TiMRS Timers, Made Readable and Simple

Final Report

Jeff Kline	jk4209
Faisal Rahman	fr2422
Daniel Rindone	dcr2165
Eric Webb	edw2139

COMS 4115 - Programming Languages and Translators
Prof. Stephen Edwards

Spring 2021

Contents

1.0 Introduction

TiMRS is a programming language designed to give a user a new way to easily design and script complex timers for any task. This concept is centered around simplicity. We want a user to be able to code any timing procedure using TiMRS' built-in timer type with easily customizable durations and possibilities for repeats, intervals, and other pre-built timing patterns to track and perform any task where it may be useful.

TiMRS is an emulation of C orientated syntax; incorporating C allows the compiler to parse easier and allow more direct program memory management, should it be needed. A major component of running a TiMRS script is providing the user with an indication of progress throughout the timing routine.

This report will go into depth to describe the TiMRS language. The goal of this language is to give users a new way to easily design and script complex timers for any task. This method allows one to code complex timing procedures using TiMRS' built-in timer types with customizable durations and functionalities. These include accounting for repetition, intervals, multiple processes, saving sessions, and other techniques to track and perform any task where it may be useful.

Our language aims to provide a simple solution to implement flexible and dynamic timing systems for multiple different uses. This paper is structured to be used as a guide to properly and effectively implementing this language.

2.0 Language Tutorial

This section is designed to present a quick instructional overview of the TiMRS language:

2.1 Programming Basics

2.1.1 Building a Program

Writing programs in TiMRS is similar to elements found in C and MicroC. Programs intended to run on TiMRS must be saved as .mc files. The following is an example of a TiMRS program:

```
main(){
      int a;
```

```
        string k;
        a = 5000;
        k = "Done";
        init_timer(1);
        start_timer(a);
        prints(k);
        timer_destroy(1);
}
```

Functions are declared such as the `main()` above. All code within the function must be encased by `{}` . Any line of code other than a function declaration, brackets or line breaks must be followed by `;` . Variables must be instantiated and declared in separate statements.

2.1.2 Compiling a Program

Compiles all code and runs runs full test suite of testers inside test directory: **`$ make all`**
Compiles all code without running test suite: **`$ make`**

2.2 Hello World

To print your first string, simply compile a program with the following code:

```
print("hello world");
```

2.3 Running the Program

To run the program, simply compile your .mc file and run the output file.

3.0 Language Reference Manual

3.1 Lexical Conventions

For TiMRS, the approach toward the lexical conventions of the language emulates many of the features found within MicroC with specific catering to areas that align with the goal of a timer-based language like ours.

3.2 Syntax Notation

Throughout the course of this manual, specific syntax as it is reflected in the TiMRS language will be designated by the `Courier New` font style.

The semi-colon character, as it does in C, acts as the statement terminator for all elements found within a function. Code blocks are indicated by the use of a tab character which itself consists of four spaces. Please see below for a use case:

Example:

```
int main()
{
     int a;
     a = 5000;
}
```

3.2.1 Comments

Comments will be introduced by using a forward slash followed by an asterisk. Everything within that asterisk to the close of the comment through the reverse of that syntax is considered a comment, an example is provided below:

Example:

```
/* this is a comment */

/* /* Nested /* Comments */ Are Unnecessary but */ supported */
```

3.2.2 Constants

There are three constants found in this language, further explanation of each constant is described in the following sections:

Numerical Constants:

- Integers
- Floating point numbers

String Constants:

- Strings

3.2.3 Numerical Constants

In the TiMRS language, a numerical constant refers to any collection of numbered digits. Only whole integers and decimal float numbers are recognized, and all other classifications are excluded from recognition. All numbers must be non-negative.

Example:

Whole numbers:

34
4
1

Float numbers:

55.0
4.4
0.3

3.2.4 String Constants

A string constant for the TiMRS language is indicated through any given sequence of numbers and characters that are surrounded by double quotation marks. These marks are represented as:

Example:

```
/* label */
int main()
{
     f = “heat oven”
     prints(“heat oven”);
}
```

A string literal classifies under the datatype ‘string’ and is terminated by a null byte `\0` to indicate the end of the string. Note that strings to not support escape sequences such as `\n` or `%d` as seen in most c-type languages.

3.2.5 Data Types

There are six primitive data types recognized in TiMRS:
`int, float, string, bool, void, and array`

Please see examples of use below:

Primitive data types:

Type	Description
`int`	32-bit Integer Example: `10`
`float`	64-bit Floating point number Example: `4.5`
`string`	Array of ASCII characters: Example: `"hello"`
`bool`	1-bit Boolean variable Example: `True`
`void`	Empty data type, no value

3.3 Functions

The user is able to formulate specific user-defined functions. These functions have arguments and return types that depend on the output of the given function. A function begins with the declaration of the function's return type followed by the function's name and any number of parameters. When initially defining a function, the function's statement is contained within curly braces to define the scope of what occurs in the function.

Syntax:

```
/* function declaration */
int name(list of parameters)
{
     statement;
}

/* function call */
name(list of parameters)
```

Example:

```
/* user-defined function declaration */
int countdown(int x, string msg)
{
     if (x <= 30)
     return msg;
}

/* function call */
countdown(10, "done")
```

3.3.1 The Main Function

The main function within TiMRS serves as the designated starting point for any program's execution. This function organizes the code by directing calls to other functions that lie within the rest of the program.

Syntax:

```
/* main function declaration */
int main(){
     statement;
}
```

3.3.2 Standard TiMRS Functions

A number of pre-constructed library functions assist with key timer management aspects within the TiMRS language. These operate exactly how functions are described above, but in this case their behavior cannot be modified by the user.

Functions included in TiMRS cover a number of basic needs when constructing a timer, these include:

`init_timer(int x)`

Creates an instance of a timer. The argument is dumped in the function body.

`start_timer(int duration)`

Starts a timing event on an indicated timer. It's parameter is the amount of time it should count down, in milliseconds.

`timer_destroy(int x)`

Frees the timer from memory. The argument passed is dumped in the function body.

`prints(str s)`

String printing function within TiMRS. String may be passed as a variable or implicitly:

```
prints("Hello world")
```

3.4 Expressions and Operators

The TiMRS language follows the conventions of MicroC when it comes to grouping expressions. Expressions in this language include integer expressions, Boolean expressions, relational expressions, and string expressions. A more in-depth explanation of these expressions and their operators are listed in the following sections.

3.4.1 Integer Expressions

Expressions involving integers often include mathematical arithmetic expressions, these include use of the infix operators:

`+, -, *, /, and ()`	Standard operations for mathematical arithmetic

Precedence rules follow standard conventions in TiMRS integer expressions. Operands may be used through one of the three int and float complex data types described in section **3.2.5 - Data**

Types, or are defined as numbers expressed through the definition at **3.2.3 - Numerical Constants.**

Example:

```
int k;

k = 1 + (2 * 3);

int m;

m = k / 22
```

3.4.2 Boolean Expressions

Examples of Boolean expressions in the TiMRS language begin with either the keywords *`true`* or *`false`*.

Logical operations in TiMRS include the following:*Logical Operators:*

`||`	Standard logical `OR` Returns true if either of the left or right boolean expressions evaluate to true. Example: `a || b`
`&&`	Standard logical `AND` Returns true if the left and right boolean expressions both evaluate to true, otherwise it returns false. Example: `b && a`
`!`	Standard logical `NOT` Applies as a unary operator to a boolean expression. Results in true if the value of the operand is false and vice-versa. Example: `!b && a`

The grouping of the above operators is from left-to-right.

Additional elements of Boolean expressions include the formation of relational expressions as defined in the following section.

3.4.3 Relational Expressions

Operators to express relational expressions are of the following form:

!=	Inequality
==	Equality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Operands for these comparisons are integers. The result of the comparison is a bool whose value is true if the comparison evaluates to true. Otherwise, it returns false.

The grouping of the above operators is from left-to-right.

3.4.4 Operator Precedence

( )	Highest
!	
*, /, %	
+, -	
>, <, <=, >=	
==, !=	

&&	
||	
=	Lowest

3.5 Declarations

Variables must be declared in TiMRS before any value can be assigned specifically to it. Declarations in TiMRS, as well as their initialization and assignment, take the following form and are explained in more detail in the subsequent sections:

Example:

```
/* various declarations */
int first_pizza;

first_pizza = “hawaiian”;

float extra_time;
extra_time = 30.0;
```

3.5.1 Initialization

There are various ways to initialize variables, TiMRS-specific calls, and functions. Specific examples are provided in the following section.

3.5.2 Variables and Assignment

Assigning values to variables in TiMRS is done using the following operator:

=	Assigns the variable on the right hand side to the variable on the left

The initialization of a variable or a timer can happen during or after declaration:

Example:

```
string first_pizza;
first_pizza = "cheese";
```

3.6 TiMRS-Specific Functions

TiMRS includes a number of built-in functions to assist with timer management within the language. Below, those commands are explained in more detail.

TiMRS-specific functions:

`init_timer()`	Create an instance of a timer
`start_timer()`	Starts a timing event
`timer_destroy()`	Frees the timer from memory
`prints()`	String printing function

Example:

```
/* simple timer that prints a string after a given period
of time (in milliseconds) */

main(){
int a;
string k;
     a = 5000;
     k = "Done";
/* creates a timer */
     init_timer(1);
/* begins the timer */
     start_timer(a);
     prints(k);
/* frees the timer's memory */
     timer_destroy(1);
}
```

3.7 Control Flow and Conditional Statements

In TiMRS, `if, elif, else,` and `while` statements are used to assist in conditional statements and loops that the rounds command can not achieve. See below for their use cases.

3.7.1 If / Elif / Else Statements

For these statements, the code content must be contained within curly braces.
In TiMRS, `if, elif, else` statements are used by implementing the following:

Syntax:

```
if (expression){
     statement;
}
elif (expression){
     statement;
}
else{
     statement;
}
```

Example:

```
/* if statement */
if (first_pizza == "hawaiian"){
     init_timer(1);
     start_timer(a);
}

/* elif statement */
elif (first_pizza == "cheese"){
     start_timer(b);
}

/* else statement */
else{
     timer_destroy(1);
}
```

3.7.2 While Loops

In TiMRS, `while` statements are used by implementing the following:

Syntax:

```
while(condition expression){
     statement;
}
```

Example:

```
/* while statement */
int a;
a = 0;
while(a < 10){
     init_timer(1);
     start_timer(a);
     a++;
}
timer_destroy(1);
```

3.8 Lexical Scope

The scope of a variable in TiMRS is where the variable lives in the program and where that variable can be accessed within the code. Global variables can be accessed throughout the entirety of the code while local variables can only be accessed within their function.

4.0 Project Plan

4.1 Process

4.1.1 Planning

We began the class by immediately establishing a well-organized and active Slack channel through which we could share ideas, collect our thoughts, and communicate. We aimed to use this feature to have consistent discourse among the group and dynamically set up time for meetings where necessary. Early on, we would also meet and identify where TA help would be required to keep progress and stay on track. Within the appropriate channels, we would establish our weekly goals, identify strengths and weaknesses, delegate tasks, request help, or address concerns in regard to our collective progress within the project.

All required documentation (including the project proposal, the language reference manual, and other submissions) was also shared and disseminated within the appropriate Slack channels, as well as notes from TA feedback and potential pivots to be made on our language.

4.1.2 Specification

Identifying the route that we wanted to take when creating a language was difficult at first, and we entertained a number of different possible paths in our initial brainstorming sessions. We wanted an idea with enough content to be a viable class project while not being overly ambitious within the scope of creating a programming language.

We identified areas within languages that are already created that we agreed with, and other areas that we saw ways to improve. After identifying a number of these traits within both C and Python, we applied these concepts to the notion of creating a language centered around the creation of simple, user-designed timers.

With this, we hoped that we could achieve a language that had practical applications while catering to a clearly defined use case. We would learn throughout the project, however, that our expectations for the overall functionality of this would need to be lowered to fit more reasonable expectations for what could actually be delivered within the timeframe of the class.

4.1.3 Development

We used the homework submission dates as benchmarks to implement the respective areas of our language. Initially, we met frequently to specify broad ideas, such as the architecture of the language and how exactly we would achieve the ultimate goal of the language, which required

simplicity in the lexicon and efficiency when running in the background. We worked off of a team model that involved executing tasks that aligned with our respective roles to avoid cluttering each other's work spaces, but later pivoted to a more overlapping and dynamic model.

After receiving initial feedback from our TA, we scaled back the functionality of our language and made important decisions on the runtime capabilities and structure of our language to align the level of ambition with the project to what could be feasibly completed within the timeframe of the class. The grammar of the language consistently changed after submission of our LRM, as we iterated through our potential language structure repeatedly as we revisited the decisions mentioned above. These decisions affected the completion of our scanner, parser, and abstract syntax tree as we adapted to many roadblocks in the development process.

4.1.4 Testing

Our testing process came much later in the semester, as the initial focus included patching the bugs that we were finding in many sections of our code. In the test cases, we include checks for about 80 different cases. By implementing these later, we were able to make more catered and specific test scenarios that could cover all of our known features within the language with the most updated code. Our test suite aimed to be both broad and specific and tested in line with other features of the language.

4.2 Project Timeline

4.2.1 Planned Timeline

This was our anticipated timeline when we began the project (dates adjusted to account for semester schedule changes). This was vaguely established at the very beginning of development and does not included the details that were discovered throughout the process of building the language, which led to many pivots and design changes:

Dates	Goals
January 11-22	Organize team and plan, brainstorm/select language idea
February 3	Submit proposal
February 24	Submit LRM
February 15-19	Organize LRM, expand sections, build parser

February 22-26	Finalize LRM, Submit LRM and parser
March 1-19	Receive feedback on LRM, review scanner/parser/AST, progress on language
March 24	Submit “Hello World”
March 25-April 12	Expand test suites, semantic checking
April 12-23	Testing, debugging, documentation
April 25	Presentation, submit Final Report

4.2.2 Project Log

Our actual timeline for the project is reflected below. More detail is included to expand further upon the milestones, goals, and actions reflected in the initial Planned Timeline.

Dates	Goals
January 11-15	Team Organization, Slack setup, initial brainstorming for language ideas
January 18-22	Continued brainstorming for language idea, sharpened vision, selected final idea
January 25-29	Expanded upon final idea, incorporated elements into proposal draft
February 1-5	Organized language proposal, submitted proposal for review
February 8-12	Received feedback on proposal, considered language derivations from feedback
February 15-19	Organized LRM, expanded sections within LRM and language features, built parser
February 22-26	Finalized LRM, Submitted LRM and parser
March 1-5	Received feedback on LRM, reviewed parser
March 8-19	Work with LLVM, runtime, AST
March 22-26	Submitted “Hello World”
March 29-April 2	Code review

April 5-16	Bug patching, code refinement
April 19-23	Documentation, finalizing code and presentation
April 25	Presented language, submitted Final Report

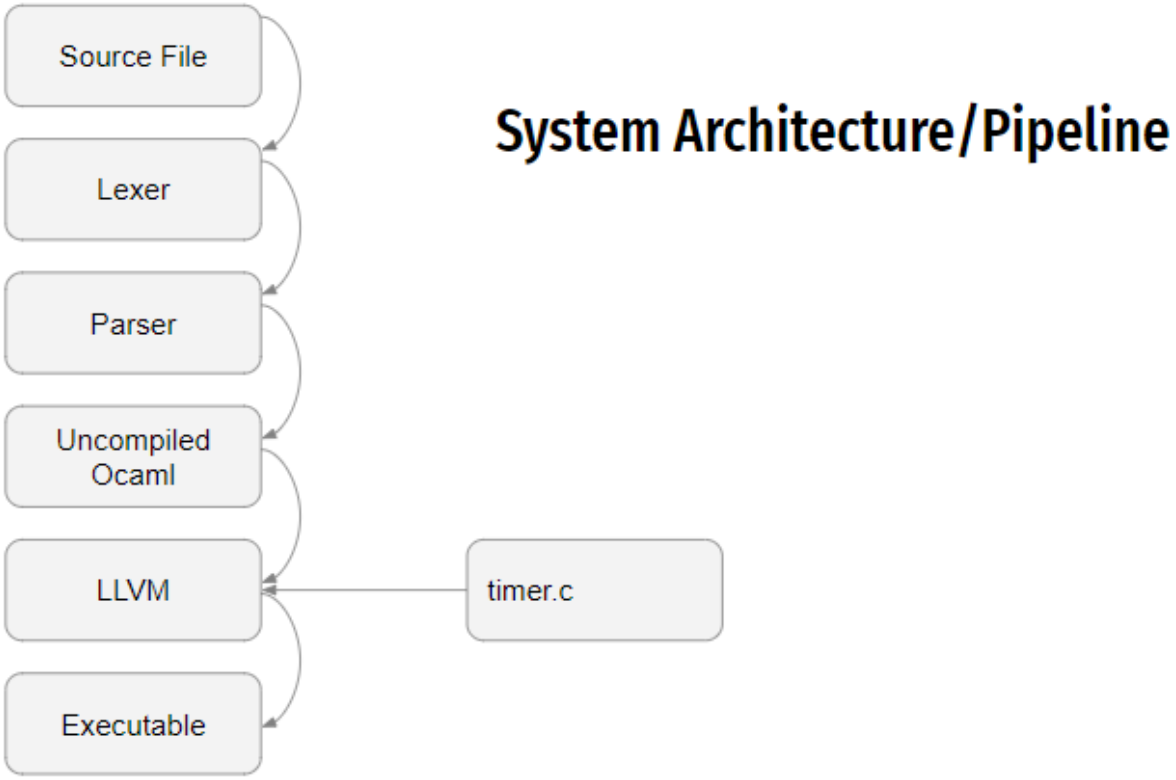
4.3 Roles and Responsibilities

Team Member	Role
Jeff Kline	Language Guru
Faisal Rahman	Systems Architect
Daniel Rindone	Manager
Eric Webb	Tester

Team responsibilities, as mentioned earlier, were initially assigned based on the scope of the role to avoid work conflict and organize flow of the project. Later in the language development process, though, these roles became more dynamic and responsibilities overlapped evenly among the team.

5.0 Architectural Design

5.1 Diagram



6.0 Test Plan

6.1 Source Programs

6.1.1 test-string.mc

```
int main()
{
str f;
f = "Hello";
 prints("Hello!");
 prints(f);
```

```
 f = "World!";
 prints(f);
}
```

6.1.2 test-timer.mc

```
int main()
{
   timer_init(1);
   start_timer(2000);
   timer_destroy(1);
}
```

6.1.3 fail-int-assign1.mc

```
int main()
{
  int i = "test";
}
```

6.2 Testing Plan

6.2.1 Tests

Tests used were the entirety of the microc test suite, as we were encouraged to take them and conform them to our needs. Building on that, we added language specific tests listed below:

- Fail-comment1-2
    - Check for whitespace errors.
- Fail-float-assign1-2
    - Check for illegally assigned floats.
- Fail-int-assign1-4
    - Check for illegally assigned integers.
- Fail-string-assign1-4
    - Check for illegally assigned strings.
- Fail-timer-assign1-8
    - Ensure appropriate arguments are passed to a timer being instantiated.
- Test-timer.mc
    - Tests timer function.
- Test-string.mc

- Tests if string will print.

All tests are listed in appendix 8.2

6.2.2 Explanation

For testing, we started out small checking variable declarations as well as any escape characters in whitespace (EX: fail-comment1.err, fail-int-assign1.mc). Once established we could ensure appropriate declarations or find illegal character escapes, we moved onto checking arguments for instantiating a timer.

6.2.3 Automation

As mentioned in lecture, the best approach for building a reliable test suite was to build off of microc's, as our language is closely related. To run the tests automatically, one can simply run "make all". When invoked, the Makefile will run testall.sh and run through all tests written within the "tests" folder.

7.0 Lessons Learned

7.1 Important Learning Points

7.1.1 Jeff

I was heavily invested in the original idea of creating a simple, readable code that was heavily adaptable for general purposes. I wanted to create a single tool for the express purpose of timers, and build it on the backbone of that functionality. I had a clear vision for how it could work, but when it came down to technical implementation, it became more of a headache trying to build everything from scratch, and we spent too much time on technical details that kept us from taking incremental steps along the way. We ended up pivoting back to microc and as a result, were not able to complete a language that had the level of functionality we desired, even with c-syntax. The lesson learned from that is don't be attached to your ideas; let yourself be flexible and remember that this is a learning experience, and not a product development course.

7.1.2 Faisal

Echoing the thoughts of most students, it is advantageous to start early, stay ahead of the game, and communicate. Programming as a team is helpful, and it's imperative to always ask for help,

as it is easy to get behind. Keep your aspirations in check and don't pivot too late in the semester or else things can get hard fast.

7.1.3 Daniel

All points made early on in the semester about planning, scheduling, communication and establishing concrete milestones ring true once the end of the semester comes, and it comes fast. Time management is crucial, start small and grow from there. The concepts come up rapidly and it can be difficult to comprehend all these new concepts while at the same time tackling the learning curve of a functional language like Ocaml in the context of creating your own language.

7.1.4 Eric

Teammates need to be communicative to one another, as well as honest. We started with high expectations which slowly fell throughout the design and building of the project. As these expectations dropped, communication became more key as each member's work was affected by one another. Future groups should keep eachother aware of these changes and plan accordingly.

8.0 Appendix and Code Listings

8.1 Dockerfile

```
# Based on 20.04 LTS
FROM ubuntu:focal


RUN apt-get -yq update && \
    apt-get -y upgrade && \
    apt-get -yq --no-install-suggests --no-install-recommends install \
    ocaml \
    menhir \
    llvm-10.0 \
    llvm-10.0-dev \
    m4 \
    git \
    aspcud \
    ca-certificates \
    python2.7 \
    pkg-config \
    cmake \
    opam
```

```
RUN ln -s /usr/bin/lli-10.0 /usr/bin/lli
RUN ln -s /usr/bin/llc-10.0 /usr/bin/llc


RUN opam init
RUN opam install \
    llvm.10.0.0 \
    ocamlfind


WORKDIR /root


ENTRYPOINT ["opam", "config", "exec", "--"]


CMD ["bash"]
```

8.2 Makefile

```
# "make test" Compiles everything and runs the regression tests


.PHONY : test
test : all testall.sh
	./testall.sh


# "make all" builds the executable as well as the "printbig" library
designed
# to test linking external code


.PHONY : all
all : timrs.native printbig.o timer.o


# "make microc.native" compiles the compiler
#
```

```
# The _tags file controls the operation of ocamlbuild, e.g., by including
# packages, enabling warnings
#
# See https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc


timrs.native :
      opam config exec -- \
      ocamlbuild -use-ocamlfind timrs.native


# "make clean" removes all generated files


.PHONY : clean
clean :
      ocamlbuild -clean
      rm -rf testall.log ocamlllvm *.diff


# Testing the "printbig" example


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


timer : timer.c
      cc -o timer -DBUILD_TEST timer.c -lm -pthread




# Building the tarball


TESTS = \
  add1 arith1 arith2 arith3 fib float1 float2 float3 for1 for2 func1 \
  func2 func3 func4 func5 func6 func7 func8 func9 gcd2 gcd global1 \
  global2 global3 hello if1 if2 if3 if4 if5 if6 local1 local2 ops1 \
  ops2 printbig var1 var2 while1 while2 timer
```

```
FAILS = \
  assign1 assign2 assign3 dead1 dead2 expr1 expr2 expr3 float1 float2 \
  for1 for2 for3 for4 for5 func1 func2 func3 func4 func5 func6 func7 \
  func8 func9 global1 global2 if1 if2 if3 nomain printbig printb print \
  return1 return2 while1 while2


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


TARFILES = ast.ml sast.ml codegen.ml Makefile _tags timrs.ml timrsparse.mly
\
      README scanner.mll semant.ml testall.sh \
      printbig.c timer.c arcade-font.pbm font2c \
      Dockerfile \
      $(TESTFILES:%=tests/%)


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

8.3 _tags

```
# Include the llvm and llvm.analysis packages while compiling
true: package(llvm), package(llvm.analysis)


# Enable almost all compiler warnings
true : warn(+a-4)


# Instruct ocamlbuild to ignore the "printbig.o" file when it's building
"printbig.o": not_hygienic
"timer.o": not_hygienic
```

8.4 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 typ = Int | Bool | Float | Str | Void


type bind = typ * string


type expr =
    Literal of int
  | Fliteral of string
  | BoolLit of bool
  | StrLit of string
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr


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


type func_decl = {
  typ : typ;
```

```
  fname : string;
  formals : bind list;
  locals : bind list;
  body : stmt list;
}


type program = bind list * func_decl list


(* Pretty-printing functions *)


let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"


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


let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Fliteral(l) -> l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | Id(s) -> s
  | StrLit(s) -> s
  | Binop(e1, o, e2) ->
    string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
```

```
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""


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


let string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Void -> "void"
  | Str -> "str"


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


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


let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
```

```
  String.concat "\n" (List.map string_of_fdecl funcs)
```

8.5 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
open Sast


module StringMap = Map.Make(String)


(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =
  let context    = L.global_context () in


  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "MicroC" in


  (* Get types from the context *)
  let i32_t      = L.i32_type    context
  and i8_t       = L.i8_type     context
  and i1_t       = L.i1_type     context
```

```
and float_t    = L.double_type context
and void_t     = L.void_type   context
and str_t      = L.pointer_type (L.i8_type context) in


(* Return the LLVM type for a MicroC type *)
let ltype_of_typ = function
    A.Int   -> i32_t
  | A.Bool  -> i1_t
  | A.Float -> float_t
  | A.Str   -> str_t
  | A.Void  -> void_t
in


(* Create a map of global variables after creating each *)
let global_vars : L.llvalue StringMap.t =
  let global_var m (t, n) =
    let init = match t with
        A.Float -> L.const_float (ltype_of_typ t) 0.0
      | _ -> L.const_int (ltype_of_typ t) 0
    in StringMap.add n (L.define_global n init the_module) m in
  List.fold_left global_var StringMap.empty globals in


let printf_t : L.lltype =
  L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func : L.llvalue =
  L.declare_function "printf" printf_t the_module in


let printbig_t : L.lltype =
  L.function_type i32_t [| i32_t |] in
let printbig_func : L.llvalue =
  L.declare_function "printbig" printbig_t the_module in


let timer_init_t : L.lltype =
  L.function_type i32_t [|  i32_t |] in
let timer_init_func : L.llvalue =
  L.declare_function "timer_init" timer_init_t the_module in
```

```
  let start_timer_t : L.lltype =
    L.function_type i32_t [|  i32_t |] in
  let start_timer_func : L.llvalue =
    L.declare_function "start_timer" start_timer_t the_module in


  let timer_destroy_t : L.lltype =
    L.function_type i32_t [|  i32_t |] in
  let timer_destroy_func : L.llvalue =
    L.declare_function "timer_destroy" timer_destroy_t the_module in


  (* Define each function (arguments and return type) so we can
     call it even before we've created its body *)
  let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
    let function_decl m fdecl =
      let name = fdecl.sfname
      and formal_types =
        Array.of_list (List.map (fun (t,_) -> ltype_of_typ t)
fdecl.sformals)
      in let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types
in
      StringMap.add name (L.define_function name ftype the_module, fdecl) m
in
    List.fold_left function_decl StringMap.empty functions in


  (* Fill in the body of the given function *)
  let build_function_body fdecl =
    let (the_function, _) = StringMap.find fdecl.sfname function_decls in
    let builder = L.builder_at_end context (L.entry_block the_function) in


    let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
    and str_format_str = L.build_global_stringptr "%s\n" "fmt" builder
    and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder in


    (* Construct the function's "locals": formal arguments and locally
       declared variables.  Allocate each on the stack, initialize their
       value, if appropriate, and remember their values in the "locals" map
*)
    let local_vars =
```

```
      let add_formal m (t, n) p =
        L.set_value_name n p;
        let local = L.build_alloca (ltype_of_typ t) n builder in
        ignore (L.build_store p local builder);
        StringMap.add n local m


      (* Allocate space for any locally declared variables and add the
       * resulting registers to our map *)
      and add_local m (t, n) =
        let local_var = L.build_alloca (ltype_of_typ t) n builder
        in StringMap.add n local_var m
      in


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


    (* Return the value for a variable or formal argument.
       Check local names first, then global names *)
    let lookup n = try StringMap.find n local_vars
      with Not_found -> StringMap.find n global_vars
    in


    (* Construct code for an expression; return its value *)
    let rec expr builder ((_, e) : sexpr) = match e with
        SLiteral i  -> L.const_int i32_t i
      | SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)
      | SFliteral l -> L.const_float_of_string float_t l
      | SStrLit s   -> L.build_global_stringptr s "str" builder
      | SNoexpr     -> L.const_int i32_t 0
      | SId s       -> L.build_load (lookup s) s builder
      | SAssign (s, e) -> let e' = expr builder e in
        ignore(L.build_store e' (lookup s) builder); e'
      | SBinop ((A.Float,_ ) as e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2 in
        (match op with
```

```
          A.Add     -> L.build_fadd
        | A.Sub     -> L.build_fsub
        | A.Mult    -> L.build_fmul
        | A.Div     -> L.build_fdiv
        | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
        | A.Neq     -> L.build_fcmp L.Fcmp.One
        | A.Less    -> L.build_fcmp L.Fcmp.Olt
        | A.Leq     -> L.build_fcmp L.Fcmp.Ole
        | A.Greater -> L.build_fcmp L.Fcmp.Ogt
        | A.Geq     -> L.build_fcmp L.Fcmp.Oge
        | A.And | A.Or ->
          raise (Failure "internal error: semant should have rejected
and/or on float")
        ) e1' e2' "tmp" builder
      | SBinop (e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2 in
        (match op with
          A.Add     -> L.build_add
        | A.Sub     -> L.build_sub
        | A.Mult    -> L.build_mul
        | A.Div     -> L.build_sdiv
        | A.And     -> L.build_and
        | A.Or      -> L.build_or
        | A.Equal   -> L.build_icmp L.Icmp.Eq
        | A.Neq     -> L.build_icmp L.Icmp.Ne
        | A.Less    -> L.build_icmp L.Icmp.Slt
        | A.Leq     -> L.build_icmp L.Icmp.Sle
        | A.Greater -> L.build_icmp L.Icmp.Sgt
        | A.Geq     -> L.build_icmp L.Icmp.Sge
        ) e1' e2' "tmp" builder
      | SUnop(op, ((t, _) as e)) ->
        let e' = expr builder e in
        (match op with
          A.Neg when t = A.Float -> L.build_fneg
        | A.Neg                  -> L.build_neg
        | A.Not                  -> L.build_not) e' "tmp" builder
      | SCall ("print", [e]) | SCall ("printb", [e]) ->
        L.build_call printf_func [| int_format_str ; (expr builder e) |]
"printf" builder
      | SCall ("prints", [e]) ->
        L.build_call printf_func [| str_format_str ; (expr builder e) |]
```

```
          "printf" builder
      | SCall ("printbig", [e]) ->
        L.build_call printbig_func [| (expr builder e) |] "printbig"
builder


      | SCall ("printf", [e]) ->
        L.build_call printf_func [| float_format_str ; (expr builder e) |]
"printf" builder


      | SCall ("timer_init", [e]) ->
        L.build_call timer_init_func [| (expr builder e)|] "timer_init"
builder


      | SCall ("start_timer", [e]) ->
        L.build_call start_timer_func [| (expr builder e)|] "start_timer"
builder


      | SCall ("timer_destroy", [e]) ->
        L.build_call timer_destroy_func [| (expr builder e)|]
"timer_destroy" builder


      | SCall (f, args) ->
        let (fdef, fdecl) = StringMap.find f function_decls in
        let llargs = List.rev (List.map (expr builder) (List.rev args)) in
        let result = (match fdecl.styp with
              A.Void -> ""
            | _ -> f ^ "_result") in
        L.build_call fdef (Array.of_list llargs) result builder
    in


    (* LLVM insists each basic block end with exactly one "terminator"
       instruction that transfers control.  This function runs "instr
builder"
       if the current block does not already have a terminator.  Used,
       e.g., to handle the "fall off the end of the function" case. *)
    let add_terminal builder instr =
      match L.block_terminator (L.insertion_block builder) with
```

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


  (* Build the code for the given statement; return the builder for
     the statement's successor (i.e., the next instruction will be built
     after the one generated by this call) *)


  let rec stmt builder = function
      SBlock sl -> List.fold_left stmt builder sl
    | SExpr e -> ignore(expr builder e); builder
    | SReturn e -> ignore(match fdecl.styp with
        (* Special "return nothing" instr *)
          A.Void -> L.build_ret_void builder
        (* Build return statement *)
        | _ -> L.build_ret (expr builder e) builder );
      builder
    | SIf (predicate, then_stmt, else_stmt) ->
      let bool_val = expr builder predicate in
      let merge_bb = L.append_block context "merge" the_function in
      let build_br_merge = L.build_br merge_bb in (* partial function *)


      let then_bb = L.append_block context "then" the_function in
      add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
        build_br_merge;


      let else_bb = L.append_block context "else" the_function in
      add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
        build_br_merge;


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


    | SWhile (predicate, body) ->
      let pred_bb = L.append_block context "while" the_function in
      ignore(L.build_br pred_bb builder);
```

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


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


        let merge_bb = L.append_block context "merge" the_function in
        ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
        L.builder_at_end context merge_bb


      (* Implement for loops as while loops *)
      | SFor (e1, e2, e3, body) -> stmt builder
                                    ( SBlock [SExpr e1 ; SWhile (e2,
SBlock [body ; SExpr e3]) ] )
    in


    (* Build the code for each statement in the function *)
    let builder = stmt builder (SBlock fdecl.sbody) in


    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.styp with
          A.Void -> L.build_ret_void
        | A.Float -> L.build_ret (L.const_float float_t 0.0)
        | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in


  List.iter build_function_body functions;
  the_module
```

8.6 font2c

```
#!/bin/bash
```

```
# Assumes the .pbm file was created from GIMP using the "Raw" output mode


# Generates C-like source code for a font that's easily added to a .c file


tail -n +4 arcade-font.pbm | od --output-duplicates --width=8 --format=x1
--address-radix=n | sed "s/^ /0x/
s/ /, 0x/g
s/$/,/"
```

8.7 printbig.c

```
/*
 *  A function illustrating how to link C code to code generated from LLVM
 */


#include <stdio.h>


/*
 * Font information: one byte per row, 8 rows per character
 * In order, space, 0-9, A-Z
 */
static const char font[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x1c, 0x3e, 0x61, 0x41, 0x43, 0x3e, 0x1c, 0x00,
  0x00, 0x40, 0x42, 0x7f, 0x7f, 0x40, 0x40, 0x00,
  0x62, 0x73, 0x79, 0x59, 0x5d, 0x4f, 0x46, 0x00,
  0x20, 0x61, 0x49, 0x4d, 0x4f, 0x7b, 0x31, 0x00,
  0x18, 0x1c, 0x16, 0x13, 0x7f, 0x7f, 0x10, 0x00,
  0x27, 0x67, 0x45, 0x45, 0x45, 0x7d, 0x38, 0x00,
  0x3c, 0x7e, 0x4b, 0x49, 0x49, 0x79, 0x30, 0x00,
  0x03, 0x03, 0x71, 0x79, 0x0d, 0x07, 0x03, 0x00,
  0x36, 0x4f, 0x4d, 0x59, 0x59, 0x76, 0x30, 0x00,
  0x06, 0x4f, 0x49, 0x49, 0x69, 0x3f, 0x1e, 0x00,
  0x7c, 0x7e, 0x13, 0x11, 0x13, 0x7e, 0x7c, 0x00,
  0x7f, 0x7f, 0x49, 0x49, 0x49, 0x7f, 0x36, 0x00,
  0x1c, 0x3e, 0x63, 0x41, 0x41, 0x63, 0x22, 0x00,
  0x7f, 0x7f, 0x41, 0x41, 0x63, 0x3e, 0x1c, 0x00,
```

```
  0x00, 0x7f, 0x7f, 0x49, 0x49, 0x49, 0x41, 0x00,
  0x7f, 0x7f, 0x09, 0x09, 0x09, 0x09, 0x01, 0x00,
  0x1c, 0x3e, 0x63, 0x41, 0x49, 0x79, 0x79, 0x00,
  0x7f, 0x7f, 0x08, 0x08, 0x08, 0x7f, 0x7f, 0x00,
  0x00, 0x41, 0x41, 0x7f, 0x7f, 0x41, 0x41, 0x00,
  0x20, 0x60, 0x40, 0x40, 0x40, 0x7f, 0x3f, 0x00,
  0x7f, 0x7f, 0x18, 0x3c, 0x76, 0x63, 0x41, 0x00,
  0x00, 0x7f, 0x7f, 0x40, 0x40, 0x40, 0x40, 0x00,
  0x7f, 0x7f, 0x0e, 0x1c, 0x0e, 0x7f, 0x7f, 0x00,
  0x7f, 0x7f, 0x0e, 0x1c, 0x38, 0x7f, 0x7f, 0x00,
  0x3e, 0x7f, 0x41, 0x41, 0x41, 0x7f, 0x3e, 0x00,
  0x7f, 0x7f, 0x11, 0x11, 0x11, 0x1f, 0x0e, 0x00,
  0x3e, 0x7f, 0x41, 0x51, 0x71, 0x3f, 0x5e, 0x00,
  0x7f, 0x7f, 0x11, 0x31, 0x79, 0x6f, 0x4e, 0x00,
  0x26, 0x6f, 0x49, 0x49, 0x4b, 0x7a, 0x30, 0x00,
  0x00, 0x01, 0x01, 0x7f, 0x7f, 0x01, 0x01, 0x00,
  0x3f, 0x7f, 0x40, 0x40, 0x40, 0x7f, 0x3f, 0x00,
  0x0f, 0x1f, 0x38, 0x70, 0x38, 0x1f, 0x0f, 0x00,
  0x1f, 0x7f, 0x38, 0x1c, 0x38, 0x7f, 0x1f, 0x00,
  0x63, 0x77, 0x3e, 0x1c, 0x3e, 0x77, 0x63, 0x00,
  0x00, 0x03, 0x0f, 0x78, 0x78, 0x0f, 0x03, 0x00,
  0x61, 0x71, 0x79, 0x5d, 0x4f, 0x47, 0x43, 0x00
};


void printbig(int c)
{
  int index = 0;
  int col, data;
  if (c >= '0' && c <= '9') index = 8 + (c - '0') * 8;
  else if (c >= 'A' && c <= 'Z') index = 88 + (c - 'A') * 8;
  do {
    data = font[index++];
    for (col = 0 ; col < 8 ; data <<= 1, col++) {
      char d = data & 0x80 ? 'X' : ' ';
      putchar(d); putchar(d);
    }
    putchar('\n');
  } while (index & 0x7);
}
```

```
#ifdef BUILD_TEST
int main()
{
  char s[] = "HELLO WORLD09AZ";
  char *c;
  for ( c = s ; *c ; c++) printbig(*c);
}
#endif
```

8.8 sast.ml

```
(* Semantically-checked Abstract Syntax Tree and functions for printing it
*)
open Ast
type sexpr = typ * sx
and sx =
    SLiteral of int
  | SFliteral of string
  | SBoolLit of bool
  | SStrLit of string
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SAssign of string * sexpr
  | SCall of string * sexpr list
  | SNoexpr
type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt
type sfunc_decl = {
    styp : typ;
    sfname : string;
    sformals : bind list;
    slocals : bind list;
    sbody : sstmt list;
  }
```

```
type sprogram = bind list * sfunc_decl list
(* Pretty-printing functions *)
let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
    SLiteral(l) -> string_of_int l
  | SBoolLit(true) -> "true"
  | SBoolLit(false) -> "false"
  | SStrLit(s) -> s
  | SFliteral(l) -> l
  | SId(s) -> s
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2

  | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
  | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
  | SCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
  | SNoexpr -> ""
                          ) ^ ")"
let rec string_of_sstmt = function
    SBlock(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
  | SIf(e, s, SBlock([])) ->
      "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
      "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^

      string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt
s
let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sformals) ^

  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.slocals) ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"
let string_of_sprogram (vars, funcs) =
```

```
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs)
```

8.9 scanner.mll

```
(* Ocamllex scanner for Timrs *)


{ open Timrsparse }


let digit = ['0' - '9']
let digits = digit+
let strlit = ('"' ([^'"']* as s) '"')


rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
```

```
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "bool"   { BOOL }
| "str"    { STR }
| "float"  { FLOAT }
| "void"   { VOID }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
| strlit   { STRLIT(s) }
| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
| ['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
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

8.10 semant.ml

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


open Ast
open Sast


module StringMap = Map.Make(String)


(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)


let check (globals, functions) =
```

```
(* Verify a list of bindings has no void types or duplicate names *)
let check_binds (kind : string) (binds : bind list) =
  List.iter (function
        (Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
      | _ -> ()) binds;
  let rec dups = function
      [] -> ()
    |     ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
      raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
    | _ :: t -> dups t
  in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
in


(**** Check global variables ****)


check_binds "global" globals;


(**** Check functions ****)


(* Collect function declarations for built-in functions: no bodies *)
let built_in_decls =
  let add_bind map (name, ty) = StringMap.add name {
      typ = Void;
      fname = name;
      formals = [(ty, "x")];
      locals = []; body = [] } map
  in List.fold_left add_bind StringMap.empty [ ("print", Int);
                                               ("printb", Bool);
                                               ("printf", Float);
                                               ("printbig", Int);
                                               ("timer_init", Int);
                                               ("start_timer", Int);
                                               ("timer_destroy", Int);
                                               ("prints", Str) ]


in
```

```
  (* Add function name to symbol table *)
  let add_func map fd =
    let built_in_err = "function " ^ fd.fname ^ " may not be defined"
    and dup_err = "duplicate function " ^ fd.fname
    and make_err er = raise (Failure er)
    and n = fd.fname (* Name of the function *)
    in match fd with (* No duplicate functions or redefinitions of
built-ins *)
        _ when StringMap.mem n built_in_decls -> make_err built_in_err
      | _ when StringMap.mem n map -> make_err dup_err
      | _ ->  StringMap.add n fd map
  in


  (* Collect all function names into one symbol table *)
  let function_decls = List.fold_left add_func built_in_decls functions
  in


  (* Return a function from our symbol table *)
  let find_func s =
    try StringMap.find s function_decls
    with Not_found -> raise (Failure ("unrecognized function " ^ s))
  in


  let _ = find_func "main" in (* Ensure "main" is defined *)


  let check_function func =
    (* Make sure no formals or locals are void or duplicates *)
    check_binds "formal" func.formals;
    check_binds "local" func.locals;


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

```
    in


    (* Build local symbol table of variables for this function *)
    let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty
m)
        StringMap.empty (globals @ func.formals @ func.locals )
    in


    (* Return a variable from our local symbol table *)
    let type_of_identifier s =
      try StringMap.find s symbols
      with Not_found -> raise (Failure ("undeclared identifier " ^ s))
    in


    (* Return a semantically-checked expression, i.e., with a type *)
    let rec expr = function
        Literal  l -> (Int, SLiteral l)
      | Fliteral l -> (Float, SFliteral l)
      | StrLit l -> (Str, SStrLit l)
      | BoolLit l  -> (Bool, SBoolLit l)
      | Noexpr     -> (Void, SNoexpr)
      | Id s       -> (type_of_identifier s, SId s)
      | Assign(var, e) as ex ->
        let lt = type_of_identifier var
        and (rt, e') = expr e in
        let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
                  string_of_typ rt ^ " in " ^ string_of_expr ex
        in (check_assign lt rt err, SAssign(var, (rt, e')))
      | Unop(op, e) as ex ->
        let (t, e') = expr e in
        let ty = match op with
            Neg when t = Int || t = Float -> t
          | Not when t = Bool -> Bool
          | _ -> raise (Failure ("illegal unary operator " ^
                                 string_of_uop op ^ string_of_typ t ^
                                 " in " ^ string_of_expr ex))
        in (ty, SUnop(op, (t, e')))
      | Binop(e1, op, e2) as e ->
        let (t1, e1') = expr e1
        and (t2, e2') = expr e2 in
```

```
        (* All binary operators require operands of the same type *)
        let same = t1 = t2 in
        (* Determine expression type based on operator and operand types *)
        let ty = match op with
            Add | Sub | Mult | Div when same && t1 = Int   -> Int
          | Add | Sub | Mult | Div when same && t1 = Float -> Float
          | Equal | Neq            when same               -> Bool
          | Less | Leq | Greater | Geq
            when same && (t1 = Int || t1 = Float) -> Bool
          | And | Or when same && t1 = Bool -> Bool
          | _ -> raise (
              Failure ("illegal binary operator " ^
                       string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                       string_of_typ t2 ^ " in " ^ string_of_expr e))
        in (ty, SBinop((t1, e1'), op, (t2, e2')))
      | Call(fname, args) as call ->
        let fd = find_func fname in
        let param_length = List.length fd.formals in
        if List.length args != param_length then
          raise (Failure ("expecting " ^ string_of_int param_length ^
                          " arguments in " ^ string_of_expr call))
        else let check_call (ft, _) e =
               let (et, e') = expr e in
               let err = "illegal argument found " ^ string_of_typ et ^
                         " expected " ^ string_of_typ ft ^ " in " ^
string_of_expr e
               in (check_assign ft et err, e')
          in
          let args' = List.map2 check_call fd.formals args
          in (fd.typ, SCall(fname, args'))
    in


    let check_bool_expr e =
      let (t', e') = expr e
      and err = "expected Boolean expression in " ^ string_of_expr e
      in if t' != Bool then raise (Failure err) else (t', e')
    in


    (* Return a semantically-checked statement i.e. containing sexprs *)
    let rec check_stmt = function
        Expr e -> SExpr (expr e)
```

```
      | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt
b2)
      | For(e1, e2, e3, st) ->
        SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
      | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
      | Return e -> let (t, e') = expr e in
        if t = func.typ then SReturn (t, e')
        else raise (
            Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                     string_of_typ func.typ ^ " in " ^ string_of_expr e))


      (* A block is correct if each statement is correct and nothing
                  follows any Return statement.  Nested blocks are
flattened. *)
      | Block sl ->
        let rec check_stmt_list = function
            [Return _ as s] -> [check_stmt s]
          | Return _ :: _   -> raise (Failure "nothing may follow a
return")
          | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten blocks
*)
          | s :: ss         -> check_stmt s :: check_stmt_list ss
          | []              -> []
        in SBlock(check_stmt_list sl)


    in (* body of check_function *)
    { styp = func.typ;
      sfname = func.fname;
      sformals = func.formals;
      slocals  = func.locals;
      sbody = match check_stmt (Block func.body) with
          SBlock(sl) -> sl
        | _ -> raise (Failure ("internal error: block didn't become a
block?"))
    }
  in (globals, List.map check_function functions)
```

8.11 testall.sh

```
#!/bin/sh


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


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


# Path to the LLVM compiler
LLC="llc"


# Path to the C compiler
CC="cc"


# Path to the microc compiler.  Usually "./microc.native"
# Try "_build/microc.native" if ocamlbuild was unable to create a symbolic
link.
TIMRS="./timrs.native"
#MICROC="_build/microc.native"


# Set time limit for all operations
ulimit -t 30


globallog=testall.log
rm -f $globallog
error=0
globalerror=0


keep=0
```

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


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


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


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


# RunFail <args>
```

```
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
      SignalError "failed: $* did not report an error"
      return 1
    }
    return 0
}


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


    echo -n "$basename..."


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


    generatedfiles=""


    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s
${basename}.exe ${basename}.out" &&
    Run "$TIMRS" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s"
&&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "printbig.o" "timer.o"
"-lm" "-pthread" &&
    Run "./${basename}.exe" > "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff


    # Report the status and clean up the generated files
```

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


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


    echo -n "$basename..."


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


    generatedfiles=""


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


    # Report the status and clean up the generated files


    if [ $error -eq 0 ] ; then
      if [ $keep -eq 0 ] ; then
          rm -f $generatedfiles
      fi
```

```
      echo "OK"
      echo "###### SUCCESS" 1>&2
    else
      echo "###### FAILED" 1>&2
      globalerror=$error
    fi
}


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


shift `expr $OPTIND - 1`


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


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


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

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


for file in $files
do
    case $file in
      *test-*)
          Check $file 2>> $globallog
          ;;
      *fail-*)
          CheckFail $file 2>> $globallog
          ;;
      *)
          echo "unknown file type $file"
          globalerror=1
          ;;
    esac
done


exit $globalerror
```

8.12 timer.c

```
#include <poll.h>
#include <pthread.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/timerfd.h>
#include <unistd.h>


#ifndef TIME_H
#define TIME_H
```

```
typedef void (*time_h)(size_t t_id, void* data);


int timer_init();
void start_timer(int interval);
void stop_timer(size_t t_id);
void timer_destroy();


#endif


#define MAX_TIMERS 500


struct timer_node {
    int fd;
    struct timer_node* n;
    time_h callback;
    void* data;
    unsigned int interval;
};


static void* timer_thread();
static pthread_t thread_id;
static struct timer_node* head = NULL;


int timer_init(int x)
{
    if (pthread_create(&thread_id, NULL, timer_thread, NULL)) {
        return 0;
    }
    return 1;
}


void start_timer(int interval)
{
    struct timer_node* node = NULL;
    struct itimerspec value;
    uint64_t buf;
```

```
    node = (struct timer_node*)malloc(sizeof(struct timer_node));


    if (node == NULL) {
//          return 0;
    }
    // node->callback = handler;
    // node->data = data;
    node->interval = interval;


    node->fd = timerfd_create(CLOCK_REALTIME, 0);


    if (node->fd == -1) {
        free(node);
//          return 0;
    }


    value.it_value.tv_sec = interval / 1000;
    value.it_value.tv_nsec = (interval % 1000) * 1000000;
    value.it_interval.tv_sec = 0;
    value.it_interval.tv_nsec = 0;
    timerfd_settime(node->fd, 0, &value, NULL);
    node->n = head;
    head = node;
    sleep((interval / 1000) + 1);
    // return (size_t)node;
}


void stop_timer(size_t timer_id)
{
    struct timer_node *t, *n;
    t = NULL;
    n = (struct timer_node*)timer_id;


    if (n != NULL) {
        close(n->fd);
```

```
        if (n == head) {
            head = head->n;
        } else {
            t = head;
            while (t && t->n != n) {
                t = t->n;
            }
            if (t) {
                t->n = t->n->n;
            }
        }
    }
    if (n) {
        free(n);
    }
}


void timer_destroy()
{
    while (head) {
        stop_timer((size_t)head);
    }
    pthread_cancel(thread_id);
    pthread_join(thread_id, NULL);
}


struct timer_node* get_timer(int fd)
{
    struct timer_node* t = head;


    while (t) {
        if (t->fd == fd) {
            return t;
        }
        t = t->n;
    }
    return NULL;
}
```

```
void* timer_thread()
{
    struct pollfd pfd[MAX_TIMERS] = { { 0 } };
    int count = 0;
    struct timer_node* t = NULL;
    int r_fds = 0;
    uint64_t e;


    while (1) {
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
        pthread_testcancel();
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);


        count = 0;
        t = head;


        memset(pfd, 0, sizeof(struct pollfd) * MAX_TIMERS);
        while (t) {
            pfd[count].fd = t->fd;
            pfd[count].events = POLLIN;
            count++;
            t = t->n;
        }


        r_fds = poll(pfd, count, 100);
        if (r_fds <= 0) {
            continue;
        }


        for (int i = 0; i < count; i++) {
            if (pfd[i].revents & POLLIN) {
                int s = read(pfd[i].fd, &e, sizeof(uint64_t));
                if (s == sizeof(uint64_t)) {
                    t = get_timer(pfd[i].fd);
                    if (t && t->callback) {
                        t->callback((size_t)t, t->data);
                    }
                    printf("Timer ran for %d\n", t->interval);
```

```
                }
            }
        }
    }
    return NULL;
}


#ifdef BUILD_TEST


int main()
{


    timer_init(1);
    start_timer(200);
    timer_destroy(1);
    return 0;
}


#endif
```

8.13 timrs.ml

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


type action = Ast | Sast | LLVM_IR | Compile


let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
```

```
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./tirms.native [-a|-s|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename)
usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Timrsparse.program Scanner.token lexbuf in
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
    match !action with
      Ast     -> ()
    | Sast    -> print_string (Sast.string_of_sprogram sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate
sast))
    | Compile -> let m = Codegen.translate sast in
      Llvm_analysis.assert_valid_module m;
      print_string (Llvm.string_of_llmodule m)
```

8.14 timrsparse.mly

```
/* Ocamlyacc parser for Timrs */
%{
open Ast
let parse_error s =
      begin
        try
          let start_pos = Parsing.symbol_start_pos ()
          and end_pos = Parsing.symbol_end_pos () in
          Printf.printf "File \"%s\", line %d, characters %d-%d: \n"
            start_pos.pos_fname
            start_pos.pos_lnum
            (start_pos.pos_cnum - start_pos.pos_bol)
            (end_pos.pos_cnum - start_pos.pos_bol)
        with Invalid_argument(_) -> ()
      end;
```

```
      Printf.printf "Syntax error: %s\n" s;
      raise Parsing.Parse_error
%}
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA PLUS MINUS TIMES DIVIDE
ASSIGN
%token NOT EQ NEQ LT LEQ GT GEQ AND OR
%token RETURN IF ELSE FOR WHILE INT BOOL STR FLOAT VOID
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID FLIT STRLIT
%token EOF
%start program
%type <Ast.program> program
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT
%%
program:
  decls EOF { $1 }
decls:
   /* nothing */ { ([], [])               }
 | decls vdecl { (($2 :: fst $1), snd $1) }
 | decls fdecl { (fst $1, ($2 :: snd $1)) }
fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { typ = $1;
       fname = $2;
       formals = List.rev $4;
       locals = List.rev $7;
       body = List.rev $8 } }
formals_opt:
    /* nothing */ { [] }
  | formal_list   { $1 }
formal_list:
    typ ID                   { [($1,$2)]     }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }
```

```
typ:
    INT   { Int   }
  | BOOL  { Bool  }
  | FLOAT { Float }
  | STR   { Str   }
  | VOID  { Void  }
vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }
vdecl:
    typ ID SEMI { ($1, $2) }
stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }
stmt:
    expr SEMI                               { Expr $1               }
  | RETURN expr_opt SEMI                    { Return $2             }
  | LBRACE stmt_list RBRACE                 { Block(List.rev $2)    }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7)        }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                            { For($3, $5, $7, $9)   }
  | WHILE LPAREN expr RPAREN stmt           { While($3, $5)         }
expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }
expr:
    LITERAL          { Literal($1)            }
  | FLIT               { Fliteral($1)           }
  | BLIT             { BoolLit($1)            }
  | STRLIT           { StrLit($1)             }
  | ID               { Id($1)                 }
  | expr PLUS   expr { Binop($1, Add,   $3)   }
  | expr MINUS  expr { Binop($1, Sub,   $3)   }
  | expr TIMES  expr { Binop($1, Mult,  $3)   }
  | expr DIVIDE expr { Binop($1, Div,   $3)   }
  | expr EQ     expr { Binop($1, Equal, $3)   }
  | expr NEQ    expr { Binop($1, Neq,   $3)   }
  | expr LT     expr { Binop($1, Less,  $3)   }
  | expr LEQ    expr { Binop($1, Leq,   $3)   }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3)   }
  | expr AND    expr { Binop($1, And,   $3)   }
```

```
  | expr OR     expr { Binop($1, Or,    $3)   }
  | MINUS expr %prec NOT { Unop(Neg, $2)      }
  | NOT expr         { Unop(Not, $2)          }
  | ID ASSIGN expr   { Assign($1, $3)         }
  | ID LPAREN args_opt RPAREN { Call($1, $3)  }
  | LPAREN expr RPAREN { $2                   }
args_opt:
    /* nothing */ { [] }
  | args_list  { List.rev $1 }
args_list:
    expr                    { [$1] }
  | args_list COMMA expr { $3 :: $1 }
```

8.15 Test Suite

```
tests
./fail-comment1.mc
./fail-comment2.mc
./fail-dead1.mc
./fail-dead2.mc
./fail-expr1.mc
./fail-expr2.mc
./fail-expr3.mc
./fail-float-assign1.mc
./fail-float-assign2.mc
./fail-float1.mc
./fail-float2.mc
./fail-for1.mc
./fail-for2.mc
./fail-for3.mc
./fail-for4.mc
./fail-for5.mc
./fail-func1.mc
./fail-func2.mc
./fail-func3.mc
./fail-func4.mc
./fail-func5.mc
./fail-func6.mc
./fail-func7.mc
./fail-func8.mc
./fail-func9.mc
./fail-global1.mc
```

```
./fail-global2.mc
./fail-if1.mc
./fail-if2.mc
./fail-if3.mc
./fail-int-assign1.mc
./fail-int-assign2.mc
./fail-int-assign3.mc
./fail-int-assign4.mc
./fail-nomain.mc
./fail-print.mc
./fail-printb.mc
./fail-printbig.mc
./fail-return1.mc
./fail-return2.mc
./fail-string-assign1.mc
./fail-string-assign2.mc
./fail-string-assign3.mc
./fail-string-assign4.mc
./fail-timer-assign1.mc
./fail-timer-assign2.mc
./fail-timer-assign3.mc
./fail-timer-assign4.mc
./fail-while1.mc
./fail-while2.mc
./test-add1.mc
./test-arith1.mc
./test-arith2.mc
./test-arith3.mc
./test-fib.mc
./test-float1.mc
./test-float2.mc
./test-float3.mc
./test-for1.mc
./test-for2.mc
./test-func1.mc
./test-func2.mc
./test-func3.mc
./test-func4.mc
./test-func5.mc
./test-func6.mc
./test-func7.mc
./test-func8.mc
./test-func9.mc
```

```
./test-gcd.mc
./test-gcd2.mc
./test-global1.mc
./test-global2.mc
./test-global3.mc
./test-hello.mc
./test-if1.mc
./test-if2.mc
./test-if3.mc
./test-if4.mc
./test-if5.mc
./test-if6.mc
./test-local1.mc
./test-local2.mc
./test-ops1.mc
./test-ops2.mc
./test-printbig.mc
./test-var1.mc
./test-var2.mc
./test-while1.mc
./test-while2.mc
```