMap.find s function\_decls
       with Not\_found -> raise (Failure ("unrecognized function " ^ s))
  in

  let \_ = 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
      | FloatLiteral \_ -> Float
      | BoolLit \_ -> Bool
      | MyStringLit \_ -> MyString
      | Id s -> type\_of\_identifier s
      | Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
  (match op with
          Add | Sub | Mult | Div  when t1 = Int && t2 = Int -> Int
        |  Add | Sub | Mult | Div  when t1 = Float && t2 = Float -> Float
  | Mod 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))
        )
      | Dotop(e1, field) -> let lt = expr e1 in
         check\_access (lt) (field)
      | Castop(t, \_) -> (*check later*) t
      | ArrayRef (e, idx) -> let t = expr e in
         (match t with
                PointerType s -> s
    | \_ -> raise (Failure("cannot dereference a " ^ string\_of\_typ t)) )
      | Unop(op, e) as ex -> let t = expr e in
   (match op with
     Neg when t = Int -> Int
   | Not when t = Bool -> Bool
         | Deref -> (match t with
    PointerType s -> s
    | \_ -> raise (Failure("cannot dereference a " ^ string\_of\_typ t)) )
         | Ref -> PointerType(t)
   | \_ -> raise (Failure ("illegal unary operator " ^ string\_of\_uop op ^
           string\_of\_typ t ^ " in " ^ string\_of\_expr ex)))
      | Noexpr -> Void
     | 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.typ
      | Assign(e1, e2) as ex ->
  (match e1 with
    Id s ->
      let lt = type\_of\_identifier s 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))
    |Unop(op, \_) ->
      (match op with
        Deref -> expr e2
        |\_ -> raise(Failure("whatever"))
      )
    |Dotop (\_, \_) -> expr e2
    | \_ -> raise (Failure("whatever"))
  )

     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
```

8.5 ast.ml

```
(* Democritus, adapted from MicroC by Stephen Edwards Columbia University *)
(* Abstract Syntax Tree and functions for printing it *)

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

type uop = Neg | Not | Deref | Ref

type typ = Int | Float | Bool | Void | MyString | StructType of string | Voidstar |
    PointerType of typ

type bind = typ * string

type expr =
    Literal of int
  | FloatLiteral of float
  | BoolLit of bool
  | MyStringLit of string
  | Id of string
  | ArrayRef of string * int
  | Binop of expr * op * expr
  | Dotop of expr * string
  | Castop of typ * expr
  | Unop of uop * expr
  | Assign of expr * 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 = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
  }

type struct\_decl = {
    sname: string;
    sformals: bind list;

}
```

```
type program = bind list * func\_decl list * struct\_decl list

(* Pretty—printing functions *)

let string\_of\_op = function
    Add —> "+"
  | Sub —> "—"
  | Mult —> "*"
  | Div —> "/"
  | Mod —> "%"
  | Equal —> "=="
  | Neq —> "!="
  | Less —> "<"
  | Leq —> "<="
  | Greater —> ">"
  | Geq —> ">="
  | And —> "&&"
  | Or —> "||"

let string\_of\_uop = function
    Neg —> "—"
  | Not —> "!"
  | Deref —> "*"
  | Ref —> "&"

let rec string\_of\_typ = function
    Int —> "int"
  | Float —> "float"
  | Bool —> "bool"
  | Void —> "void"
  | MyString —> "string"
  | StructType(s) —> "struct" ^ s
  | Voidstar —> "voidstar"
  | PointerType(s) —> "pointerof" ^ (string\_of\_typ s)

let rec string\_of\_expr = function
    Literal(l) —> string\_of\_int l
  | FloatLiteral(l) —> string\_of\_float l
  | BoolLit(true) —> "true"
  | BoolLit(false) —> "false"
  | MyStringLit(s) —> s
  | Id(s) —> s
  | ArrayRef(s, l) —> s ^ "[" ^ string\_of\_int l ^ "]"
  | 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
  | Dotop(e1, e2) —> string\_of\_expr e1 ^ ". " ^ e2
  | Castop(t, e) —> "(" ^ string\_of\_typ t ^ ")" ^ string\_of\_expr e
  | Assign(v, e) —> string\_of\_expr v ^ " = " ^ 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\_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\_sdecl sdecl = sdecl.sname

let string\_of\_program (vars, funcs, structs) =
  String.concat "" (List.map string\_of\_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string\_of\_fdecl funcs) ^ "\n" ^
  String.concat "\n" (List.map string\_of\_sdecl structs)
```

8.6 codegen.ml

```
module L = Llvm
module A = Ast
module StringMap = Map.Make(String)

let translate (globals, functions, structs) =
  let context = L.global\_context () in
  let the\_module = L.create\_module context "Democritus"
  and i32\_t  = L.i32\_type  context
(*  and i8\_t   = L.i8\_type   context *)
  and i1\_t   = L.i1\_type   context
  and void\_t = L.void\_type context
  and ptr\_t  = L.pointer\_type (L.i8\_type (context))
  and float\_t = L.double\_type context
  in


  let struct\_types:(string, L.lltype) Hashtbl.t = Hashtbl.create 50 in

        let add\_empty\_named\_struct\_types sdecl =
    let struct\_t = L.named\_struct\_type context sdecl.A.sname in
    Hashtbl.add struct\_types sdecl.A.sname struct\_t
  in
  let \_  =
    List.map add\_empty\_named\_struct\_types structs
  in
```

```
let rec ltype\_of\_typ = function
  A.Int -> i32\_t
|       A.Float -> float\_t
|   A.Bool -> i1\_t
| A.Void -> void\_t
|   A.StructType s ->  Hashtbl.find struct\_types s
| A.MyString -> ptr\_t
|   A.Voidstar -> ptr\_t
| A.PointerType t -> L.pointer\_type (ltype\_of\_typ t) in
let populate\_struct\_type sdecl =
  let struct\_t = Hashtbl.find struct\_types sdecl.A.sname in
  let type\_list = Array.of\_list(List.map (fun(t, \_) -> ltype\_of\_typ t) sdecl.A.
      sformals) in
  L.struct\_set\_body struct\_t type\_list true
in
  ignore(List.map populate\_struct\_type structs);

let string\_option\_to\_string = function
None -> ""
| Some(s) -> s
in

(*struct\_field\_index is a map where key is struct name and value is another map*)
(*in the second map, the key is the field name and the value is the index number*)
let struct\_field\_index\_list =
let handle\_list m individual\_struct =
  (*list of all field names for that struct*)
  let struct\_field\_name\_list = List.map snd individual\_struct.A.sformals in
  let increment n = n + 1 in
  let add\_field\_and\_index (m, i) field\_name =
    (*add each field and index to the second map*)
    (StringMap.add field\_name (increment i) m, increment i) in
  (*struct\_field\_map is the second map, with key = field name and value = index*)
  let struct\_field\_map =
    List.fold\_left add\_field\_and\_index (StringMap.empty, -1) struct\_field\_name
        \_list
  in
  (*add field map (the first part of the tuple) to the main map*)
  StringMap.add individual\_struct.A.sname (fst struct\_field\_map) m
in
List.fold\_left handle\_list StringMap.empty structs
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

let append\_strings\_t = L.function\_type void\_t [| ptr\_t; ptr\_t |] in
let append\_strings\_func = L.declare\_function "append\_strings" append\_strings\_t
     the\_module in

let int\_to\_string\_t = L.function\_type void\_t [| i32\_t; ptr\_t |] in
let int\_to\_string\_func = L.declare\_function "int\_to\_string" int\_to\_string\_t
```

```
      the\_module in

  let printf\_t = L.var\_arg\_function\_type i32\_t [| ptr\_t |] in
  let printf\_func = L.declare\_function "printf" printf\_t the\_module in

  let execl\_t = L.var\_arg\_function\_type i32\_t [| ptr\_t |] in
  let execl\_func = L.declare\_function "exec\_prog" execl\_t the\_module in

  let free\_t = L.function\_type void\_t [| ptr\_t |] in
  let free\_func = L.declare\_function "free" free\_t the\_module in

  let malloc\_t = L.function\_type ptr\_t [| i32\_t |] in
  let malloc\_func = L.declare\_function "malloc" malloc\_t the\_module in

  let request\_from\_server\_t = L.function\_type ptr\_t [| ptr\_t |] in
  let request\_from\_server\_func = L.declare\_function "request\_from\_server"
     request\_from\_server\_t the\_module in

  let memset\_t = L.function\_type ptr\_t [| ptr\_t; i32\_t; i32\_t |] in
  let memset\_func = L.declare\_function "memset" memset\_t the\_module in

  (* File I/O functions *)
  let open\_t = L.function\_type i32\_t [| ptr\_t; i32\_t; i32\_t |] in
  let open\_func = L.declare\_function "open" open\_t the\_module in

  let close\_t = L.function\_type i32\_t [| i32\_t |] in
  let close\_func = L.declare\_function "close" close\_t the\_module in

  let read\_t = L.function\_type i32\_t [| i32\_t; ptr\_t; i32\_t |] in
  let read\_func = L.declare\_function "read" read\_t the\_module in

  let write\_t = L.function\_type i32\_t [| i32\_t; ptr\_t; i32\_t |] in
  let write\_func = L.declare\_function "write" write\_t the\_module in

  let lseek\_t = L.function\_type i32\_t [| i32\_t; i32\_t; i32\_t |] in
  let lseek\_func = L.declare\_function "lseek" lseek\_t the\_module in

  let sleep\_t = L.function\_type i32\_t [| i32\_t |] in
  let sleep\_func = L.declare\_function "sleep" sleep\_t the\_module in

  let default\_t = L.function\_type ptr\_t [|ptr\_t|] in
  let default\_func = L.declare\_function "default\_start\_routine" default\_t the\
     \_module in

  let param\_ty = L.function\_type ptr\_t [| ptr\_t |] in (* a function that returns
     void\_star and takes as argument void\_star *)
let param\_ptr = L.pointer\_type param\_ty in
let thread\_t = L.function\_type void\_t [| param\_ptr; ptr\_t; i32\_t|] in (*a
   function that returns void and takes (above) and a voidstar and an int *)
  let thread\_func = L.declare\_function "init\_thread" thread\_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 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 -> try StringMap.find n global\_vars
          with Not\_found -> raise (Failure ("undeclared variable " ^ n))
 in

 (* Construct code for an expression; return its value *)
 let rec llvalue\_expr\_getter builder = function
  A.Id s -> lookup s
|A.Dotop(e1, field) ->
 (match e1 with
  A.Id s -> let etype = fst(
   try List.find (fun t->snd(t)=s) fdecl.A.locals
   with Not\_found -> raise (Failure("Unable to find" ^ s ^ "in dotop")))
   in
   (try match etype with
    A.StructType t->
     let index\_number\_list = StringMap.find t struct\_field\_index\_list in
     let index\_number = StringMap.find field index\_number\_list in
     let struct\_llvalue = lookup s in
     let access\_llvalue = L.build\_struct\_gep struct\_llvalue index\_number "
       dotop\_terminal" builder in
     access\_llvalue

   | \_ -> raise (Failure("No structype."))
```

```
       with Not\_found -> raise (Failure("unable to find" ^ s)) )
    | \_ as e1\_expr ->  let e1'\_llvalue = llvalue\_expr\_getter builder e1\_expr in
      let loaded\_e1' = expr builder e1\_expr in
      let e1'\_lltype = L.type\_of loaded\_e1' in
      let e1'\_struct\_name\_string\_option = L.struct\_name e1'\_lltype in
      let e1'\_struct\_name\_string = string\_option\_to\_string e1'\_struct\_name\
          \_string\_option in
      let index\_number\_list = StringMap.find e1'\_struct\_name\_string struct\_field
          \_index\_list in
      let index\_number = StringMap.find field index\_number\_list in
      let access\_llvalue = L.build\_struct\_gep e1'\_llvalue index\_number "gep\_in\
          \_dotop" builder in
      access\_llvalue )


  |A.Unop(op, e)  ->
    (match op with
      A.Deref ->
        let e\_llvalue = (llvalue\_expr\_getter builder e) in
              let e\_loaded = L.build\_load e\_llvalue "loaded\_deref" builder in
        e\_loaded
      |\_ -> raise (Failure("nooo"))
    )
  |\_ -> raise (Failure ("in llvalue\_expr\_getter but not a dotop!"))
    and
    expr builder = function
  A.Literal i -> L.const\_int i32\_t i
(*      | A.MyStringLit str -> L.const\_stringz context str *)
      | A.FloatLiteral f -> L.const\_float float\_t f
      | A.MyStringLit 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.ArrayRef (e, l) ->
          let e\_llvalue = (llvalue\_expr\_getter builder e) in
          let e\_loaded = L.build\_load e\_llvalue "loaded\_deref" builder in
          e\_loaded
      | A.Binop (e1, op, e2) ->
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Add     -> (let e1\_type\_string = L.string\_of\_lltype (L.type\_of e1') in
        (match e1\_type\_string with
           "double" -> L.build\_fadd
                |"i32" -> L.build\_add
          | \_ -> raise(Failure("Can only add ints or floats")) ))
     |A.Sub     -> (let e1\_type\_string = L.string\_of\_lltype (L.type\_of e1') in
        (match e1\_type\_string with
           "double" -> L.build\_fsub
                |"i32" -> L.build\_sub
          | \_ -> raise(Failure("Can only subtract ints or floats")) ))
     |A.Mult     -> (let e1\_type\_string = L.string\_of\_lltype (L.type\_of e1') in
        (match e1\_type\_string with
           "double" -> L.build\_fmul
                |"i32" -> L.build\_mul
          | \_ -> raise(Failure("Can only multiply ints or floats")) ))
     |A.Div     -> (let e1\_type\_string = L.string\_of\_lltype (L.type\_of e1') in
        (match e1\_type\_string with
           "double" -> L.build\_fdiv
                |"i32" -> L.build\_sdiv
          | \_ -> raise(Failure("Can only divide ints or floats")) ))
    | A.Mod     -> L.build\_srem
    | 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.Dotop(e1, field) -> let \_ = expr builder e1 in
   (match e1 with
    A.Id s -> let etype = fst(
      try List.find (fun t->snd(t)=s) fdecl.A.locals
      with Not\_found -> raise (Failure("Unable to find" ^ s ^ "in dotop")))
      in
      (try match etype with
        A.StructType t->
          let index\_number\_list = StringMap.find t struct\_field\_index\_list in
          let index\_number = StringMap.find field index\_number\_list in
          let struct\_llvalue = lookup s in
          let access\_llvalue = L.build\_struct\_gep struct\_llvalue index\_number "
              dotop\_terminal" builder in
          let loaded\_access = L.build\_load access\_llvalue "loaded\_dotop\_terminal"
               builder in
          loaded\_access

        | \_ -> raise (Failure("No structype."))
       with Not\_found -> raise (Failure("unable to find" ^ s)) )
    | \_ as e1\_expr ->  let e1'\_llvalue = llvalue\_expr\_getter builder e1\_expr in
      let loaded\_e1' = expr builder e1\_expr in
      let e1'\_lltype = L.type\_of loaded\_e1' in
      let e1'\_struct\_name\_string\_option = L.struct\_name e1'\_lltype in
      let e1'\_struct\_name\_string = string\_option\_to\_string e1'\_struct\_name\
          \_string\_option in
      let index\_number\_list = StringMap.find e1'\_struct\_name\_string struct\_field
          \_index\_list in
      let index\_number = StringMap.find field index\_number\_list in
      let access\_llvalue = L.build\_struct\_gep e1'\_llvalue index\_number "gep\_in\
          \_dotop" builder in
      L.build\_load access\_llvalue "loaded\_dotop" builder )
      | A.Unop(op, e) ->
    let e' = expr builder e in
    (match op with
      A.Neg     -> L.build\_neg e' "tmp" builder
          | A.Not     -> L.build\_not e' "temp" builder
    | A.Deref -> let e\_loaded = L.build\_load e' "loaded\_deref" builder in
      e\_loaded
    | A.Ref -> let e\_llvalue = (llvalue\_expr\_getter builder e) in
    e\_llvalue
    )
```

```
     | A.Castop(ast\_cast\_type, e) ->
   let cast\_lltype = ltype\_of\_typ ast\_cast\_type in
   let e\_llvalue = expr builder e in
   L.build\_pointercast e\_llvalue cast\_lltype "plz" builder
     | A.Assign (lhs, e2) -> let e2' = expr builder e2 in
     (match lhs with
     A.Id s ->ignore (L.build\_store e2' (lookup s) builder); e2'
     |A.Dotop (e1, field) ->
       (match e1 with

         A.Id s -> let e1typ = fst(
         try List.find (fun t -> snd(t) = s) fdecl.A.locals
         with Not\_found -> raise(Failure("unable to find" ^ s ^ "in Sassign")))
         in
         (match e1typ with
           A.StructType t -> (try
             let index\_number\_list = StringMap.find t struct\_field\_index\_list in
             let index\_number = StringMap.find field index\_number\_list in
             let struct\_llvalue = lookup s in
             let access\_llvalue = L.build\_struct\_gep struct\_llvalue index\_number
                  field builder in
             (try (ignore(L.build\_store e2' access\_llvalue builder);e2')
               with Not\_found -> raise (Failure("unable to store " ^ t )) )
             with Not\_found -> raise (Failure("unable to find" ^ s)) )
           | \_ -> raise (Failure("StructType not found.")))
       |\_ as e1\_expr -> let e1'\_llvalue = llvalue\_expr\_getter builder e1\_expr
           in
         let loaded\_e1' = expr builder e1\_expr in
         let e1'\_lltype = L.type\_of loaded\_e1' in
         let e1'\_struct\_name\_string\_option = L.struct\_name e1'\_lltype in
         let e1'\_struct\_name\_string = string\_option\_to\_string e1'\_struct\_name
             \_string\_option in
         let index\_number\_list = StringMap.find e1'\_struct\_name\_string struct\
             \_field\_index\_list in
         let index\_number = StringMap.find field index\_number\_list in
         let access\_llvalue = L.build\_struct\_gep e1'\_llvalue index\_number "gep\
             \_in\_Sassign" builder in
         let \_ = L.build\_store e2' access\_llvalue builder in
         e2'
       )

      |A.Unop(op, e)  ->
         (match op with
           A.Deref ->
             let e\_llvalue = (llvalue\_expr\_getter builder e) in
                   let e\_loaded = L.build\_load e\_llvalue "loaded\_deref" builder
                       in
             let \_ = L.build\_store e2' e\_loaded builder in
             e2'
           |\_ -> raise (Failure("nooo"))
         )
      |\_ -> raise (Failure("can't match in assign"))
     )
     | A.Call ("print\_int", [e]) | A.Call ("printb", [e]) ->
   L.build\_call printf\_func [| int\_format\_str ; (expr builder e) |]
   "printf" builder
```

```
  | A.Call ("print\_float", [e]) ->
L.build\_call printf\_func [| float\_format\_str; (expr builder e) |] "printf"
   builder

  | A.Call ("print", [e])->
      L.build\_call printf\_func [| (expr builder e) |] "printf" builder

  | A.Call ("append\_strings", e) ->
let evald\_expr\_list = List.map (expr builder)e in
let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
L.build\_call append\_strings\_func evald\_expr\_arr "" builder

  | A.Call ("int\_to\_string", e) ->
let evald\_expr\_list = List.map (expr builder)e in
let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
L.build\_call int\_to\_string\_func evald\_expr\_arr "" builder

  | A.Call ("exec\_prog", e) ->
let evald\_expr\_list = List.map (expr builder)e in
let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
L.build\_call execl\_func evald\_expr\_arr "exec\_prog" builder

  | A.Call("free", e) ->
L.build\_call free\_func (Array.of\_list (List.map (expr builder) e)) "" builder

  | A.Call ("malloc", e) ->
      let evald\_expr\_list = List.map (expr builder)e in
let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
L.build\_call malloc\_func evald\_expr\_arr "malloc" builder

  | A.Call ("memset", e) ->
let evald\_expr\_list = List.map (expr builder)e in
let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
L.build\_call memset\_func evald\_expr\_arr "memset" builder

(* File I/O functions *)
  | A.Call("open", e) ->
let evald\_expr\_list = List.map (expr builder)e in
let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
L.build\_call open\_func evald\_expr\_arr "open" builder

  | A.Call("close", e) ->
let evald\_expr\_list = List.map (expr builder)e in
let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
  L.build\_call close\_func evald\_expr\_arr "close" builder

  | A.Call("read", e) ->
let evald\_expr\_list = List.map (expr builder)e in
let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
L.build\_call read\_func evald\_expr\_arr "read" builder

  | A.Call("write", e) ->
let evald\_expr\_list = List.map (expr builder)e in
let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
L.build\_call write\_func evald\_expr\_arr "write" builder
```

```
    | A.Call("lseek", e) ->
  let evald\_expr\_list = List.map (expr builder)e in
  let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
  L.build\_call lseek\_func evald\_expr\_arr "lseek" builder

    | A.Call("sleep", e) ->
  let evald\_expr\_list = List.map (expr builder)e in
  let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
  L.build\_call sleep\_func evald\_expr\_arr "sleep" builder

    | A.Call ("thread", e)->
(*  L.build\_call printf\_func [| int\_format\_str ; L.const\_int i32\_t 8 |] "printf"
     builder  *)
  let evald\_expr\_list = List.map (expr builder)e in
(*  let target\_func\_strptr = List.hd evald\_expr\_list in  (* jsut get the string by
     doing List.hd on e *)
  let target\_func\_str = L.string\_of\_llvalue target\_func\_strptr in *)
  let get\_string v = match v with
    | A.MyStringLit i -> i
    | \_ -> "" in
  let target\_func\_str = get\_string (List.hd e) in
  (*let target\_func\_str = Option.default "" Some(target\_func\_str\_opt) in *)
  let target\_func\_llvalue\_opt = L.lookup\_function target\_func\_str the\_module in
  let deopt x = match x with
    |Some f -> f
    | None -> default\_func in
  let target\_func\_llvalue = deopt target\_func\_llvalue\_opt in
  let remaining\_list = List.tl evald\_expr\_list in
  let new\_arg\_list = target\_func\_llvalue :: remaining\_list in
  let new\_arg\_arr = Array.of\_list new\_arg\_list in
    L.build\_call thread\_func
    new\_arg\_arr
                "" builder

    | A.Call ("request\_from\_server", e) ->
  let evald\_expr\_list = List.map (expr builder) e in
  let evald\_expr\_arr = Array.of\_list evald\_expr\_list in
  L.build\_call request\_from\_server\_func evald\_expr\_arr "request\_from\_server"
      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;

  let llmem = Llvm.MemoryBuffer.of\_file "bindings.bc" in
  let llm = Llvm\_bitreader.parse\_bitcode context llmem in
```

```
  ignore(Llvm\_linker.link\_modules the\_module llm Llvm\_linker.Mode.PreserveSource);

  the\_module
```

8.7 bindings.c

```
#include <pthread.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define BUFSIZE 4096

void append\_strings(void *str1, void *str2)
{
    strcat((char *)str1, (char *)str2);
}

void int\_to\_string(int n, void *buf)
{
    sprintf(buf, "%d", n);
}

int exec\_prog(void *str1, void *str2, void *str3)
{

    execl((char *)str1, (char *)str2, (char *)str3, NULL);
    return 0;
}

/*
 * Given a URL, send a GET request.
 */
//void *get\_request(void *url, void *filePath)
void *request\_from\_server(void *urlVoid)
{
    // www.xkcd.com/index.html
    char *urlStr = (char *) urlVoid;
    int idxslash = strchr(urlStr, ’/’) − urlStr;
    char *url = malloc(idxslash + 1);
    char *filePath = malloc(strlen(urlStr) − (idxslash) + 1);
    memset(url, 0, idxslash − 1);
    memset(filePath, 0, strlen(urlStr) − (idxslash));

    strncat(url, urlStr, idxslash);
    strncat(filePath, urlStr + idxslash, strlen(urlStr) − (idxslash));
    char *fileName = strrchr(urlStr, ’/’) + 1;

    char *serverIP;
    int sock;   // socket we connect to remote on
```

```
  struct sockaddr\_in serverAddr;
  struct hostent *he;
  char recvbuf[BUFSIZE];

  if ((he = gethostbyname((char *) url)) == NULL) {
fprintf(stderr, "gethostbyname() failed.");
      exit(1);
  }

  sock = socket(AF\_INET, SOCK\_STREAM, 0);
  if (sock < 0) {
      fprintf(stderr, "socket() failed.");
      exit(1);
  }
  serverIP = inet\_ntoa(*(struct in\_addr *)he->h\_addr);
  memset(&serverAddr, 0, sizeof(serverAddr));
  serverAddr.sin\_addr.s\_addr = inet\_addr(serverIP);
  serverAddr.sin\_family = AF\_INET;
  serverAddr.sin\_port = htons(80);

  int connected = connect(sock, (struct sockaddr *)&serverAddr, sizeof(serverAddr));
  if(connected < 0) {
fprintf(stderr, "connect() failed.");
      exit(1);
  }

  // send HTTP request
  if (((char *) url)[strlen((char *) url) - 1] == '/') {
      strcat(url, "index.html");
  }

  snprintf(recvbuf, sizeof(recvbuf),
          "GET %s HTTP/1.0\r\n"
          "Host: %s:%s\r\n"
          "\r\n",
          filePath, url, "80");
  if (send(sock, recvbuf, strlen(recvbuf), 0) != strlen(recvbuf)) {
      fprintf(stderr, "send() failed.");
      exit(1);
  }

  // wrap the socket with a FILE* so that we can read the socket using fgets()
  FILE *fd;
  if ((fd = fdopen(sock, "rb")) == NULL) {
fprintf(stderr, "fdopen() failed.");
      exit(1);
  }

  /* check header for valid protocol and status code */
  if (fgets(recvbuf, sizeof(recvbuf), fd) == NULL) {
      fprintf(stderr, "server terminated connection without response.");
      exit(1);
  }
  if (strncmp("HTTP/1.0 ", recvbuf, 9) != 0 && strncmp("HTTP/1.1 ", recvbuf, 9) !=
      0) {
fprintf(stderr, "unknown protocol response: %s.", recvbuf);
exit(1);
  }
  if (strncmp("200", recvbuf + 9, 3) != 0) {
fprintf(stderr, "request failed with status code %s.", recvbuf);
exit(1);
  }
  /* ignore remaining header lines */
  do {
      memset(recvbuf, 0, BUFSIZE);
if (fgets(recvbuf, sizeof(recvbuf), fd) == NULL) {
          fprintf(stderr, "server terminated connection without sending file.");
          exit(1);
}
  } while (strcmp("\r\n", recvbuf) != 0);


  char *filePathName = malloc(100);
  memset(filePathName, 0, 100);
  char *last\_slash;
  if ((last\_slash = strrchr(filePath, '/')) != NULL) {
      if (strlen(last\_slash) == 1) {
           strcpy(filePathName, "index.html");
      } else {
           strcpy(filePathName, last\_slash + 1);
      }
  }

  /* open and read into file */
  printf("%s\n", filePathName);
  FILE *outputFile = fopen(filePathName, "wb");
  if (outputFile == NULL) {
fprintf(stderr, "fopen() failed.");
      exit(1);
  }

  size\_t n;
  int total = 0;
  memset(recvbuf, 0, BUFSIZE);
  printf("buffer contents: %s\n", recvbuf);
  while ((n = fread(recvbuf, 1, BUFSIZE, fd)) > 0) {
if (fwrite(recvbuf, 1, n, outputFile) != n) {
    fprintf(stderr, "fwrite() failed.");
          exit(1);
}
      memset(recvbuf, 0, BUFSIZE);
total += n;
  }
  fprintf(outputFile, "\n");
  fprintf(stderr, "total bytes written: %d\n", total);

  if (ferror(fd)) {
fprintf(stderr, "fread() failed.");
      exit(1);
  }

  fclose(outputFile);
```

```
    fclose(fd);

    return NULL;
}

void *default\_start\_routine(void *arg)
{
    return arg;
}

void init\_thread(void *(*start\_routine) (void *), void *arg, int nthreads)
{
    pthread\_t thread[nthreads];
    int i;
    for (i = 0; i < nthreads; i ++) {
  pthread\_create(&thread[i], NULL, start\_routine, arg);
    }

    for (i = 0; i < nthreads; i++) {
  pthread\_join(thread[i], NULL);
    }
}
```