PLT final report

Guihao Liang, Jincheng Li, Xue Wang, Zizhang Hu[1]
Columbia University, 2016

Introduction

Harmonica is a general-purpose programming language with features inspired by Python, Go and functional languages. It is statically typed, supports first-class functions and has easy-to-use multithreading capabilities. It is built for performing fast parallel scientific computations and developing multithreaded applications.

The Harmonica compiler is written in OCaml and compiles Harmonica source code to an intermediate LLVM representation (LLVM IR). LLVM IR can be then interpreted or compiled to the target platform.

Motivation

Scientific computation on massive datasets are becoming the norm, and Python is an increasingly popular language within this community for its powerful built-in language features and ease of use. However, one of Python's major pain points is that it does not support concurrent execution of multiple threads. Harmonica aims to borrow from another language, Go, to complement Python's lack of multi-threading capabilities and introduce a simple language for both scientific computations and general multithreaded applications.

Features

Notable features of Harmonica include simple management of threads, synchronization primitives (mutex), first-class functions, lambda expressions, preprocessor macros, a standard math library, and a thread-safe binary search tree library.

Language Reference Manual

1. **Lexical Elements**
    a. Identifiers
    Identifiers are arbitrary strings starting with [a-z A-Z] and followed by [a-z A-Z 0-9] (not underscores). We recommend camelCase.
    b. Key words
    `if, else, elseif, bool, int, float, string, list, struct, # (comment), return, parallel, channel, import` and various operators defined below.
    c. Constants
    A constant is a literal numeric or character value, such as 5 or 'm'

[1] Order by first name

   i. Integer: decimal integers, [1-9]+[0-9]*
   ii. Float: real numbers following the C standard. Examples are 1.4, -0.3, 5e2, etc.
   iii. String: a string instance enclosed with double quotation marks.

2. Data Types

2.1 primitive types

a. Int
   Integer type. We only support 32-bit integers, ranging from -2^31 to 2^31-1.
b. Float
   Floats are 8-byte double-precision floating point numbers.
c. Boolean
   Boolean is just a boolean, true or false.
d. String
   String is a sequence of ascii characters. Its length is limited by maximum integer value, which is 2^31-1. For string indexing, not yet implemented.

2.2 compound types

a. List[]
   List is a sequence of elements of the same type, with maximum length 2^31-1. Lists are implemented as basic arrays (similar to C). There is not support for getting the size of a list or dynamically increasing available memory space.

```
[1]; # literal
list lst = []; # assign empty list
list lst;  # declare and assign
lst = [1, 2];
```

   More support on dynamic array in standard library `lib/vector.ha`.
   For `int` vectors, you can make use of built-ins below:

```
# declare a vector_int variable first;
vector_int v;
# create a dynamic list object on the fly
vector_int_create();
# append an int element to dynamic list
vector_int_append(vector_int, int ele);
# indexing element in list
vector_int_get(int index);
# remove element at certain index
vector_int_remove(int index);
# destroy the whole list
vector_int_destroy(vector_int v);
```

Of course you can declare different types of dynamic list instead of int list. For more information, please refer template and preprocessor part.

b. Struct

Struct is a composite data structure supporting member variables and methods.

```
struct foo {
      int a;
      float b;
};

foo v;
v.a = 1
```

You can also set member as functions since functions are first class in our language.

```
# declare a function.
bool baz(int a) { return a; }

struct foo {
      <int bool> bar;
};

foo.bar = baz;
printi(foo.bar(10)); # prints 10 to stdout.
```

c. Function Types

In Harmonica, functions are first class members that can be passed around as regular variables. And function types are defined by the types of their input parameters and their return type. The notation for function types are simply a list of types separated by space, enclosed in angle brackets.

The last type in the sequence is assumed to be the function's return type. For example, the type `<Int Int List[Int]>` is the type of a function that takes two integers and returns a list of integers.

3. Expressions and Operators

a. Expressions

The definition of expression is same with C. An *expression* consists of at least one operand and zero or more operators.

Operands are typed objects such as constants, variables, and function calls that return values. *Operators* specify an operation to be applied on its operands.

b. Assignment Expressions
   Assignment expression stores value to the variable.

   The standard assignment operator = simply stores the value of its right operand in the variable specified by its left operand. As with all assignment operators, the left operand (commonly referred to as the “lvalue”) cannot be a literal or constant value, it should should be a variable that can be modified.

```
int x = 10;
float y = 45.12 + 2.0;
list[int] z = [4,5,6];
```

   You can also use the plain assignment expression to store values of a structure type.

   Also, we support compound assignments, such as +=, -=, *=, /=, %=, <<=, >>=, >>=, &=, ^=, |=.
   `a += b` is equal to `a = a + b;`

c. Arithmetic operators
   Unary operators are - +, with highest priority, followed by binary + - % operators, followed by binary + - operations.

d. Comparison operators
   Binary logical and relational operators: ==, , >, >=, !=, <=, <. These operators do shallow comparison, meaning the lvalue and rvalue.

   For structure, the equality comparison will compare each field by value, here’s an instance:

```
struct float_string {
      string str = “str”;
      float   flt = 1.0;
};
float_string a, b;
```

   `a == b` is equal to `a.str == b.str && a.fl t == b.flt.`
   Like java, we don’t propagate other non-zero type into bool. That is, in C or C++, non-zero can represent `true`, while in our language, you should use `a.flt > 0` explicitly.

e. Array access
   Arrays can be accessed with an index number enclosed in square brackets. For example, `arr[3]` accesses the fourth element of the array.

f. Function calls as expressions
Functions which return values can be expressions.
```
int function(int);
a = 9 + function(9);
```

g. Comma Operator
You use the comma operator , to separate two expressions. The first expression must take effect before being used in the second expression.
```
int x = 1, y = 2;
x += 1, y += x;
```
The return value of comma expression is the value of second expression. In the above example, the return value should be 4.

If you want to use comma expression in function, you should use it with parentheses because in function call, comma has a different meaning, separating arguments.

h. Member Access Expressions
You can use access operator dot . to access the members of a structure variable.
```
struct foo {
     int x, y;
};
struct foo bar;
bar.x = 0;
```

i. Conditional expressions
You use the conditional operator to cause the entire conditional expression to evaluate on either second operand or the third operand. If a is `true`, then the expression will evaluate `b`, otherwise `c`.
```
a ? b : c
```
The return type of `b` and `c` should be compatible, meaning the same type in our language.

j. Operator Precedence

1. **Function calls or grouping**, array subscripting, and membership access operator expressions.
2. Unary operators, including logical negation, bitwise complement, unary positive, unary negative, indirection operator, type casting. When several unary operators are consecutive, the later ones are nested within the earlier ones: !-x means !(-x). **(right to left)**
3. Multiplication, division, and modular division expressions.
4. Addition and subtraction expressions.

5. Bitwise shifting expressions.
6. Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions.
7. Equal-to and not-equal-to expressions.
8. **Bitwise** AND expressions.
9. Bitwise exclusive OR expressions.
10. Bitwise inclusive OR expressions.
11. **Logical** AND expressions.
12. Logical OR expressions.
13. Conditional expressions (using ?:). When used as subexpressions, these are evaluated right to left.
14. All assignment expressions, including compound assignment. When multiple assignment statements appear as subexpressions in a single larger expression, they are evaluated right to left. **(right to left)**
15. Comma operator expressions.

We took the precedence here:
http://www.cs.bilkent.edu.tr/~guvenir/courses/CS101/op_precedence.html

4. Statements

a. Expression Statements
Similar to C, any expression with a semicolon appended is considered as a statement.

b. `if-else` statements
`if-else` statements are in the following forms.
`if (statement) { statement }`
or
`if (statement) { statement } else {statement}`
or
`if (statement) {statement} [elseif (statement){ statement }]+ [else {statement}]`

c. `while` statements
Format: `while (statement) { statement }`

d. `for` statements
`for` statements can take the form of
`for (statement; statement; statement) {statement}`
We also support C++11's for-each statement style, namely,
`for (element : list-like) {statement}`
to iterate over any list-like data structure.

e. Blocks
Similar to C, Harmonica uses braces to group zero or more statements. Blocks can be nested. Variables declared inside a block are local to that block.

f. `break, continue`
   `break` terminates a while or for control structure;
   `continue` terminates an iteration of the for or while loop and begins the next iteration.
g. `return`
   Used to end the execution of a function. It should be followed by a return value matching the declared return type (no return value is needed for a function returning `void`).
h. `typedef`
   `typedef` is used to create new names (aliases) for data types. The syntax is:
   ```
   typedef oldType newType
   ```

5. **Functions**
   You can write functions to separate parts of your program into distinct sub-procedures. Every program requires at least one function named `main`, which is where the program begins execution.
   a. Definition
      Functions are defined with the following syntax:
      ```
      returnType functionName (parameterList) {
        functionBody;
      }
      ```
      A parameter list is a comma-separated list of parameters, where each parameter consists of a parameter type followed by a parameter name.
   b. Calling a function
      Functions are called by their name and appropriate parameters. For example:
      ```
      foo (1, "bar");
      ```

   c. Lambda functions
      Functions can also be defined with the `lambda` keyword. However, lambda functions are restricted to a single line and meant to be used for quick, one-liner functions just like what python does. For example:
      ```
      list[int] onePlus = map(lambda (int a) int (a+1),
      [1,2,3]);
      ```

   d. First-class functions
      Functions are first-class members in Harmonica, which means that they can be assigned to variables and passed as parameters just like any other variable. Type declaration for functions compose of a sequence of types representing the types of function parameters followed by a single return type. However, we maintain a bottom line to prevent any abuse of higher than 2^31-1 orders of functions.

6. **Program Structure**
   A Harmonica source file consists of a list of statements and function definitions. Each source file defines a module that can be imported in other files. The compiler compiles several source files, with a single definition of the main function, into an executable program. monica is a language based on C and borrows ideas from Python / Go. It natively supports concurrency features and utilities commonly found in scripting languages.

7. **Lexical Elements**
   a. Identifiers
      Identifiers are arbitrary strings starting with [a-z A-Z] and followed by [a-z A-Z 0-9] (not underscores). We recommend camelCase.
   b. Key words
      `if, else, elseif, bool, int, float, string, list, struct, # (comment), return, parallel, channel, import` and various operators defined below.
   c. Constants
      A constant is a literal numeric or character value, such as 5 or 'm'
      i. Integer: decimal integers, [1-9]+[0-9]*
      ii. Float: real numbers following the C standard. Examples are 1.4, -0.3, 5e2, etc.
      iii. String: a string instance enclosed with double quotation marks.

8. **Data Types**
   a. Int
      Integer type. We only support 32-bit integers, ranging from -2^31 to 2^31-1.
   b. Float
      Floats are 8-byte double-precision floating point numbers.
   c. Boolean
      Boolean is just a boolean, true or false.
   d. Byte
      Byte represents 8-bits of raw data.
   e. String
      String is a sequence of ascii characters. Its length is limited by maximum integer value, which is 2^31-1.
   f. Tuple[]
      A tuple is a sequence of elements (of non-uniform types), with maximum length 2^31-1.
   g. List[]
      List is a sequence of elements of the same type, with maximum length 2^31-1.
   h. Struct
      Struct is a composite data structure supporting member variables and methods.
   i. Function Types

In Harmonica, function types are defined by the types of their input parameters and their return type. The notation for function types are simply a list of types separated by space, where the last type in the sequence is assumed to be the function's return type. For example, the type `Int Int List[Int]` is the type of a function that takes two integers and returns a list of integers.

9. **Concurrency Support**
   a. `Parallel`
   The `parallel` keyword spawns multiple child threads that execute the same function. It takes 2 required parameters: a function, and an iterable collection of elements. The function would be called on each of the elements in the collection in a separate thread. An optional 3rd argument can be specified to control the number of threads spawned from `parallel`.

```
# create 4 parallel thread to print out square.
void foo(int a)  { printi(a * a);
parallel(foo, [1,2,3,4]);
```

   b. `Mutex`
   We provide mutex for basic concurrency control. `mutex` is backed by POSIX C, pthread.h. It's been used avoid conflict when

```
mutex m;
list[int] arr;
int i = 0;

int f(int v) {
    mutex_lock(m);
    arr[i] = i;
    i += 1;
    mutex_unlock(m);
}

int main() {
   m = mutex_create();
   arr = malloc(16);
   parallel(f, [0,0,0,0], 4);
   mutex_destroy(m);

   for (i = 0; i < 4; i += 1) {
       printi(arr[i]);
   }
```

```
    free(arr);
}
```
The expected answer should be [1,2,3,4]

10. Template and preprocessor

a. `Alias`

`alias key1 key2`

`alias` is the keyword for preprocessing the template program. The order of using `alias` matters. For every alias directives, preprocessor will go through the whole program to replace all matching `key1` with `key2`.

```
alias U T
alias T int
```
In this case, the preprocessor first replace all U in the program into T, then traverse the program from the beginning again to replace T to int.

If the order is reversed,
```
alias T int
alias U T
```
The result should be totally different.

b. `Template<T>`

In the template system, capitalize the key word you want to do macro replacement later. The preprocessor will follow alias directives to do full context macro replacement. Then the type-specified copy of template program will be glued in the beginning of the client program. At last, the compiler will compile the client program.
```
alias T int

struct vector_T {
  list[T] elements;
  int length;
  int memsize;
};
```

After alias directive alias T int, the template will be generated as:
```
struct vector_int {
  list[int] elements;
  int length;
  int memsize;
};
```

How we test it

Overview

Building programs in OCaml is in general less error-prone given the type-checking system. In addition to the ocamlyacc shift-reduce detection, most of the bugs are those related to LLVM. During our development, we haven't encountered complex bugs other than the LLVM type mismatching, so the major tests we run is to check the target LLVM IR code matches the desired ones besides unit-tests on individual functions.

Test Framework

We employ test-driven development: for each feature we add, there are test cases serving as unit-tests. Then, for each feature that passes unit-tests, there is a regression test to run all previous test-cases to check it does not break existing features. For all scripts and test-cases, please see the source code at the end of this report (provided as a link).

ocamlyacc => OCaml Compiler => Unit Tests => Regression Tests => FeatureAdded

Experience and Lessons

The more difficult bugs are often related to cryptic LLVM error messages. As we debugged more of them we began to learn the inner workings of LLVM, its architecture and idiosyncrasies. This greatly helped in our debugging of further problems.

Tutorial

<u>Environment Setup</u>

What you will need: llvm, clang, ocaml, ocaml-llvm bindings, opam. The LLVM installed must be version 3.4, and the clang must be clang 3.4 since it generates LLVM 3.4 code. OCaml can be any version you like, but the ocaml-llvm bindings must be version 3.4 also.

Based on our experiences, it is recommended to build Harmonica on a Ubuntu trusty (14.04) build. It was difficult to run it on Mac since macOS comes with Apple's proprietary clang/llvm version.

<u>Make Harmonica</u>

Simply run 'make', but please make sure that you have correct opam settings. After this, you should get a 'harmonica.native' in the same directory. This is the LLVM-IR generator.

```
make
```

<u>Compile Your Program</u>

Run 'compile' to compile your source code files into LLVM IR. The compile script takes multiple files as arguments and compiles them into one LLVM IR file. If you are including libraries, make sure you get the order of source files right.

```
./compile <list of source files>
```

<u>Run Your Program</u>

After you get your LLVM IR code, direct them into the 'lli' from your llvm build. For more information on LLVM and its IR generator, please see LLVM's website.

Examples:
```
./harmonica.native < examples/statistician.ha | lli
./compile lib/vector.ha tests/test-vector1.ha | lli
```

Standard Libraries

We made a couple of libraries to demonstrate the power of our language. The math library that handles numeric calculations. The vector library utilizes our struct, memory management, and macro substitution features. The binary search tree library (and tests) makes use of our multithreading and concurrency support.

Math

The library contains the pow, exp, and log function that are able to calculate into 6-digit accuracy. With these three functions, it is theoretically possible to calculate or approximate all mathematical expressions.

Vector

The library implements dynamic array as an example on how to use the macro system, built-in malloc and free functions, and container types. The append operation on a vector triggers a memory allocation if this element goes beyond the available memory of the vector. The new memory allocation doubles the size of the previously-allocated memory, and all contents of the array are copied into the new memory block.

Binary Search Tree

The binary search tree library implements a simple thread-safe binary search tree that holds integers. Thread safety is achieved through fine-grained locking, where each operation only locks the node being operated on and its parent. Tests for this library uses the parallel function to spawn multiple threads inserting pseudo-random numbers into the tree, and then prints them out through an in-order traversal. The resulting list should be sorted.

Code Listing

The source code / commit log for this project is available at https://github.com/anachromeJ/harmonica (and hopefully will be continually updated). We also include the full source code listing here as required.

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

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

.PHONY : harmonica.native

harmonica.native : bindings.bc scanner.mll parser.mly ast.ml
semant.ml
	ocamlbuild -use-ocamlfind -pkgs
llvm,llvm.analysis,llvm.bitreader,llvm.linker -cflags -w,+a-4,-annot
\
		harmonica.native

bindings.bc : bindings.c
	clang -c -pthread -emit-llvm bindings.c

# "make clean" removes all generated files

.PHONY : clean
clean :
	ocamlbuild -clean
	rm -rf testall.log *.diff harmonica scanner.ml parser.ml
parser.mli bindings.bc
	rm -rf *.err *.ll
	rm -rf *.cmx *.cmi *.cmo *.cmx *.o
	rm -rf *~ \#* *.out

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

OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx
harmonica.cmx
```

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

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 $<

parser: parser.ml scanner.ml
	ocamlc -c ast.ml
	ocamlc -c parser.mli
	ocamlc -c scanner.ml
	ocamlc -c parser.ml
	ocamlc -c main.ml
	ocamlc -o main ast.cmo scanner.cmo parser.cmo main.cmo

### Generated by "ocamldep *.ml *.mli" after building scanner.ml and
parser.ml
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
harmonica.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
harmonica.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 if1 if2 if3 if4 if5 local1 local2 ops1 ops2 var1 var2
\
    while1 while2

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

TARFILES = ast.ml codegen.ml Makefile harmonica.ml parser.mly README
scanner.mll \
      semant.ml testall.sh $(TESTFILES:%=tests/%)

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

Ast.ml

```
(* 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

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

type typ =
    DataType of primitive
  | Tuple of typ list
  | List of typ
  | Channel of typ
  | Struct of string * bind list
  | UserType of string
  | FuncType of typ list
  | Any

and bind = typ * string

type id =
    NaiveId of string
  | MemberId of id * string
  | IndexId of id * expr

and expr =
    IntLit of int
  | BoolLit of bool
  | StringLit of string
  | FloatLit of float
  | TupleLit of expr list
  | ListLit of expr list
  | Id of id
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of id * expr
  | Call of id * expr list
  | Lambda of typ * bind list * expr
  | Null
  | Noexpr
```

```
type var_decl =
    Bind of typ * string
  | Binass of typ * string * expr

type global_stmt =
    Typedef of typ * string
  | Global of var_decl

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

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

type program = global_stmt list * func_decl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "Add"
  | Sub -> "Sub"
  | Mult -> "Mult"
  | Div -> "Div"
  | Equal -> "Equal"
  | Neq -> "Neq"
  | Less -> "Less"
  | Leq -> "Leq"
  | Greater -> "Greater"
  | Geq -> "Geq"
  | And -> "And"
  | Or -> "Or"
```

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

let map_and_concat f list = String.concat ", " (List.map f list)

let rec string_of_typ = function
    DataType(Int) -> "Int"
  | DataType(Bool) -> "Bool"
  | DataType(Void) -> "Void"
  | DataType(Float) -> "Float"
  | DataType(String) -> "String"
  | Any -> "Any"
  | Tuple(tlist) -> "Tuple(" ^ string_of_tlist tlist ^ ")"
  | List(t) -> "List(" ^ string_of_typ t ^ ")"
  | Struct(id, blist) -> "Struct(" ^ id ^ ", " ^ string_of_blist
blist ^ ")"
  | Channel(t) -> "Channel(" ^ string_of_typ t ^ ")"
  | UserType(id) -> "UserType(" ^ id ^ ")"
  | FuncType(tlist) -> "FuncType(" ^ string_of_tlist tlist ^ ")"

and string_of_bind (t, id) = "Bind(" ^ string_of_typ t ^ ", " ^ id ^
")"
and string_of_tlist tlist = map_and_concat string_of_typ  tlist
and string_of_blist blist = map_and_concat string_of_bind blist

let rec string_of_id = function
    NaiveId(s) -> s
  | MemberId(id, s) -> "Member(" ^ string_of_id id ^ ", " ^ s ^ ")"
  | IndexId(id, e) -> "Index(" ^ string_of_id id ^ ", " ^
string_of_expr e ^ ")"

and string_of_expr = function
    IntLit(l) -> "IntLit(" ^ string_of_int l ^ ")"
  | BoolLit(true) -> "BoolLit(True)"
  | BoolLit(false) -> "BoolLit(False)"
  | StringLit(s) -> "StringLit(\"" ^ s ^ "\")"
  | FloatLit(f) -> "FloatLit(" ^ string_of_float f ^ ")"
  | TupleLit(elist) -> "TupleLit(" ^ String.concat ", " (List.map
string_of_expr elist) ^ ")"
  | ListLit(elist) -> "ListLit(" ^ String.concat ", " (List.map
string_of_expr elist) ^ ")"
```

```
  | Null -> "NULL"
  | Id(s) -> "Id(" ^ string_of_id s ^ ")"
  | Binop(e1, o, e2) ->
      "Binop(" ^ string_of_expr e1 ^ ", " ^ string_of_op o ^ ", " ^
string_of_expr e2 ^ ")"
  | Unop(o, e) -> "Unop(" ^ string_of_uop o ^ ", " ^ string_of_expr e
^ ")"
  | Assign(v, e) -> "Assign(" ^ string_of_id v ^ ", " ^
string_of_expr e ^ ")"
  | Call(f, el) ->
     "Call(" ^ string_of_id f ^ ", " ^ String.concat ", " (List.map
string_of_expr el) ^ ")"
  | Lambda(rt, blist, e) ->
     "Lambda(" ^ string_of_typ rt ^ ", " ^ string_of_blist blist ^ ",
" ^ string_of_expr e ^ ")"
  | Noexpr -> "Noexpr"


let string_of_vdecl = function
    Bind(t, s) -> "Bind(" ^ string_of_typ t ^ ", " ^ s ^ ")\n"
  | Binass(t, s, e) -> "Binass(" ^ string_of_typ t ^ ", " ^ s ^ ", "
^ string_of_expr e ^ ")\n"

let string_of_global_stmt = function
    Typedef(t, s) -> "Typedef(" ^ string_of_typ t ^ ", " ^ s ^ ")\n"
  | Global(vd) -> string_of_vdecl vd

let rec string_of_stmt = function
    Block(stmts) ->
      "Block(\n" ^ String.concat "" (List.map string_of_stmt stmts) ^
")\n"
  | Expr(expr) -> "Expr(" ^ string_of_expr expr ^ ")\n";
  | Return(expr) -> "Return(" ^ string_of_expr expr ^ ")\n";
  | If(e, s, Block([])) -> "If(" ^ string_of_expr e ^ ", " ^
string_of_stmt s ^ ")\n"
  | If(e, s1, s2) ->  "If(" ^ string_of_expr e ^ ", " ^
      string_of_stmt s1 ^ ", " ^ string_of_stmt s2 ^ ")\n"
  | For(e1, e2, e3, s) ->
      "For(" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; "
^
      string_of_expr e3  ^ ", " ^ string_of_stmt s ^ ")\n"
  | While(e, s) -> "While(" ^ string_of_expr e ^ ", " ^
string_of_stmt s ^ ")\n"
```

```
  | Local(vd) -> string_of_vdecl vd

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

let string_of_program (gstmts, funcs) =
  String.concat "" (List.map string_of_global_stmt gstmts) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

scanner.ml

```
(* Ocamllex scanner for Harmonica *)

{
     open Parser

}

let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let str = (ascii | escape)*

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }  (* Whitespace *)
| "#"      { comment lexbuf }             (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '\"'     { QUOTE }
| '\''     { QUOTE }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "+="     { CPLUS }
| "-="     { CMINUS }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "."      { DOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
(* type keywords *)
| "int"    { INT }
| "float"  { FLOAT }
| "bool"   { BOOL }
| "string" { STRING }
| "void"   { VOID }
| "tuple"  { TUPLE }
| "list"   { LIST }
| "struct" { STRUCT_STMT }
| "true"   { TRUE }
| "false"  { FALSE }
| "typedef" { TYPEDEF }
| "channel" { CHANNEL }
| "chan"    { CHAN }
| "parallel" { PARALLEL }
| "lambda"  { LAMBDA }
| "NULL"   { NULL }

(* string literal *)
| '"' (([' '-'!' '#'-'&' '('-'[' ']'-'~'] | '\\' [ '\\' '"' 'n' 'r'
't' '''])* as lxm) '"'  { STRING_LITERAL(lxm) }
(* int literal *)
| ['0'-'9']+ as lxm { INT_LITERAL(int_of_string lxm) }
(* float literal *)
| ['-''+']?['0'-'9']*('.')['0'-'9']+(['e''E']['-''+']?['0'-'9']+)? as
lxm { FLOAT_LITERAL(float_of_string lxm) }
(* ID *)
| ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm {
ID(lxm) }

| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped
char)) }

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

(*
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")) }*)
```

parser.ml

```
/* Ocamlyacc parser for Harmonica */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA QUOTE
DOT
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token CPLUS CMINUS
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE INT FLOAT BOOL STRING VOID TUPLE LIST
STRUCT_STMT TYPEDEF
%token CHANNEL PARALLEL CHAN LAMBDA
%token NULL
%token <int> INT_LITERAL
%token <float> FLOAT_LITERAL
%token <string> ID STRING_LITERAL
%token EOF

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

%start program
%type <Ast.program> program

%%

program:
  decls EOF { List.rev (fst $1), List.rev (snd $1) }

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

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

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

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

tuple_typ_list:
    /* nothing */ { [] }
  | tuple_typ_list typ { $2 :: $1 }

func_typ_list:
    typ { [$1] }
  | func_typ_list typ { $2 :: $1 }

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

typ:
    primitive  { DataType($1) }
  | TUPLE LBRACKET tuple_typ_list RBRACKET { Tuple(List.rev $3) }
  | LIST LBRACKET typ RBRACKET { List($3) }
  | CHANNEL LBRACKET typ RBRACKET { Channel($3) }
  | ID { UserType($1) }
  | LT func_typ_list GT { FuncType((List.hd $2) :: List.rev (List.tl
$2)) }
```

```
vdecl:
    typ ID SEMI { Bind($1, $2) }
  | typ ID ASSIGN expr SEMI { Binass($1, $2, $4) }

global:
    STRUCT_STMT ID LBRACE bind_list RBRACE SEMI { Typedef(Struct($2,
List.rev $4), $2) }
  | TYPEDEF typ ID SEMI { Typedef($2, $3) }
  | vdecl { Global($1) }

bind_list:
    /* nothing */ { [] }
  | bind_list bind { $2 :: $1 }

bind:
    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_opt SEMI expr_opt RPAREN stmt
     { if $5 = Noexpr
       then For($3, BoolLit(true), $7, $9)
       else For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | vdecl { Local($1) }

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

expr_list_opt:
    /* nothing */ { [] }
  | expr_list { List.rev $1 }
```

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

expr_comma_list_opt:
    /* nothing */ { [] }
  | expr COMMA { [$1] }
  | expr_comma_list { List.rev $1 }

expr_comma_list:
    expr COMMA expr { $3 :: [$1] }
  | expr_comma_list COMMA expr { $3 :: $1 }

id_expr:
    ID                  { NaiveId($1) }
  | id_expr  DOT  ID    { MemberId($1, $3) }
  | id_expr  LBRACKET expr RBRACKET { IndexId($1, $3) }

expr:
    literals          { $1 }
  | NULL              { Null }
  | id_expr               { Id($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | id_expr CPLUS expr  { Assign($1, Binop(Id($1), Add, $3))}
  | id_expr CMINUS expr { Assign($1, Binop(Id($1), Sub, $3))}
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr         { Unop(Not, $2) }
  | id_expr ASSIGN expr   { Assign($1, $3) }
  | id_expr LPAREN actuals_opt RPAREN { Call($1, $3) }
  | PARALLEL LPAREN actuals_opt RPAREN { Call(NaiveId("parallel"),
$3) }
  | CHAN LPAREN chan_actuals_opt RPAREN { Call(NaiveId("chan"), $3) }
  | LAMBDA LPAREN formals_opt RPAREN typ LPAREN expr RPAREN {
Lambda($5, $3, $7) }
  | LPAREN expr RPAREN { $2 }

primitives:
    INT_LITERAL      { IntLit($1) }
  | TRUE             { BoolLit(true) }
  | FALSE            { BoolLit(false) }
  | FLOAT_LITERAL    { FloatLit($1) }
  | STRING_LITERAL   { StringLit($1) }

literals:
    primitives         { $1 }
  | LPAREN expr_comma_list_opt RPAREN { TupleLit($2) }
  | LBRACKET expr_list_opt RBRACKET { ListLit($2) }

chan_actuals_opt:
    typ { [StringLit(string_of_typ $1)] }
  | typ COMMA expr { StringLit(string_of_typ $1) :: [$3] }

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

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

semant.ml

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

open Ast

module StringMap = Map.Make(String)

type environment = {
    externals: typ StringMap.t;
    locals: typ StringMap.t;
  }

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

let rec resolve_user_type usert utypes =
  (match usert with
     UserType(s) -> (try resolve_user_type (StringMap.find s utypes)
utypes
                    with Not_found -> raise (Failure ("undefined
type " ^ s)))
   | _ -> usert)

let mutex_t = Struct ("mutex", [])

let check (global_stmts, functions) =

  (* User-defined types *)

  let user_types = List.fold_left
                     (fun map -> function
                         Typedef(t, s) -> StringMap.add s t map
                       | _ -> map)
                     (StringMap.add "mutex" mutex_t StringMap.empty)
                    global_stmts
  in

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

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


  (* Structural type equality *)
  let rec typ_equal t1 t2 =
    let t1 = resolve_user_type t1 user_types in
    let t2 = resolve_user_type t2 user_types in
    (match (t1, t2) with
        (Any, _) -> true
      | (_, Any) -> true
      | (DataType(p1), DataType(p2)) -> p1 = p2
      | (Tuple(tlist1), Tuple(tlist2)) ->
          List.for_all2 typ_equal tlist1 tlist2
      | (List(t1'), List(t2')) -> typ_equal t1' t2'
      | (Channel(t1'), Channel(t2')) -> typ_equal t1' t2'
      | (Struct(name1, _), Struct(name2, _)) -> name1 = name2 (* TODO:
ok? *)
      | (UserType(_), UserType(_)) ->
          typ_equal (resolve_user_type t1 user_types)
(resolve_user_type t2 user_types)
      | (FuncType(tlist1), FuncType(tlist2)) ->
          List.for_all2 typ_equal tlist1 tlist2
      | _ -> false
    ) in

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

  (**** Checking Functions Definitions ****)
  let builtins = ["print"; "printb"; "printf"; "printi"; "printendl";
"concat"; "parallel"; "free"; "malloc"; "mutex_create";
"mutex_destroy"; "mutex_lock"; "mutex_unlock" ; "sizeof" ] in
```

```
  let builtin_duplicates = List.filter (fun fname -> List.mem fname
builtins)
                                       (List.map (fun fd -> fd.fname)
functions) in
  if List.length builtin_duplicates > 0
  then raise (Failure ("function " ^
                         (String.concat ", " builtin_duplicates) ^
                           " may not be defined")) else ();

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

  (* Global variable table *)
  let builtins = List.fold_left
                   (fun map (name, t) -> StringMap.add name t map)
                   StringMap.empty
                   [("print", FuncType([DataType(Void);
DataType(String)]));
                    ("printb", FuncType([DataType(Void);
DataType(Bool)]));
                    ("printf", FuncType([DataType(Void);
DataType(Float)]));
                    ("printi", FuncType([DataType(Void);
DataType(Int)]));
                    ("printendl", FuncType([DataType(Void);
DataType(String)]));
                    ("concat", FuncType([DataType(String);
DataType(String); DataType(String)]));
                    ("parallel", FuncType([DataType(Int);
FuncType([Any; Any]); List(Any); DataType(Int)]));
                    ("free", FuncType([DataType(Void); Any]));
                    ("malloc", FuncType([List(Any); DataType(Int)]));
                    ("mutex_create", FuncType([mutex_t]));
                    ("mutex_lock", FuncType([DataType(Int);
mutex_t]));
                    ("mutex_unlock", FuncType([DataType(Int);
mutex_t]));
                    ("mutex_destroy", FuncType([DataType(Int);
mutex_t]));
                    ("sizeof",  FuncType([DataType(Int); Any ]))
                   ]
  in
  let get_functype fdecl = FuncType(fdecl.typ :: (List.map fst
fdecl.formals)) in
  let global_funcs = List.fold_left
                     (fun map fd -> StringMap.add fd.fname
                                                  (get_functype fd)
                                                  map)
                     builtins
                     functions
  in

  (* Ensure "main" is defined *)
  ignore (try List.find (fun f -> f.fname = "main") functions
          with Not_found -> raise (Failure ("main function
undefined")));

  (*** Checking Global Variables ***)
  let add_bind env t name =
    check_not_void (fun n -> "illegal void variable " ^ n) (t, name);
    if StringMap.mem name env.locals then
      raise (Failure ("redefinition of " ^ name))
    else {
        externals = env.externals;
        locals = StringMap.add name t env.locals
      }
  in

  (* NOTE: inner-scope variable overrides outer-scope variable with
same name *)
  let rec type_of_id env = function
      NaiveId(s) ->
      let t =
        (try StringMap.find s (env.locals)
         with Not_found -> (
           try StringMap.find s (env.externals)
           with Not_found -> raise (Failure ("undeclared identifier "
^ s))))
      in
      resolve_user_type t user_types
    | MemberId(id, n) ->
       let container_type = resolve_user_type (type_of_id env id)
user_types in
       (match container_type with
          Struct(_, blist) ->
          let (t, _) = List.find (fun (_, n') -> n' = n) blist in
          resolve_user_type t user_types
        | _ -> raise (Failure (string_of_id id  ^ " is not a struct
type")))
    | IndexId(id, e) ->
       let container_type = resolve_user_type (type_of_id env id)
user_types in
       (match container_type with
          List(t) -> (match expr env e with DataType(Int) ->
                                          resolve_user_type t
user_types
                                        | _ -> raise (Failure "WTF.
Must be int."))
        | _ -> raise (Failure "WTF. Must be list.")
       )

  (* Return the type of an expression or throw an exception *)
  and expr env = function
      IntLit _ -> DataType(Int)
    | BoolLit _ -> DataType(Bool)
    | StringLit _ -> DataType(String)
    | FloatLit _ -> DataType(Float)
    | TupleLit elist -> Tuple (List.map (expr env) elist)
    | ListLit elist as e ->
       let tlist = List.map (expr env) elist in
       if (List.length tlist) = 0
       then List(Any)
       else
         let canon = List.hd tlist in
         if List.for_all (fun t -> t = canon) tlist
         then List(canon)
         else raise (Failure ("inconsistent types in list literal "
                              ^ string_of_expr e))
    | Null -> Any
    | Id s -> type_of_id env s
    | Binop(e1, op, e2) as e ->
       let t1 = expr env e1 and t2 = expr env e2 in
       (match op with
          Add | Sub | Mult | Div  ->
                 (match t1 with
                    DataType(Float) ->  DataType(Float)
                  | _ ->
                     (match t2 with
                      DataType(Float) -> DataType(Float)
                     | _ -> DataType(Int)
                    )
                )
          | Equal | Neq when typ_equal t1 t2 -> DataType(Bool)
          | Less | Leq | Greater | Geq
               when (t1 = t2) && (t1 = DataType(Int) || t1 =
DataType(Float))
            -> DataType(Bool)
          | And | Or when t1 = DataType(Bool) && t2 = DataType(Bool)
-> DataType(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 env e in
       (match op with

          Neg ->
          (match t with
             DataType(Float) -> DataType(Float)
           | DataType(Int) -> DataType(Int)
           |  _ -> raise (Failure ("illegal unary operator " ^
string_of_uop op ^
                                    string_of_typ t ^ " in " ^
string_of_expr ex))
          )
         | Not when t = DataType(Bool) -> DataType(Bool)
         | _ -> raise (Failure ("illegal unary operator " ^
string_of_uop op ^
                                string_of_typ t ^ " in " ^
string_of_expr ex)))

    | Noexpr -> DataType(Void)
    | Assign(var, e) as ex -> let lt = type_of_id env var
                              and rt = expr env e in
                              check_assign lt rt (Failure ("illegal
assignment " ^ string_of_typ lt ^
                                                              " = " ^
string_of_typ rt ^ " in " ^
```

```
string_of_expr ex))
    | Call(fname, actuals) as call ->
       let ftype = type_of_id env fname in
       (match ftype with
          FuncType(tlist) ->
          let get_tp tp = resolve_user_type tp user_types in
          let resolved_formals = List.map get_tp (List.tl tlist) in
          let formals = resolved_formals in
          let ret = List.hd tlist in
          if List.length actuals != List.length formals then
            raise (Failure ("expecting " ^
                              string_of_int (List.length formals) ^
                                " arguments in " ^ string_of_expr
call))
          else
            List.iter2
              (fun ft e ->
                let et = expr env e in
                ignore (check_assign ft et (Failure ("illegal actual
argument in call to " ^ string_of_id fname ^ ": found " ^
string_of_typ et ^ ", expected " ^ string_of_typ ft ^ " in " ^
string_of_expr e))))
              formals actuals;
          ret
        | _ -> raise (Failure (string_of_id fname ^ " is not a
function"))
       )
    | Lambda(rt, blist, e) ->
       let new_env = {
           externals = StringMap.merge (fun _ xo yo -> match xo,yo
with
                                                       | Some x, Some
_ -> Some x
                                                       | None, yo ->
yo
                                                       | xo, None ->
xo ) env.locals env.externals;
           locals = StringMap.empty
         } in
       let fenv = List.fold_left
                    (fun env' (t, name) -> add_bind env' t name)
new_env blist in
```

```
       let rt = resolve_user_type rt user_types in
       let et = expr fenv e in
       if typ_equal rt et
       then FuncType(rt :: (List.map fst blist))
       else raise (Failure ("expression does not match lambda return
type: found " ^
                              string_of_typ et ^ ", expected " ^
string_of_typ rt))
  in

  let add_vdecl env = function
      Bind(t, name) -> add_bind env t name
    | Binass(t, name, e) ->
       let rtype = expr env e in
       ignore (check_assign t rtype
                            (Failure ("illegal assignment " ^
string_of_typ t ^
                                        " = " ^ string_of_typ rtype ^
" in " ^
                                          string_of_expr e)));
       add_bind env t name
  in

  let global_env = List.fold_left
                     (fun env -> function
                         Global(vd) -> add_vdecl env vd
                       | _ -> env)
                     { externals = StringMap.empty;
                       locals = global_funcs }
                     global_stmts
  in

  let global_env = { externals = global_env.locals;
                     locals = StringMap.empty } in

  (*** Checking Function Contents ***)
  let check_function fenv 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);

    (* Local variables and formals *)
    let fenv = List.fold_left
                (fun e (t, name) -> add_bind e t name) fenv
func.formals in

    let check_bool_expr env e =
      let t = expr env e in
      if typ_equal t (DataType Bool)
      then ()
      else raise (Failure ("expected boolean expression in " ^
string_of_expr e))
    in

    (* Verify a statement or throw an exception, returns updated
environment *)
    let rec stmt senv = function
        Block sl ->
        let rec check_block benv = function
            [Return _ as s] -> stmt benv s
          | Return _ :: _ -> raise (Failure "nothing may follow a
return")
          | s :: ss -> check_block (stmt benv s) ss
          | [] -> benv
        in
        (* New symbol table for new scope *)
        let benv = {
            locals = StringMap.empty;
            externals =
              StringMap.merge (fun _ xo yo -> match xo,yo with
                                                | Some x, Some _ ->
Some x
                                                | None, yo -> yo
                                                | xo, None -> xo )
senv.locals senv.externals } in
        ignore (check_block benv sl);
        senv
      | Expr e -> ignore (expr senv e); senv
      | Return e -> let t = expr senv e in
                    let expected = resolve_user_type func.typ
user_types in
                    if typ_equal t expected then senv else
```

```
                      raise (Failure ("return gives " ^ string_of_typ
t ^ " expected " ^
                                            string_of_typ expected ^ " in
" ^ string_of_expr e))
      | If(p, b1, b2) -> check_bool_expr senv p;
                         ignore (stmt senv b1);
                         ignore (stmt senv b2);
                         senv
      | For(e1, e2, e3, st) ->
         ignore (expr senv e1);
         check_bool_expr senv e2;
         ignore (expr senv e3); ignore (stmt senv st);
         senv
      | While(p, s) -> check_bool_expr senv p; ignore (stmt senv s);
senv
      | Local(vd) -> add_vdecl senv vd
    in
    ignore (stmt fenv (Block func.body))

  in

  List.iter (check_function global_env) functions
```

codegen.ml

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

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

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

Detailed documentation on the OCaml LLVM library:

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

 *)

module L = Llvm
module A = Ast
module S = Semant

module StringMap = Map.Make(String)

type environment = {
    externals: L.llvalue StringMap.t;
    locals: L.llvalue StringMap.t;
    builder: L.llbuilder;
  }

let debug msg =
  if false
  then prerr_endline msg
  else ()

let translate (global_stmts, functions) =
  let context = L.global_context () in
  let the_module = L.create_module context "Harmonica"
  and i32_t  = L.i32_type    context
  and i8_t   = L.i8_type     context
  and i1_t   = L.i1_type     context
  and dbl_t  = L.double_type context
  and void_t = L.void_type   context in
  let string_t = L.pointer_type i8_t in
  let voidstar_t = string_t in
  (* user-defined types *)
  let user_types = List.fold_left
                     (fun map -> function
                         A.Typedef(t, s) -> StringMap.add s t map
                       | _ -> map)
                     (StringMap.add "mutex" S.mutex_t
StringMap.empty)
                     global_stmts
  in

  (* map of struct names to their fields *)
  let struct_map = List.fold_left
                     (fun map -> function
                         A.Typedef(A.Struct(name, bind_list), _) ->
                         StringMap.add name bind_list map
                       | _ -> map)
                     StringMap.empty
                     global_stmts
  in

  let typ_cache = Hashtbl.create 10 in

  let rec ltype_of_typ = function
      A.DataType(A.Int) -> i32_t
    | A.DataType(A.Bool) -> i1_t
    | A.DataType(A.Float) -> dbl_t
    | A.DataType(A.Void) -> void_t
    | A.DataType(A.String) -> string_t
    | A.Tuple(tlist) -> L.struct_type context (Array.of_list
(List.map ltype_of_typ tlist))
    (* TODO: implement dynamic arrays *)
    | A.List(t) -> L.pointer_type (ltype_of_typ t)
    (* TODO: channels *)
    | A.Struct(name, blist) ->
       let t =
         (try Hashtbl.find typ_cache name
          with Not_found ->
            let struct_t = L.named_struct_type context name in
            Hashtbl.add typ_cache name struct_t;
            L.struct_set_body struct_t (Array.of_list (List.map
ltype_of_typ (List.map fst blist))) false;
            struct_t) in
       L.pointer_type t
    | A.UserType(_) as t -> let t' = S.resolve_user_type t user_types
in
                            ltype_of_typ t'
    | A.FuncType(tlist) ->
       let ftype =
         let llist = List.map ltype_of_typ tlist in
         let return_t = List.hd llist in
         let params_t = Array.of_list (List.tl llist) in
         L.function_type return_t params_t
       in
       L.pointer_type ftype
    | _ -> raise (Failure "Not yet implemented")
  in

  (* let rec typ_of_ltype ltype = match (L.classify_type ltype) with
*)
  (*     L.TypeKind.Integer -> if ltype = i1_t *)
  (*                           then A.DataType(A.Bool)  *)
  (*                           else A.DataType(A.Int) *)
  (*   | L.TypeKind.Double -> A.DataType(A.Float) *)
  (*   | L.TypeKind.Void -> A.DataType(A.Void) *)
  (*   | L.TypeKind.Pointer -> A.DataType(A.String) *)
  (*   | L.TypeKind.Array -> A.List (typ_of_ltype (L.element_type
ltype)) *)
  (*   | L.TypeKind.Struct ->  *)
  (*      (match (L.struct_name ltype) with *)
  (*         None -> A.Tuple (List.map typ_of_ltype (Array.to_list
(L.struct_element_types ltype))) *)
  (*       | Some(name) -> A.Struct (name, StringMap.find name
struct_map)) *)
  (*   | L.TypeKind.Function -> *)
  (*      let return_t = L.return_type ltype in *)
  (*      let param_ts = Array.to_list (L.param_types ltype) in *)
  (*      A.FuncType (List.map typ_of_ltype (List.rev (return_t ::
(List.rev param_ts)))) *)
  (*   | _ -> raise (Failure "Unsupported llvm type") *)
  (* in *)

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

```
  let str_concat_t = L.function_type string_t [| string_t; string_t|]
in
  let str_concat_func = L.declare_function "str_concat" str_concat_t
the_module in

  let subroutine_t = L.pointer_type (L.function_type voidstar_t [|
voidstar_t |] ) in
  let parallel_t = L.function_type i32_t [| subroutine_t;
L.pointer_type voidstar_t; i32_t; i32_t |] in
  let parallel_func = L.declare_function "parallel" parallel_t
the_module in

  let mutex_t = voidstar_t in
  let mutex_create_t = L.function_type mutex_t [||] in
  let mutex_create_func = L.declare_function "mutex_create"
mutex_create_t the_module in

  let mutex_lock_t = L.function_type i32_t [| mutex_t |] in
  let mutex_lock_func = L.declare_function "lock" mutex_lock_t
the_module in

  let mutex_unlock_t = L.function_type i32_t [| mutex_t |] in
  let mutex_unlock_func = L.declare_function "unlock" mutex_unlock_t
the_module in

  let mutex_destroy_t = L.function_type i32_t [| mutex_t |] in
  let mutex_destroy_func = L.declare_function "destroy"
mutex_destroy_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 in
      let 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
      let fval  = L.define_function name ftype the_module in
      StringMap.add name (fval, fdecl) m in
    List.fold_left function_decl StringMap.empty functions in

  let (main_function, _) = StringMap.find "main" function_decls in
  let global_builder = L.builder_at_end context (L.entry_block
main_function) in

  let int_format_str = L.build_global_stringptr "%d\n" "fmt"
global_builder in
  let float_format_str = L.build_global_stringptr "%f\n" "fmt"
global_builder in
  let printendl_str = L.build_global_stringptr "%s\n" "fmt"
global_builder in
  let true_str = L.build_global_stringptr "true\n" "true_str"
global_builder in
  let false_str = L.build_global_stringptr "false\n" "false_str"
global_builder in

  let llstore lval laddr builder =
    let ptr = L.build_pointercast laddr (L.pointer_type (L.type_of
lval)) "" builder in
    let store_inst = (L.build_store lval ptr builder) in
    debug ((L.string_of_llvalue store_inst));
    ()
  in

  (* Return the address of a variable or formal argument *)
  let rec lookup env x =
    begin match x with
      A.NaiveId(n) ->
      let result =
        (try StringMap.find n env.locals
         with Not_found ->
           (try StringMap.find n env.externals
            with Not_found -> raise (Failure ("undeclared variable "
^ n))))
      in
      (* debug ("lookup: " ^ n ^ " = " ^ (L.string_of_llvalue
result)); *)
      result
    | A.MemberId(id, field_name) ->
       let rec find_index_of field_name list =
         match list with
           [] -> raise (Failure ("undeclared field " ^ field_name))
```

```
          | hd :: tl -> if field_name = (snd hd)
                        then 0
                        else 1 + find_index_of field_name tl
       in
       let container_pp = lookup env id in
       let container_addr = L.build_load container_pp "" env.builder
in
       let container = L.build_load container_addr "" env.builder in
       debug ("container: " ^ L.string_of_llvalue container);
       let container_tname_opt = L.struct_name (L.type_of container)
in
       (match container_tname_opt with
          None -> raise (Failure ("expected struct, found tuple: " ^
A.string_of_id id))
        | Some(container_tname) ->
           let fields = StringMap.find container_tname struct_map in
           let idx = find_index_of field_name fields in
           let addr = L.build_struct_gep container_addr idx ""
env.builder in
           addr)
    | A.IndexId(id, e) ->
       let array_pp = lookup env id in (* ptr to ptr to array *)
       let array_ptr = L.build_load array_pp "" env.builder in
       let index = snd (expr env e) in
       debug ("index = " ^ (L.string_of_llvalue index));
       let eaddr = L.build_gep array_ptr [|index|] "" env.builder in
       debug ("eaddr = " ^ (L.string_of_llvalue eaddr));
       (* let etype = L.element_type (L.type_of array_ptr) in *)
       (* let ptr = L.build_pointercast eaddr (L.pointer_type etype)
"" env.builder in *)
       eaddr
    end

  (* Construct code for an expression; return its value *)
  and expr env =
    let evaluate_exprs env exprs =
      let (env', relements) = List.fold_left
                                (fun (env, values) e ->
                                  let (env', v) = expr env e in
                                  (env', v :: values))
                                (env, [])
                                exprs in
      (env', List.rev relements)
    in
    function
      A.IntLit i -> (env, L.const_int i32_t i)
    | A.BoolLit b ->
       let llb = L.const_int i1_t (if b then 1 else 0) in
       debug (L.string_of_llvalue llb);
       (env, llb)
    | A.StringLit s -> (env, L.build_global_stringptr s ""
env.builder)
    | A.FloatLit f -> (env, L.const_float dbl_t f)
    | A.TupleLit _ -> raise (Failure "tuples are currently not
supported")
    (* let (env, elements) = evaluate_exprs env elist in *)
    (* (env, L.const_struct context (Array.of_list elements)) *)
    | A.ListLit elist ->
       if List.length elist == 0
       then raise (Failure "Empty lists are not supported")
       else
         let (env, elements) = evaluate_exprs env elist in
         debug ("elements = " ^ String.concat ", " (List.map
L.string_of_llvalue elements));
         let etype = L.type_of (List.hd elements) in
         debug ("etype = " ^ L.string_of_lltype etype);
         (* let array = L.const_array etype (Array.of_list elements)
in *)
         (* debug ("array = " ^ L.string_of_llvalue array); *)
         let num_elems = List.length elist in
         let ptr = L.build_array_malloc
                     etype
                     (L.const_int i32_t num_elems)
                     ""
                     env.builder in
         (* debug ("ptr = " ^ L.string_of_llvalue ptr); *)
         (* let eptr = L.build_pointercast ptr  *)
         (*                              (L.pointer_type etype)  *)
         (*                              "" *)
         (*                              env.builder in *)
         ignore (List.fold_left
                    (fun i elem ->
                     let ind = L.const_int i32_t i in
                     let eptr = L.build_gep ptr [|ind|] ""
env.builder in
                     llstore elem eptr env.builder;
                    i+1
                  ) 0 elements);
         (env, ptr)
    | A.Null -> (env, L.const_null voidstar_t)
    | A.Noexpr -> (env, L.const_int i32_t 0)
    | A.Id id ->
        (env, L.build_load (lookup env id) "" env.builder)
    | A.Binop (e1, op, e2) ->
        let (env, e1') = expr env e1 in
        let (env, e2') = expr env e2 in
        let e2' = if e2 = A.Null
                  then L.const_null (L.type_of e1')
                  else e2' in
        let exp_type = L.classify_type(L.type_of e1') in
        let exp_type2 = L.classify_type(L.type_of e2') in
        (match exp_type with
           L.TypeKind.Double ->
           let e2_ = match exp_type2 with
               L.TypeKind.Double -> e2'
             | L.TypeKind.Integer -> (L.build_sitofp e2' dbl_t ""
env.builder)
             | _ -> raise (Failure "Algebra only supports float and
int.") in
            (env,
             (match op 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.Ult
              | 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" env.builder)
         | _ ->
             (match exp_type2 with
                L.TypeKind.Double ->
                let e1_ = match exp_type with
                    L.TypeKind.Double -> e1'
                  | L.TypeKind.Integer -> (L.build_sitofp e1' dbl_t ""
  env.builder)
                  | _ -> raise (Failure "Algebra only supports float
  and int.") in
                (env,
                 (match op 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.Ult
                  | 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" env.builder)
              | _ ->
                 (env,
                  (match op with
                     A.Add     -> L.build_add
                   | A.Sub     -> L.build_sub
                   | A.Mult    -> L.build_mul
                   | A.Div     -> L.build_sdiv
                   | A.And     -> L.build_and
                   | A.Or      -> L.build_or
                   | A.Equal   -> L.build_icmp L.Icmp.Eq
                   | A.Neq     -> L.build_icmp L.Icmp.Ne
                   | A.Less    -> L.build_icmp L.Icmp.Slt
                   | A.Leq     -> L.build_icmp L.Icmp.Sle
                   | A.Greater -> L.build_icmp L.Icmp.Sgt
                   | A.Geq     -> L.build_icmp L.Icmp.Sge
                  ) e1' e2' "tmp" env.builder)
             )
         )
      | A.Unop(op, e) ->
         let (env, e') = expr env e in
         (env,
          (match op with
             A.Neg     -> L.build_neg
           | A.Not     -> L.build_not) e' "tmp" env.builder)
      | A.Assign (s, e) ->
         let (env, e') = expr env e in
         let addr = lookup env s in
         ignore (llstore e' addr env.builder);
         (env, e')

      | A.Call (A.NaiveId("printi"), [e]) ->
         let (env, v) = expr env e in
         (env, (L.build_call printf_func
                             [| int_format_str ; v |]
                             "printi" env.builder))

      | A.Call (A.NaiveId("printb"), [e]) ->
         let (env, v) = expr env e in
         let bstring = L.build_select v true_str false_str "select"
  env.builder in
         (env, (L.build_call printf_func
                             [| bstring |]
                             "printb" env.builder))

      | A.Call (A.NaiveId("print"), [e]) ->
         let (env, v) = expr env e in
         (env, L.build_call printf_func
                            [| v |] "print"
                            env.builder)

      | A.Call (A.NaiveId("printendl"), [e]) ->
         let (env, v) = expr env e in
         (env, L.build_call printf_func
                            [| printendl_str; v |] "printendl"
                            env.builder)


      | A.Call (A.NaiveId("printf"), [e]) ->
         let (env, v) = expr env e in
         (env, L.build_call printf_func
                            [| float_format_str ; v |]
                            "printf" env.builder)

      | A.Call(A.NaiveId("concat"), [c1; c2]) ->
         let (env', v1) = expr env c1 in
         let (env'', v2) = expr env' c2 in
         (env'', L.build_call str_concat_func [| v1; v2 |] ""
  env''.builder)

      | A.Call(A.NaiveId("sizeof"), [e]) ->
         let (env', v1) = expr env e in
         let i64_size = L.size_of (L.type_of v1 )in
         let i32_size = L.build_intcast i64_size i32_t "" env'.builder
  in
         (env', i32_size)

      | A.Call(A.NaiveId("parallel"), [f; pool; nthread]) ->
         let (env1, llf) = expr env f in
         let (env2, llpool) = expr env1 pool in
         let (env3, llnthread) = expr env2 nthread in

         let etype = L.element_type (L.type_of llpool) in
         let size = L.size_of etype in
         let i32_size = L.build_intcast size i32_t "" env3.builder in

         let fcast = L.build_bitcast llf subroutine_t "" env3.builder
  in
         let poolcast = L.build_bitcast llpool (L.pointer_type
  voidstar_t) "" env3.builder in
         let v_arr = [| fcast; poolcast; i32_size; llnthread |]  in
         (env3, L.build_call parallel_func v_arr "" env3.builder)

      | A.Call(A.NaiveId("free"), [e]) ->
         let (env1, e') = expr env e in
         let ept = L.build_bitcast e' voidstar_t "" env1.builder in
         (env1, L.build_free ept env1.builder)

      | A.Call(A.NaiveId("malloc"), [e]) ->
         let (env1, e') = expr env e in
         (env1, L.build_array_malloc i8_t e' "" env1.builder)

      | A.Call(A.NaiveId("mutex_create"), _) ->
         (env, L.build_call mutex_create_func [||] "" env.builder)

      | A.Call(A.NaiveId("mutex_lock"), [e]) ->
         let (env1, e') = expr env e in
         let mut = L.build_bitcast e' voidstar_t "" env1.builder in
         (env1, L.build_call mutex_lock_func [|mut|] "" env1.builder)
      | A.Call(A.NaiveId("mutex_unlock"), [e]) ->
         let (env1, e') = expr env e in
         let mut = L.build_bitcast e' voidstar_t "" env1.builder in
         (env1, L.build_call mutex_unlock_func [|mut|] "" env1.builder)

      | A.Call(A.NaiveId("mutex_destroy"), [e]) ->
         let (env1, e') = expr env e in
         let mut = L.build_bitcast e' voidstar_t "" env1.builder in
         (env1, L.build_call mutex_destroy_func [|mut|] ""
  env1.builder)

      | A.Call (f, act) ->
         let fptr = lookup env f in
         debug ("fptr = " ^ L.string_of_llvalue fptr);
         let fdef = L.build_load fptr "" env.builder in
         debug ("fdef = " ^ L.string_of_llvalue fdef);
         debug ("fdef type = " ^ L.string_of_lltype (L.type_of fdef));
         let (env, actuals) = List.fold_right
                                (fun e (env, values) ->
                                  let (env', v) = expr env e in
                                  (env', v :: values))
                                act
                                (env, [])
         in
         (env, L.build_call fdef (Array.of_list actuals) ""
  env.builder)

      | A.Lambda (rt, blist, e) ->
         let name = "lambda_func" in
         let formal_types =
           Array.of_list (List.map (fun (t, _) -> ltype_of_typ t)
  blist)
         in
         let ftype = L.function_type (ltype_of_typ rt) formal_types in
         let fval  = L.define_function name ftype the_module in
         let fdecl = {
             A.typ   = rt;
             A.fname = name;
             A.formals = blist;
             A.body = [A.Return e]
           } in
         build_function_body env (fval, fdecl);
         (env, fval)
```

```
  (* Fill in the body of the given function *)
  and build_function_body fenv (fval, fdecl) =
    let fbuilder = L.builder_at_end context (L.entry_block fval) in

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

    (* Build the code for the given statement; return the builder for
       the statement's successor *)
    let rec stmt env = function
        A.Block sl ->
        let new_bb = L.append_block context "block" fval in
        let cont_bb = L.append_block context "cont" fval in
        let nenv = {
            locals = StringMap.empty;
            externals =
              StringMap.merge (fun _ xo yo -> match xo,yo with
                                              | Some x, Some _ ->
Some x
                                              | None, yo -> yo
                                              | xo, None -> xo )
env.locals env.externals;
            builder = L.builder_at_end context new_bb
          } in

        let nenv' = (List.fold_left stmt nenv sl) in
        add_terminal env (L.build_br new_bb);
        add_terminal nenv' (L.build_br cont_bb);
        { env with builder = L.builder_at_end context cont_bb }

      | A.Expr e -> fst (expr env e)
      | A.Return e ->
         let (env', v) = expr env e in
         let ret = (match fdecl.A.typ with
                     A.DataType(A.Void) -> L.build_ret_void
env'.builder
                   | _ -> L.build_ret v env'.builder) in
         debug ("ret = " ^ L.string_of_llvalue ret);
         env'
      | A.If (predicate, then_stmt, else_stmt) ->
         let (env', bool_val) = expr env predicate in
         let merge_bb = L.append_block context "merge" fval in

         let then_bb = L.append_block context "then" fval in
         add_terminal (stmt {env' with builder = (L.builder_at_end
context then_bb)} then_stmt)
                      (L.build_br merge_bb);

         let else_bb = L.append_block context "else" fval in
         add_terminal (stmt {env' with builder = (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
env.builder);
         {env with builder = L.builder_at_end context merge_bb}

      | A.While (predicate, body) ->
         let pred_bb = L.append_block context "while" fval in
         ignore (L.build_br pred_bb env.builder);

         let pred_builder = L.builder_at_end context pred_bb in
         let (env', bool_val) = expr {env with builder =
pred_builder} predicate in

         let body_bb = L.append_block context "while_body" fval in
         add_terminal (stmt {env' with builder = (L.builder_at_end
context body_bb)} body)
                      (L.build_br pred_bb);

         let merge_bb = L.append_block context "merge" fval in
         ignore (L.build_cond_br bool_val body_bb merge_bb
pred_builder);
         {env with builder = L.builder_at_end context merge_bb}

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

      | A.Local (vd) ->
         begin match vd with
```

```
          A.Bind(t, id) ->
          let local_var =
            begin match (S.resolve_user_type t user_types) with
              A.Struct(_, _) ->
              let struct_ptr_t = ltype_of_typ t in
              debug ("struct_ptr_t = " ^ L.string_of_lltype
struct_ptr_t);
              let struct_t = L.element_type struct_ptr_t in
              debug ("struct_t = "  ^ L.string_of_lltype struct_t);
              let struct_ptr = L.build_alloca struct_ptr_t ""
env.builder in
              let struct_val = L.build_malloc struct_t ""
env.builder in
              ignore (llstore struct_val struct_ptr env.builder);
              struct_ptr
            | _ -> L.build_alloca (ltype_of_typ t) id env.builder
            end in
          {env with locals = StringMap.add id local_var env.locals}
        | A.Binass(t, id, e) ->
           let local_var =  L.build_alloca (ltype_of_typ t) id
env.builder in
           let (env, e') = expr env e in
           let env' = {env with locals = StringMap.add id local_var
env.locals} in
           let p = (lookup env' (A.NaiveId id)) in
           ignore (llstore e' p env.builder);
           env'
        end
    in

    (* Build the code for each statement in the function *)
    let add_formal m (t, n) p =
      L.set_value_name n p;
      debug ("adding formal " ^ n ^ " of type " ^ L.string_of_lltype
(ltype_of_typ t));
      let local = L.build_alloca (ltype_of_typ t) n fbuilder in
      ignore (L.build_store p local fbuilder);
      StringMap.add n local m
    in

    let formals = List.fold_left2 add_formal StringMap.empty
fdecl.A.formals
                                  (Array.to_list (L.params fval)) in
    let init_env = {
        externals = StringMap.merge (fun _ xo yo ->
                        match xo,yo with
                        | Some x, Some _ -> Some x
                        | None, yo -> yo
                        | xo, None -> xo ) fenv.locals
fenv.externals;
        locals = formals;
        builder = fbuilder
      } in

    let env = List.fold_left stmt init_env fdecl.A.body in

    (* Add a return if the last block falls off the end *)
    begin match fdecl.A.typ with
      A.DataType(A.Void) -> add_terminal env L.build_ret_void
    | _ ->
       begin match L.block_terminator (L.insertion_block env.builder)
with
         Some _ -> ()
       | None -> raise (Failure ("missing return statement in
function " ^ fdecl.A.fname))
       end
    end;
  in

  let rec init_of_type t =
    match t with
      A.DataType(A.Int) -> L.const_int (ltype_of_typ t) 0
    | A.DataType(A.Bool) -> L.const_int (ltype_of_typ t) 0
    | A.DataType(A.Float) -> L.const_float (ltype_of_typ t) 0.0
    | A.DataType(A.String) -> L.const_string context ""
    | A.List(_) -> L.const_null (ltype_of_typ t)
    | A.Struct(_, _) -> L.const_null (ltype_of_typ t)
    | A.UserType(_) -> let t' = S.resolve_user_type t user_types in
                        init_of_type t'
    | A.FuncType(_) -> L.const_null (ltype_of_typ t)
    | _ -> raise (Failure ("Global variable with unsupported type: "
^ (A.string_of_typ t)))
  in

  (* Declare each global variable; remember its value in a map *)
  let add_global env stmt = match stmt with
      A.Global(vd) ->
      begin match vd with
        A.Bind(t, id) ->
        let init = init_of_type t in
        debug ("init = " ^ (L.string_of_llvalue init));
        let var = L.define_global id init the_module in
        { env with externals = StringMap.add id var env.externals }
      | A.Binass(t, id, e) ->
         let init = init_of_type t in
         debug ("init = " ^ (L.string_of_llvalue init));
         let var = L.define_global id init the_module in
         debug ("gvar type = " ^ (L.string_of_lltype (L.type_of
var)));
         let (env', lval) = (expr env e) in
         ignore (llstore lval var env'.builder);
         { env' with externals = StringMap.add id var env'.externals
}
      end
    | _ -> env
  in

  let global_func_map =
    StringMap.mapi (
        fun name (fval, _) ->
        let ft = L.type_of fval in
        let fvar = L.define_global (name ^ "_ptr")
(L.const_pointer_null ft) the_module in
        ignore (llstore fval fvar global_builder);
        debug ("fvar type = " ^ (L.string_of_lltype (L.type_of
fvar)));
        fvar
      ) function_decls
  in

  let global_env =
    List.fold_left add_global
                   { externals = global_func_map;
                     locals = StringMap.empty;
                     builder = global_builder }
                   global_stmts
  in
  StringMap.iter (fun _ fd -> build_function_body global_env fd)
function_decls;

  let llmem = L.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
(* Top-level of the Harmonica 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)
```

preprocess.py

```
import sys

KEYWORD = "alias"

if __name__ == "__main__":
    for fname in sys.argv[1:]:
        f = open(fname)
        lines = f.readlines()
        for i in range(len(lines)):
            line = lines[i]
            if line.startswith(KEYWORD):
                components = line.split()
                if len(components) != 3:
                    print >> sys.stderr, ('invalid syntax: ' + line)
                    sys.exit(1)
                orig = components[1]
                new  = components[2]
                for j in range(i+1, len(lines)):
                    lines[j] = lines[j].replace(orig, new)
        for line in lines:
            if line.startswith(KEYWORD):
                continue
            print line,
```

bindings.c
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void* str_concat(void *str1, void *str2)
{
  strcat((char *)str1, (char *)str2);
  return str1;
}

void int_to_string(int n, void *buf)
{
  sprintf(buf, "%d", n);
}

int parallel(void *(*start_routine) (void *), void *arg, int asize,
int nthreads)
{
  pthread_t thread[nthreads];
  int i;
  for (i = 0; i < nthreads; i ++) {
    void *addr = (void *) (((char *) arg) + i * asize);
    int err = 0;
    err = pthread_create(&thread[i], NULL, start_routine, *(void **)
addr);
    if (err != 0) {
      exit(err);
    }
  }

  int err = 0;
  for (i = 0; i < nthreads; i++) {
    err = pthread_join(thread[i], NULL);
    if (err != 0) {
      break;
    }
  }

  if (err != 0)
    perror("pthread_join: ");
  return err;
}

void* mutex_create(void) {
      pthread_mutex_t* mtx = (pthread_mutex_t *)
malloc(sizeof(pthread_mutex_t));
      if (mtx == NULL) {
            perror("mutex allocation failed: ");
      }
      int err = pthread_mutex_init(mtx, NULL);

      if (err != 0) {
            perror("mutext init failed");
            free(mtx);
            return NULL;
      }

      return mtx;
}

int lock(void* mtx) {
      int err = pthread_mutex_lock((pthread_mutex_t *) mtx);
      if (err != 0) {
            perror("mutex lock");
            return err;
      }
      return 0;
}

int unlock(void* mtx) {
      int err = pthread_mutex_unlock((pthread_mutex_t *) mtx);
      if (err != 0) {
            perror("mutex unlock");
            return err;
      }
      return 0;
}

int destroy(void* mtx) {
      int err = pthread_mutex_destroy((pthread_mutex_t *) mtx);
      if (err != 0) {
            perror("mutex destroy");
            return err;
      }
      return 0;
}
```

compile (script)

```
#!/bin/bash

python preprocess.py $@ | ./harmonica.native
```

Libraries

```
==> bst.ha <==
struct Node {
      int  value;
      Node lchild;
      Node rchild;
  mutex lock;
};

Node dbRoot = createNode(0);

Node createNode(int val) {
  Node node;
  node.lchild = NULL;
  node.rchild = NULL;
  node.value = val;
  node.lock  = mutex_create();
  return node;
}

void destroyNode(Node n) {
  mutex_destroy(n.lock);
  free(n);
}

# returns list with two elements: [target, parent of target]
# assumption: root node is locked by calling thread
list[Node] search(Node root, int val) {
      Node next = NULL;
  Node parent = NULL;
  Node target = NULL;

      if (val < root.value) {
            if ((next = root.lchild) == NULL) {
                  target = NULL;
            } else {
                  mutex_lock(next.lock);
                  if (val == next.value) {
                        target = next;
                  } else {
                        mutex_unlock(root.lock);
                        return search(next, val);
                  }
            }
        } else {
            if ((next = root.rchild) == NULL) {
                target = NULL;
            } else {
                mutex_lock(next.lock);
                if (val == next.value) {
                    target = next;
                } else {
                    mutex_unlock(root.lock);
                    return search(next, val);
                }
            }
        }

  parent = root;
      return [target, root];
}

int add(Node root, int value)
{
      mutex_lock(root.lock);
  list[Node] result = search(root, value);
  Node target = result[0];
      Node parent = result[1];

  if (target != NULL) {
    if (parent != NULL) {
     mutex_unlock(result[1].lock);
    }
    mutex_unlock(result[0].lock);
    return 0;
  }

  Node newnode = createNode(value);

  if (value < parent.value)
    parent.lchild = newnode;
  else
    parent.rchild = newnode;

  mutex_unlock(parent.lock);
  return 1;
```

```
}

void inorderMap(<Node void> f, Node root) {
  if (root == NULL)
    return ;
  inorderMap(f, root.lchild);
  f(root);
  inorderMap(f, root.rchild);
}

struct ThreadArgs {
  int threadNum;
  int seed;
};

int a = 97;
int c = 1;
int m = 65536;

int insertNumbers(ThreadArgs args) {
  int rand = args.seed;
  int i;
  for (i = 0; i < 30; i += 1) {
    print("inserting at thread ");
    printi(args.threadNum);
    rand = a * rand + c;
    rand = rand - (rand / m) * m;
    add(dbRoot, rand);

    # go do something else
    int j = 0;
    int s = 0;
    for (j = 0; j < 100000; j += 1)
      s += j;
  }

  return 0;
}

int main() {
  list[ThreadArgs] argsList = [NULL, NULL, NULL, NULL];

  int i;
  for (i = 0; i < 4; i += 1) {
    ThreadArgs args;
    args.threadNum = i + 1;
    args.seed = i * 100 + 364;
    argsList[i] = args;
  }

  parallel(insertNumbers, argsList, 4);

  <Node void> printNode = lambda (Node n) void ( printi(n.value) );
  inorderMap(printNode, dbRoot);
  return 0;
}

==> math.ha <==
# Math Library

float powi(float x, int n){
    if (n==0){
        return 1.0;
    }

    if (n>0){
        int i = 0;
        float y = 1.0;
        for (i=0; i<n; i=i+1){
            y = y*x;
        }
        return y;
    } else {
        int n_ = 0 - n;
        return (1.0/powi(x, n_));
    }
}

float factorialf(float x){
    if (x==0.0){
        return 1.0;
    }
    return x*factorialf(x-1);
}

int factorial(int x){
    if (x==0){
        return 1;
    }
    if (x<0){
        return 0;
    }
    return x*factorial(x-1);
}

float exp(float x){
    float taylor = 0.0;
    int i = 0;
    float fi = 0.0;
    float up;
    float down;
    for (i=0; i<99; i=i+1){
        up = powi(x, i);
        down = factorialf(i/1.0);
        taylor = taylor + (up/down);
        fi = fi + 1.0;
    }
    return taylor;
}

float ln(float x){
    float taylor = 0.0;
    int i;
    float tmp;
    for (i=0; i<99; i=i+1){
        int i_p = 2*i + 1;
        tmp = 2*( powi((x-1)/(x+1), i_p) )/i_p;
        taylor = taylor + tmp;
    }
    return taylor;
}

float pow(float x, float y){
    return exp(y*ln(x));
}

float sqrt(float x){
    return pow(x, 0.5);
}
```

```
==> vector.ha <==
alias T int
alias INIT_SIZE 16

struct vector_T {
  list[T] elements;
  int length;
  int memsize;
};

T __dummy_T;

vector_T vector_T_create() {
  vector_T v;
  v.elements = malloc(sizeof(__dummy_T) * INIT_SIZE);
  v.length = 0;
  v.memsize = INIT_SIZE;
  return v;
}

void vector_T_append(vector_T v, T elem) {
  if (v.length >= v.memsize) {
    v.memsize = v.memsize * 2;
    list[T] dest = malloc(sizeof(__dummy_T) * v.memsize);
    int i;
    for (i = 0; i < v.length; i += 1) {
      dest[i] = v.elements[i];
    }
    v.elements = dest;
  }

  v.elements[v.length] = elem;
  v.length += 1;
}

T vector_T_get(vector_T v, int i) {
  if (i >= v.length)
    print("ERROR: vector index out of bounds");
  return v.elements[i];
}

void vector_T_set(vector_T v, int i, T elem) {
  if (i >= v.length)
    print("ERROR: vector index out of bounds");
  v.elements[i] = elem;
}

void vector_T_remove(vector_T v, int i) {
  if (i >= v.length)
    print("ERROR: vector index out of bounds");

  int j;
  for (j = i; j < v.length - 1; j += 1) {
    v.elements[j] = v.elements[j+1];
  }
  v.length -= 1;
}

void vector_T_destroy(vector_T v) {
  free(v.elements);
  free(v);
}
```

Tests

```
==> fail-assign1.ha <==
int main()
{
  int i;
  bool b;

  i = 42;
  i = 10;
  b = true;
  b = false;
  i = false; # Fail: assigning a bool to an integer
}

==> fail-assign2.ha <==
int main()
{
  int i;
  bool b;

  b = 48; # Fail: assigning an integer to a bool
}

==> fail-assign3.ha <==
void myvoid()
{
  return;
}

int main()
{
  int i;

  i = myvoid(); # Fail: assigning a void to an integer
}

==> fail-assign4.ha <==
int main()
{
  int i = "42"; # fail: assigning string to integer
}

==> fail-assign5.ha <==
typedef int userint;
typedef userint ss;
typedef ss newtype;

int main() {
  newtype a = [1,2,3];
  return 0;
}

==> fail-dead1.ha <==
int main()
{
  int i;

  i = 15;
  return i;
  i = 32; # Error: code after a return
}

==> fail-for1.ha <==
int main()
{
  int i;
  for (;;) {} # OK: Forever

  for (i = 0 ; i < 10 ; i = i + 1) {
    if (i == 3) return 42;
  }

  for (j = 0; i < 10 ; i = i + 1) {} # j undefined

  return 0;
}

==> fail-for2.ha <==
int main()
{
  int i;

  for (i = 0; j < 10 ; i = i + 1) {} # j undefined

  return 0;
}
```

```
==> fail-for3.ha <==
int main()
{
  int i;

  for (i = 0; i ; i = i + 1) {} # i is an integer, not Boolean

  return 0;
}

==> fail-for4.ha <==
int main()
{
  int i;

  for (i = 0; i < 10 ; i = j + 1) {} # j undefined

  return 0;
}

==> fail-for5.ha <==
int main()
{
  int i;

  for (i = 0; i < 10 ; i = i + 1) {
    foo(); # Error: no function foo
  }

  return 0;
}

==> fail-func1.ha <==
int foo() {}

int bar() {}

int baz() {}

void bar() {} # Error: duplicate function bar

int main()
```

```
{
  return 0;
}

==> fail-func10.ha <==
bool foo(<int bool> f, int c) {
     return f(c);
}

bool bar(int c) {
     return true;
}

int main() {
    foo(bar, 3);
    foo(bar, (false,2));
}

==> fail-func2.ha <==
int foo(int a, bool b, int c) { }

void bar(int a, bool b, int a) {} # Error: duplicate formal a in bar

int main()
{
  return 0;
}

==> fail-func3.ha <==
int foo(int a, bool b, int c) { }

void bar(int a, void b, int c) {} # Error: illegal void formal b

int main()
{
  return 0;
}

==> fail-func4.ha <==
int foo() {}

void bar() {}
```

```
int printi() {} # Should not be able to define printi

void baz() {}

int main()
{
  return 0;
}

==> fail-func5.ha <==
int foo() {}

int bar() {
  int a;
  void b; # Error: illegal void local b
  bool c;

  return 0;
}

int main()
{
  return 0;
}

==> fail-func6.ha <==
void foo(int a, bool b)
{
}

int main()
{
  foo(42, true);
  foo(42); # Wrong number of arguments
}

==> fail-func7.ha <==
void foo(int a, bool b)
{
}

int main()
{
  foo(42, true);
  foo(42, true, false); # Wrong number of arguments
}

==> fail-func8.ha <==
void foo(int a, bool b)
{
}

void bar()
{
}

int main()
{
  foo(42, true);
  foo(42, bar()); # int and void, not int and bool
}

==> fail-func9.ha <==
void foo(int a, bool b)
{
}

int main()
{
  foo(42, true);
  foo(42, 42); # Fail: int, not bool
}

==> fail-global1.ha <==
int c;
bool b;
void a; # global variables should not be void


int main()
{
  return 0;
}

==> fail-global2.ha <==
int b;
```

```
bool c;
int a;
int b; # Duplicate global variable

int main()
{
  return 0;
}

==> fail-global3.ha <==
int a = 3;
bool b = true;
int c = f(a);
int d = c;

int f(int x) {
    return x;
}

int e = f(b);

int main() {
}

==> fail-if1.ha <==
int main()
{
  if (true) {}
  if (false) {} else {}
  if (42) {} # Error: non-bool predicate
}

==> fail-if2.ha <==
int main()
{
  if (true) {
    foo; # Error: undeclared variable
  }
}

==> fail-if3.ha <==
int main()
{
  if (true) {
    42;
  } else {
    bar; # Error: undeclared variable
  }
}

==> fail-lambda1.ha <==
struct node {
  int x;
  string name;
};


int main() {
    node n;
    <node node> f = lambda (node n) node (n.x);
    f(n);

    return 0;
}
==> fail-return1.ha <==
int main()
{
  return true; # Should return int
}

==> fail-return2.ha <==
void foo()
{
  if (true) return 42; # Should return void
  else return;
}

int main()
{
  return 42;
}

==> fail-scope1.ha <==
int a = 3;

int main() {
    printi(a);
    {
        int b = 3;
        printi(b);
    }

    printi(b);
    return 0;
}

==> fail-while1.ha <==
int main()
{
  int i;

  while (true) {
    i = i + 1;
  }

  while (42) { # Should be boolean
    i = i + 1;
  }

}

==> fail-while2.ha <==
int main()
{
  int i;

  while (true) {
    i = i + 1;
  }

  while (true) {
    foo(); # foo undefined
  }

}

==> fib.ha <==
int fib(int x)
{
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}

int main()
{
  printi(fib(0));
  printi(fib(1));
  printi(fib(2));
  printi(fib(3));
  printi(fib(4));
  printi(fib(5));
  return 0;
}

==> test-add1.ha <==
int add(int x, int y)
{
  return x + y;
}

int main()
{
  printi( add(17, 25) );
  return 0;
}

==> test-arith1.ha <==
int main()
{
  printi(39 + 3);
  return 0;
}

==> test-arith2.ha <==
int main()
{
  printi(1 + 2 * 3 + 4);
  return 0;
}

==> test-array1.ha <==
int main() {
  list[int] l = [999,999,999,999];
  int a = 3;
  l[a] = 1;

  printi(l[0]);
  printi(l[3]);
  free(l);

  list[int] l2 = malloc(16);
  l2[0] = 14;
  l2[2] = 33;
  l2[3] = 75;
  l2[1] = 5;
  printi(l2[1]);

  return 0;
}

==> test-array2.ha <==
int main() {
  list[int] arr = [1,2,3];
  arr = [1,2,3,4,5,6];

  arr[1] = 4;
  printi(arr[0]);
  printi(arr[1]);

  arr[1] = 1340;
  printi(arr[1]);
  printi(arr[2]);
  printi(arr[4]);

  return 0;
}

==> test-array3.ha <==
int main() {
  int x = 10;
  list[int] arr = [x,3,x];
  printi(arr[0]);
  printi(arr[1]);
  printi(arr[2]);
  return 0;
```

```
}

==> test-cast1.ha <==
int main(){
    float a = 2.0;
    printf(3/a);
    float b = 2.0/4;
    printf(2.0/4);

    int i = 1;
    printf(i/1.0);

    return 0;
}

==> test-compound-assign.ha <==
int main() {
     int a;
     int range = 10;
     for (a = 0; a < range; a += 1) {
          printi(a);
     }
     return 0;
}

==> test-elementary-math.ha <==
int mod(int a, int b) {
  # mod function for natural numbers
  int x;
  int y;
  if (a>b){
    x = a;
    y = b;
  }else{
    return a;
  }

  while (x >= y){
    x = x - y;
  }
  return x;
}
```

```
int isPrime(int a){
  # see if an integer is prime number
  if (a<2){
    return 0;
  }else {
    int i = 0;
    for (i=2; i<a; i=i+1){
      if (mod(a, i)==0){
        return 0;
      }
    }
    return 1;
  }
  return 0;
}

float abs(float x){
  if (x>0.0) {
    return x;
  }

  if (x==0.0){
    return 0.0;
  }

  float y = 0.0 - x;
  return y;
}

float sqrt(float a){
  # a naive function to return the sqrt approximation. return -1 if a
is negative
  float err_tol = 0.000001;
  if (a < -0.0){
    return (-1.0);
  }

  float min = 0.0;
  float max = a;
  float middle = (min+max)/2.0;

  float x = middle*middle;
  while ( abs(x-a)>err_tol ){
    if (x < a){
      min = middle;
      max = max;
    }else{
      min = min;
      max = middle;
    }
    middle =  (min+max)/2.0;
    x = middle*middle;

  }

  return middle;

}


int main() {
  # test mod
  printi(mod(12, 5)); # 2
  printi(mod(7,21)); # 7
  printi(mod(35, 6)); # 5

  # test isPrime
  printi(isPrime(1)); # 0
  printi(isPrime(4)); # 0
  printi(isPrime(7)); # 1

  # test sqrt
  printf(sqrt(2.0)); # 1.414214
  printf(sqrt(-2.0)); # -1.000000
  printf(sqrt(4.0)); # 2.000000


  return 0;
}

==> test-fib.ha <==
int fib(int x)
{
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
```

```
}

int main()
{
  printi(fib(0));
  printi(fib(1));
  printi(fib(2));
  printi(fib(3));
  printi(fib(4));
  printi(fib(5));
  return 0;
}

==> test-for1.ha <==
int main()
{
  int i;
  for (i = 0 ; i < 5 ; i = i + 1) {
    printi(i);
  }
  printi(42);
  return 0;
}

==> test-for2.ha <==
int main()
{
  int i;
  i = 0;
  for ( ; i < 5; ) {
    printi(i);
    i = i + 1;
  }
  printi(42);
  return 0;
}

==> test-func-list.ha <==
int bar(int i) {
      return i;
}

int foo(int i) {
     return -i;
}

int main() {
     list[<int int>] arr = [bar, foo];

     int i;
     for (i = 0; i < 2; i = i + 1) {
          printi(arr[i](i * 10));
     }
     return 0;
}

==> test-func1.ha <==
int add(int a, int b)
{
  return a + b;
}

int main()
{
  int a;
  a = add(39, 3);
  printi(a);
  return 0;
}

==> test-func10.ha <==
list[int] f(int x) {
  list[int] ret = [x, x+1, x-1];
  return ret;
}

int main() {
  list[int] arr = f(3);
  printi(arr[0]);
  printi(arr[1]);
  printi(arr[2]);
  return 0;
}

==> test-func2.ha <==
int fun(int x, int y)
```

```
{
  return 0;
}

int main()
{
  int i;
  i = 1;

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

  printi(i);
  return 0;
}

==> test-func3.ha <==
void printiem(int a, int b, int c, int d)
{
  printi(a);
  printi(b);
  printi(c);
  printi(d);
}

int main()
{
  printiem(42,17,192,8);
  return 0;
}

==> test-func4.ha <==
int add(int a, int b)
{
  int c;
  c = a + b;
  return c;
}

int main()
{
  int d;
  d = add(52, 10);
  printi(d);
  return 0;
}

==> test-func5.ha <==
int foo(int a)
{
  return a;
}

int main()
{
  return 0;
}

==> test-func6.ha <==
void foo() {}

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

int main()
{
  printi(bar(17, false, 25));
  return 0;
}

==> test-func7.ha <==
int a;

void foo(int c)
{
  a = c + 42;
}

int main()
{
  foo(73);
  printi(a);
  return 0;
}

==> test-func8.ha <==
void foo(int a)
{
  printi(a + 3);
}

int main()
{
  foo(40);
  return 0;
}

==> test-func9.ha <==
bool foo(<int bool> f, int c) {
     return f(c);
}

bool bar(int c) {
     if (c == 3) {
        printi(c + 1);
        return false;
     }
     return true;
}

int main() {
    <int bool> g = bar;
    printb(foo(g, 3));
    return 0;
}

==> test-gcd.ha <==
int gcd(int a, int b) {
  while (a != b) {
    if (a > b) a = a - b;
    else b = b - a;
  }
  return a;
}

int main()
{
  printi(gcd(2,14));
  printi(gcd(3,15));
  printi(gcd(99,121));
  return 0;
```

```
}

==> test-gcd2.ha <==
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}

int main()
{
  printi(gcd(14,21));
  printi(gcd(8,36));
  printi(gcd(99,121));
  return 0;
}

==> test-global1.ha <==
int a;
int b;

void printia()
{
  printi(a);
}

void printib()
{
  printi(b);
}

void incab()
{
  a = a + 1;
  b = b + 1;
}

int main()
{
  a = 42;
  b = 21;
  printia();
  printib();
  incab();
  printia();
  printib();
  return 0;
}

==> test-global2.ha <==
bool i;

int main()
{
  int i; # Should hide the global i

  i = 42;
  printi(i + i);
  return 0;
}

==> test-global3.ha <==
int i;
bool b;
int j;

int main()
{
  i = 42;
  j = 10;
  printi(i + j);
  return 0;
}

==> test-global4.ha <==
list[int] arr = [1,2,3];

int main() {
  arr = [1,2,3,4,5,6];

  arr[1] = 4;
  printi(arr[0]);
  printi(arr[1]);

  arr[1] = 1340;
  printi(arr[1]);
  printi(arr[2]);
  printi(arr[4]);

  return 0;
}

==> test-hello.ha <==
# This program printis "hello world".
int main()
{
  print("hello world");
  return 0;
}

==> test-if1.ha <==
int main(){
      int i = 0;

      if (i == 0){
            printi(7);
      } else {
            printi(8);
      }

      int b;
      printi(9);

  return 0;
}

==> test-if2.ha <==
int main()
{
  if (true) printi(42); else printi(8);
  printi(17);
  return 0;
}

==> test-if3.ha <==
int main()
{
  if (false) printi(42);
  printi(17);
  return 0;
}

==> test-if4.ha <==
int main()
{
  if (false) printi(42); else printi(8);
  printi(17);
  return 0;
}

==> test-if5.ha <==
int cond(bool b)
{
  int x;
  if (b)
    x = 42;
  else
    x = 17;
  return x;
}

int main()
{
 printi(cond(true));
 printi(cond(false));
 return 0;
}

==> test-lambda1.ha <==
<int bool> isThree = lambda (int x) bool (x == 3);
<int void> printInt = lambda (int x) void ( printi(x) );

int main() {
    printb(isThree(3));
    printb(isThree(0));
    printInt(10);
    return 0;
}
==> test-lambda2.ha <==
void map(<int int> f, list[int] arr, int length) {
  int i;
  for (i = 0; i < length; i += 1) {
    arr[i] = f(arr[i]);
  }
}

int reduce(<int int int> f, list[int] arr, int length) {
  if (length <= 0)
    return 0;
  int acc = arr[0];
  int i = 1;
  for (i = 1; i < length; i += 1) {
    acc = f(acc, arr[i]);
  }
  return acc;
}

int main() {
  <int int> plus1  = lambda (int x) int ( x + 1 );
  <int int> times2 = lambda (int x) int ( x * 2 );
  <int int int> plus = lambda (int x, int y) int ( x + y );

  list[int] arr = [1,2,3,4,5,6];
  map(plus1, arr, 6);
  map(times2, arr, 6);

  int i;
  for (i = 0; i < 6; i += 1) {
    printi(arr[i]);
  }

  print("sum: ");
  printi(reduce(plus, arr, 6));

  return 0;
}

==> test-local1.ha <==
void foo(bool i)
{
  int i; # Should hide the formal i

  i = 42;
  printi(i + i);
```

```
}

int main()
{
  foo(true);
  return 0;
}

==> test-local2.ha <==
int foo(int a, bool b)
{
  int c;
  bool d;

  c = a;

  return c + 10;
}

int main() {
 int i = foo(37, false);
 printi(i);
 return 0;
}

==> test-member1.ha <==
struct person {
    int age;
    string name;
};

int main() {
    person p;
    p.age = 3;
    printi(p.age);

    person p2;
    p2.age = 4;
    printi(p2.age);
    printi(p.age);

    return 0;
}
```

```
==> test-mutex1.ha <==
mutex m;
list[int] arr;
int i = 0;

int f(int v) {
    mutex_lock(m);
    int j;
    int s = 0;
    arr[i] = i;
    i += 1;
    mutex_unlock(m);

    return 0;
}

int main() {
   m = mutex_create();
   arr = malloc(16);
   parallel(f, [0,0,0,0], 4);
   mutex_destroy(m);

   for (i = 0; i < 4; i += 1) {
       printi(arr[i]);
   }

   free(arr);

   return 0;
}

==> test-ood1.ha <==
struct point{
    float x;
    float y;
};



float abs(float x){
  if (x>0.0) {
    return x;
  } else {
    if (x==0.0){
      return 0.0;
    }
    float y = 0.0 - x;
    return y;
  }

  return x;
}

float sqrt(float a){
  # a naive function to return the sqrt approximation. return -1 if a
is negative
  float err_tol = 0.000001;
  if (a < -0.0){
    return (-1.0);
  }

  float min = 0.0;
  float max = a;
  float middle = (min+max)/2.0;

  float x = middle*middle;

  while ( abs(x-a)>err_tol ){
    if (x < a){
      min = middle;
      max = max;
    }else{
      min = min;
      max = middle;
    }
    middle =  (min+max)/2.0;
    x = middle*middle;

  }

  return middle;

}

struct line{
    point s;
    point e;
};

float l2norm(line l){
    float xdiff = l.s.x - l.e.x;
    float ydiff = l.s.y - l.e.y;
    float square = xdiff*xdiff+ydiff*ydiff;
    return sqrt(square);
}



int main(){
    point a;
    point b;
    a.x = 0.0;
    a.y = 1.0;
    b.x = 2.0;
    b.y = 3.0;

    line l;
    l.s = a;
    l.e = b;

    printf(l2norm(l)); # 2.828427

    return 0;
}

==> test-ops1.ha <==
int main()
{
  printi(1 + 2);
  printi(1 - 2);
  printi(1 * 2);
  printi(100 / 2);
  printi(99);
  printb(1 == 2);
  printb(1 == 1);
  printi(99);
  printb(1 != 2);
  printb(1 != 1);
  printi(99);
  printb(1 < 2);
  printb(2 < 1);
  printi(99);
  printb(1 <= 2);
  printb(1 <= 1);
  printb(2 <= 1);
  printi(99);
  printb(1 > 2);
  printb(2 > 1);
  printi(99);
  printb(1 >= 2);
  printb(1 >= 1);
  printb(2 >= 1);
  return 0;
}

==> test-ops2.ha <==
int main()
{
  printb(true);
  printb(false);
  printb(true && true);
  printb(true && false);
  printb(false && true);
  printb(false && false);
  printb(true || true);
  printb(true || false);
  printb(false || true);
  printb(false || false);
  printb(!false);
  printb(!true);
  printi(-10);
  printi(--42);

  return 0;
}

==> test-pow1.ha <==
float powi(float x, int n){
    if (n==0){
        return 1.0;
    }
    if (n>0){
        int i = 0;
        float y = 1.0;
        for (i=0; i<n; i=i+1){
            y = y*x;
        }
        return y;
    }

    int n_ = 0 - n;
    return (1.0/powi(x, n_));
}

float factorialf(float x){
    if (x==0.0){
        return 1.0;
    }
    return x*factorialf(x-1);
}

int factorial(int x){
    if (x==0){
        return 1;
    }
    if (x<0){
        return 0;
    }
    return x*factorial(x-1);
}

float exp(float x){
    float taylor = 0.0;
    int i = 0;
    float fi = 0.0;
    float up;
    float down;
    for (i=0; i<99; i=i+1){
        up = powi(x, i);
        down = factorialf(i/1.0);
        taylor = taylor + (up/down);
        fi = fi + 1.0;
    }
    return taylor;
}

float ln(float x){
    float taylor = 0.0;
    int i;
    float tmp;
    for (i=0; i<99; i=i+1){
        int i_p = 2*i + 1;
        tmp = 2*( powi((x-1)/(x+1), i_p) )/i_p;
        taylor = taylor + tmp;
    }
    return taylor;
}

float pow(float x, float y){
    return exp(y*ln(x));
}


int main(){
    printf(powi(2.5, 2)); # 6.250000
    printf(powi(0.5, -2)); # 4.000000

    printf(factorialf(4.0)); # 24.000000
    printf(exp(1.0)); # 2.718282
    printf(exp(2.0)); # 7.389056
    printf(exp(0.5)); # 1.648721

    printf(ln(3.0)); # 1.098612
    printf(ln(2.5)); # 0.916291

    printf(pow(2.0, 0.5)); # 1.414214
    printf(pow(7.2, -0.14));# 0.758531


    return 0;
}

==> test-scope1.ha <==
int main() {
    int b = 3;
    {
        int b = 4;
        printi(b);
    }
    printi(b);

    return 0;
}

==> test-scope2.ha <==
int a = 5;

int fb() {
  return a + 5;
}

int main() {
  int a = 99;
  printi(fb());

  return 0;
}
==> test-selection-sort.ha <==
void selectionSort(list[int] arr, int length) {
    if (length < 0)
      return ;

    int i;
    int j;
    for (i = 0; i < length; i = i + 1) {
      int k = i;
      for (j = i+1; j < length; j = j + 1) {
        if (arr[j] < arr[k])
          k = j;
      }

      int temp = arr[k];
      arr[k] = arr[i];
      arr[i] = temp;
    }
}

int main() {
  list[int] arr = [10,-356,2147483647,4,3,2,1];
  int length = 7;
  selectionSort(arr, length);
  int i;
  for (i = 0; i < length; i = i + 1) {
    printi(arr[i]);
  }

  return 0;
}

==> test-sizeof1.ha <==
struct person {
  int age;
};
person p;

int main() {
  printi(sizeof(0));
  printi(sizeof(p));
  return 0;
}
==> test-var1.ha <==
int main()
{
  int a;
  a = 42;
  printi(a);
  return 0;
}

==> test-var2.ha <==
int a;

void foo(int c)
{
  a = c + 42;
}

int main()
{
  foo(73);
  printi(a);
  return 0;
```

```
}

==> test-while1.ha <==
int main()
{
  int i;
  i = 5;
  while (i > 0) {
    printi(i);
    i = i - 1;
  }
  printi(42);
  return 0;
}

==> test-while2.ha <==
int foo(int a)
{
  int j;
  j = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}

int main()
{
  printi(foo(7));
  return 0;
}
==> test-parallel1.ha <==
struct p {
  int a;
  int b;
};

void f1(int x) {
  printi(x);
}

void f2(p x) {
  x.a += 1;
}

int main() {
  int a = 3;
  int b = 4;
  int c = 5;
  int d = 6;

  parallel(f1, [a, b, c, d], 4);

  p e;
  p f;

  e.a = 31;
  f.a = 41;

  parallel(f2, [e,f], 2);

  printi(e.a);
  printi(f.a);
}

==> test-preprocess1.ha <==
alias U T
alias T int

T addT(U x, U y) {
  return (x + y);
}

==> test-preprocess2.ha <==
alias U T
alias T float

T addT(U x, U y) {
  return (x + y);
}

int main() {
    printi(addint(6, 7));
    printf(addfloat(3.0, 7.0));
    return 0;
```

```
}

==> test-vector1.ha <==
int main() {
  vector_int vec = vector_int_create();

  int i;
  for (i = 0; i < 8; i += 1) {
    vector_int_append(vec, i * i);
  }

  vector_int_set(vec, 3, 1000);
  vector_int_set(vec, 6, 1000000);

  vector_int_remove(vec, 4);

  for (i = 0; i < vec.length; i += 1)
    printi(vector_int_get(vec, i));

  vector_int_destroy(vec);

  return 0;
}
```