State Machine Generator Final Report

COMS W4115

Oliver Zhou
ohz2101

May 14, 2014

Contents

1. Introduction
    1.1 Description
    1.2 Program
    1.3 Execution Results
2. Language Tutorial
    2.1 Program Execution
    2.2 Program Layout
        2.2.1 Signals
        2.2.2 States
        2.2.3 Actions
3. Language Manual
    3.1 Lexical Conventions
        3.1.1 Keywords
        3.1.2 Commenting Code
    3.2 Syntax
        3.2.1 Machine Definition and Instantiation
    3.3 Program Execution
4. Project Plan
    4.1 Planning
    4.2 Specification
    4.3 Development
    4.4 Testing
    4.5 Style Guide
    4.6 Project Timeline
5. Architectural Design

5.1 Block Diagram of Translator

6. Test Plan

6.1 Representative Language Programs

6.2.1 Full Example Test Program

6.2.2 simple.smgl

6.2.3 error1.smgl

6.2.4 error2.smgl

7. Lessons Learned

8. Appendix

1 Introduction

1.1 Description

The State Machine Generator Language, or SMGL, is a programming language focused on facilitating development utilizing finite state machines.

Finite state machines, or FSM, are powerful mathematical models that can simulate and solve nearly any type of problem in a clear and understandable way. There are many areas of study in this area, and are considered part of the field of Automata Theory.

Using only high level statements, SMGL allows for anyone to quickly and easily create a simulation of a state machine without worrying about implementation details. The State Machine Generator Language is designed to facilitate rapid simulation and prototyping of the logic of state machines, and can be used to test the logic of your system.

1.2 Program

A program in SMGL is a file consisting of several major sections. The first section will define the signals involved with the input and output to the state machines. The second section will define the various states in the state machine to be tested. The third section can define the actions and inputs to the state machine, or optionally be left blank if further low level of modification of the resulting C is desired without the C providing any simulated output.

1.3 Execution Results

The result of the execution of a SMGL program would be a C file that represents the state machine defined in SMGL. The optional actions portion of a SMGL program also allow for rapid testing and viewing of the state machine being designed in SMGL if the C file is compiled and executed.

2. Language Tutorial

2.1 Program Execution

SMGL files can be compiled into C by the execution of ./smgl in the directory

To compile the C as well, a shell script was created ./compilesmgl.sh if you wish to go directly from the smgl language to generate the compiled C code as well.

2.2 Program Layout

SMGL is laid out with three sections to a source file. The language expects the user to properly define a Signal Section, a States Section, and an Actions section.

In each section, a portion of the logic of the state machine is defined

2.2.1 Signals

In this section, the language expects all input signals that generate the transitions for the state machine to be defined. The signals are typed to be boolean style.

Example :

```
SIGNALS
{
 CTRL_IN coin; //Coin detection
 CTRL_IN push; //Push gate Detection
}
```

2.2.2 States

In this section the various states of the state machine are defined. The transitions for each state given a particular action are defined.

Example :

```
STATES
{
locked:
	if coin => unlocked;
	if push => locked;
unlocked:
	if coin => unlocked;
	if push => locked;
}
```

2.2.3 Actions

The interactions between the various states of the state machine are defined here. For example, the default state of the state machine will be set at this time.

```
ACTIONS
{
```

```
//DEFAULT_STATE needs to be defined
DEFAULT_STATE = locked;
//actions happen here
LIST_ACTIONS = push, coin, push;
coin : "Coin Inserted \n";
push : "Lock Pushed \n";
locked : "Door Locked \n";
unlocked : "Door Unlocked \n";
```

3. Language Reference Manual

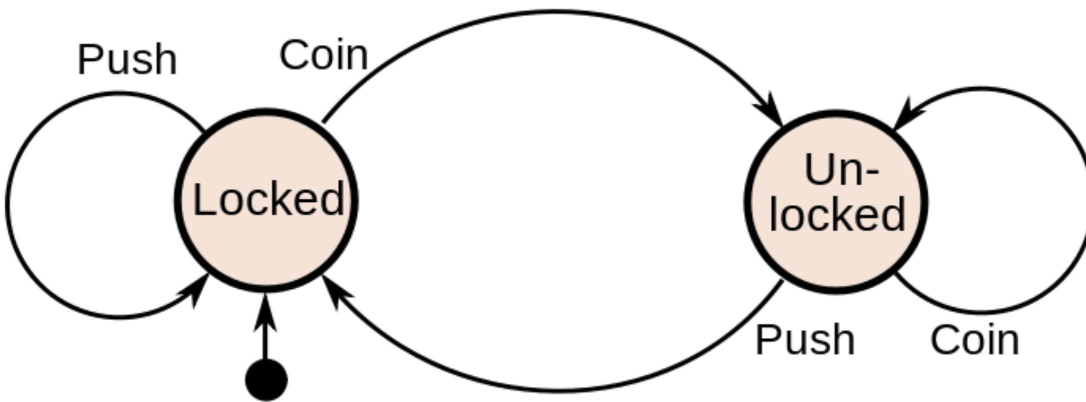
3.1 Lexical Conventions

3.1.1 Keywords
Identifiers reserved by the language and may not be user defined

CTRL_IN, short for control, is used to define an input variable that
controls the transitions from state to state

SIGNALS, used to create a section to pre-define all of the inputs and outputs that
the state machine will be using boolean style logic that translates well to the systems that we
are replicating.

STATES, used to create a section to define all states of the state machine being created



ACTIONS, used to define a section that allows for simulating

DEFAULT_STATE, used to define the starting state

LIST_ACTIONS, used to define a list of the signals that will be executed in the compiled C

if, Used during state definition for the logic of defining next states

In addition to those above, there is a selection of keywords used in loops and boolean logic
operations that are reserved.

Table 1. Other symbols reserved for use by SMGL

Operator Name	Symbol	Example

State define start	:	statename: is used at the beginning to define a state signals: is used to define a message caused by the action
Parathesis	()	Used for order of operations
standard closing statement	;	The semicolon lets the parser know the definition has ended
Next State	=>	Defining the next state that occurs after an action

Constants

In SMGL, all constants will be in the form of booleans, which will be represented by 0 and 1, with 0 being false, and 1 being true.

Simulated Actions

In the actions portion of a SMGL program, the actions of the various states are defined in array format.

The “ACTIONS” portion of a SMGL program will define the default starting state of the state machine

3.1.2 Commenting Code

Comments are delineated as everything between the string “//” and the new line character, effectively meaning that the double backslash prevents the remainder of the line from being read.

```
 // commented out
STATE: run_test_idle(TMS) // this text is commented out
{
….
}
```

3.2 Syntax

Figure 1 Example of Turnstile state machine (source: Wikipedia)
This figure is used to assist in demonstrating the syntax of a program

3.2.1 Machine Definition and Instantiation

Signals are defined first in its own group in a bracketed segment after the keyword SIGNALS is called. All signals are expected to be boolean style and will be used in both controlling state transitions.

```
SIGNALS
{
        CTRL_IN X
}
```

Example fitting the example machine in Fig. 1

```
SIGNALS
{
 CTRL_IN coin; //Coin detection
 CTRL_IN push; //Push gate Detection
}
```

States are then defined next. Each state will be declared with lower case characters for differentiation from the signals. Each state is required to have one if statement to determine action for state to state transitions. An else statement is used if the state does not loop back into itself. All state definitions will fall under the brackets of the overall STATES keyword. State transitions are declared with an if statement that defines the next state given an specific input signal.

```
STATES
{
locked:
	if coin => unlocked;
	if push => locked;
unlocked:
	if coin => unlocked;
	if push => locked;
}
```

The final section is separated by the ACTIONS keyword. This is where the chain of inputs to the state would be put in, which would also be simulated in the resulting C code after execution of the program. Starting default state needs to be defined here. The list of input actions also needs to be defined. The example below, the inputs are a push, then a coin insertion, then finally, a push. Each action and state has a defined message for when either that action occurs, or when that state is reached. This output needs to be in string form, and will be output in string form in the final compiled output. Perhaps extended in the future for more flexible output.

```
ACTIONS
{
	//DEFAULT_STATE needs to be defined
	DEFAULT_STATE = locked;
	//actions happen here
	LIST_ACTIONS = push, coin, push;
	coin : "Coin Inserted \n";
	push : "Lock Pushed \n";
	locked : "Door Locked \n";
	unlocked : "Door Unlocked \n";
```

3.3 Program Execution

The program file will be in an ASCII based file with the file ending of .smgl.
The file after it compiles will then be a C file that simulates the structures defined in the .smgl file. Scripts for automation can be utilized to take the smgl code directly to a compiled output of gcc.

If the ACTIONS portion of the program is defined properly, then code will also be generated in C that will print out the state of the state machine after every inputted signals that was provided in the code.

i.e., based on the state machine in Figure 1, if the following code would report the specific output of the state machine after each action.

LIST_ACTIONS = push, coin, push;

Yields results of

```
Door Locked
Lock Pushed
Door Locked
Coin Inserted
Door Unlocked
Lock Pushed
```

4. Project Plan

4.1 Planning

Planning was done concurrently during development of the language proposal and language reference manual. After that process, a review of the C code that would be represent what the end-goal of the language was created in order to create a language that would be able to capture all the data needed to compile into C.

Development and testing plans were created along the suggestions of the course, and sample tests were drawn up to test the various functions of the program.

4.2 Specification

Continuously evolving and changing. Decisions about the output of the compiler were continuously changing. As the development progressed, the complexity of the compiler was modified and the code scrapped after playing with the development and realizing what kind of language was best to interpret.

4.3 Development

Since this project was an individual effort, the commits and utilization of a revision control system were not used as frequently as a team effort requires. Based on methodology from industry, commits are traditionally only made after a fix is successfully working overall. Therefore, commits were rare and focused on items that were actually fixed. With additional team members, development would have required more utilization of the SVN repo.

4.4 Testing

Initial development did not utilize testing properly, and development was unable to proceed and was destroyed. Finally, additional tests were created with the development of the revamped language specification, and was used to develop the language better

4.5 Style Guide

The general style of the project was to be done with standard coding conventions with as much commenting as possible, and utilizing camel case as needed. For the SMGL language itself, it was decided that keeping many of the keywords capitalized made the language easier to read

4.6 Project Timeline

2/9/2014 Proposal was initially begun and the ideas for the language were drafted
2/11/2014 Initial Proposal was submitted
3/10/2014 LRM was initially begun, modification occurred over the subsequent days
3/13/2014 LRM Submission
3/14/2014 Initial modification of the AST and Scanner
4/18/2014 Development of the Final Project report begins
5/2/2014 Initial design of the C output was scrapped and redesigned
5/4/2014 Architecture of project laid out
5/9/2014 Development of new tests
5/14/2014 Submission of tests/codebase/final report

4.7 Tools

BBEdit on OSX was the primary development tool and platform used during the development process. VIM was used when terminal editing was faster during the debug process.

The compiler was developed with OCaml, or Objective Caml. The output of the SMGL compiler was C, which was compiled using the GCC C compiler.

Revision control wasn't as important since the project was developed by one person only, but SVN was used to help control the development process.

4.8 Project Log

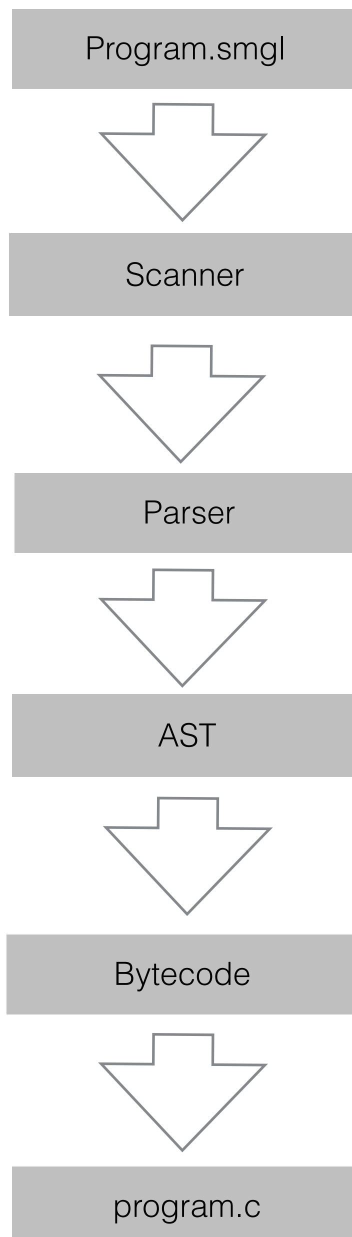
```
olivers-mbp:smgl ozhou$ svn log
------------------------------------------------------------------------
r11 | zhou.oliver@gmail.com | 2014-05-14 23:49:27 -0700 (Wed, 14 May 2014) | 2 lines

Modifying tokens and operations

------------------------------------------------------------------------
r10 | zhou.oliver@gmail.com | 2014-05-14 23:41:30 -0700 (Wed, 14 May 2014) | 2 lines

Final commits and changes to scanner and parser

------------------------------------------------------------------------
r9 | zhou.oliver@gmail.com | 2014-05-14 21:35:21 -0700 (Wed, 14 May 2014) | 2 lines

tests rearranged

------------------------------------------------------------------------
r8 | zhou.oliver@gmail.com | 2014-05-14 20:57:04 -0700 (Wed, 14 May 2014) | 1 line

More clean up
------------------------------------------------------------------------
r7 | zhou.oliver@gmail.com | 2014-05-14 20:56:41 -0700 (Wed, 14 May 2014) | 1 line

Clean up Directory
------------------------------------------------------------------------
r6 | zhou.oliver@gmail.com | 2014-05-11 16:23:30 -0700 (Sun, 11 May 2014) | 1 line

example
------------------------------------------------------------------------
r5 | zhou.oliver@gmail.com | 2014-05-09 22:02:39 -0700 (Fri, 09 May 2014) | 2 lines
```

```
Moving working copy into revision control

------------------------------------------------------------------------
r4 | zhou.oliver@gmail.com | 2014-05-04 21:28:32 -0700 (Sun, 04 May 2014) | 1 line

updated scanner
------------------------------------------------------------------------
r3 | zhou.oliver@gmail.com | 2014-05-04 12:04:22 -0700 (Sun, 04 May 2014) | 1 line

smgl files put in
------------------------------------------------------------------------
r2 | zhou.oliver@gmail.com | 2014-04-27 23:21:05 -0700 (Sun, 27 Apr 2014) | 2 lines

First commit with microc examples

------------------------------------------------------------------------
r1 | (no author) | 2014-04-27 23:16:24 -0700 (Sun, 27 Apr 2014) | 1 line

Initial directory structure.
------------------------------------------------------------------------
```

5. Architectural Design

SMGL follows traditionally taught compiler design by utilizing a scanner and parser to read and understand the language. The scanner tokenizes the input, removing whitespace and comments. The parser will then read the tokens and creates an AST. Then, the abstract syntax tree processes the language into bytecode that the C compiler is then able to understand. Following that, the external gcc C compiler is able to handle the code and create a final executable.

5.1 Block Diagram of Translator



6. Test Plan

6.1 Representative Language Programs

Several test programs were used to test the error handling and validity of individual portions of the language. A small functional program was created to test the basic ability to handle one state and one input. Then, a larger functional program was created to illustrate the example of the turnstile utilized in the LRM. testall.sh was modified to run the compiler on all .smgl files in the test directory, and then run the gcc compiler on to take it to the final step. The final results and output of the compiled C code was compared against the expected output to determine success.

6.2.1 Full Example Test Program

example.smgl

```
SIGNALS
{
 CTRL_IN coin; //Coin detection
 CTRL_IN push; //Push gate Detection
}
STATES
{
locked:
        if coin => unlocked;
        if push => locked;
unlocked:
        if coin => unlocked;
        if push => locked;
}
ACTIONS

{
        //DEFAULT_STATE needs to be defined
        DEFAULT_STATE = locked;
        //actions happen here
        LIST_ACTIONS = push, coin, push;
        coin : "Coin Inserted \n";
        push : "Lock Pushed \n";
        locked : "Door Locked \n";
        unlocked : "Door Unlocked \n";
}
```

example.c

```
#include <stdio.h>
#define input_size 3

typedef enum STATES {
locked,
unlocked
} STATES;
```

```
typedef enum SIGNALS {
coin,
push
} SIGNALS;

void coin_action ()
{
        printf("Coin Inserted \n");
}
void push_action ()
{
        printf("Lock Pushed \n");
}
void lock_action ()
{
        printf("Door Locked \n");
}
void unlock_action ()
{
   printf("Door Unlocked \n");
}
int main()
{
        STATES state = locked;
        int event = 0;
        SIGNALS input_signals[input_size] = {push, coin, push};
        for(; event < input_size; event++)
        {
                        switch(input_signals[event])
                        {
                                case push:
                                {
                                        switch(state)
                                        {
                                                case locked:
                                                {
                lock_action();
                push_action();
                                                        state = locked;
                                                        break;
                                                }
                                                case unlocked:
                                                {
                unlock_action();
                                                        push_action();
                                                        state = locked;
                                                        break;
                                                }
                                        }
                                break;
                                }
                                case coin:
                                {
                                        switch(state)
```

```
                                        {
                                                case locked:
                                                {
                        lock_action();
                                                        coin_action();
                                                        state = unlocked;
                                                        break;
                                                }
                                                case unlocked:
                                                {
                        unlock_action();
                                                        coin_action();
                                                        state = unlocked;
                                                        break;
                                                }
                                        }
                                        break;
                                }
                        }
                }
        return 0;
}
```

output of compiled example.out

```
Door Locked
Lock Pushed
Door Locked
Coin Inserted
Door Unlocked
Lock Pushed
```

6.2.2 simple.smgl

```
SIGNALS
{
 CTRL_IN x; //Coin detection
}
STATES
{
s1:
        if x => s1;
}
ACTIONS

{
        //DEFAULT_STATE needs to be defined
        DEFAULT_STATE =  s1;
        //actions happen here
        LIST_ACTIONS = x;
        s1 : "s1 \n";
}
```

simple.c

```
#include <stdio.h>
#define input_size 1

typedef enum STATES {
s1,
} STATES;

typedef enum SIGNALS {
x
} SIGNALS;

void s1_action ()
{
        printf("s1 \n");
}
void x_action ()
{
   printf("x \n");
}
int main()
{
        STATES state = s1;
        int event = 0;
        SIGNALS input_signals[input_size] = {x};
        for(; event < input_size; event++)
        {
                        switch(input_signals[event])
                        {
                                case x:
                                {
                                        switch(state)
                                        {
                                                case s1:
                                                {
                                                        s1_action();
                x_action();
                                                        state = s1;
                                                        break;
                                                }
                                        }
                                break;
                                }
                        }
                }
        return 0;
}
```

simple.out

```
s1
x
```

6.2.3 error1.smgl

```
SIGNALS
{
 CTRL_IN x; //Coin detection
}
STATES
{
s1:
        if x : s1;
}
```

error1.smgl output : Error : Missing ACTIONS

6.2.4 error2.smgl

```
SIGNALS
{
 CTRL_IN x; //Coin detection
}
ACTIONS

{
        //DEFAULT_STATE needs to be defined
        DEFAULT_STATE =  s1;
        //actions happen here
        LIST_ACTIONS = x;
        s1 : "s1 \n";
}
```

error2.smgl output : Error : Missing STATES

6.2.4 error3.smgl

```
STATES
{
s1:
        if x : s1;
}
ACTIONS

{
        //DEFAULT_STATE needs to be defined
        DEFAULT_STATE =  s1;
        //actions happen here
        LIST_ACTIONS = x;
        s1 : "s1 \n";
}
```

error3.smgl output : Error : Missing SIGNALS

7 Lessons Learned

The most important item learned from the development of this project is that development has a large learning curve. During development, it seems easy to follow a condensed development schedule, but as the development process occurred, continual changes and learning occurred, forcing further changes to the project. In a constrained time period, it may become difficult to recover.

The advice to for future teams is to expect larger delays than other normal programming assignments.

8. Appendix

Complete code listing of translator

testall.sh script that compiles and runs the tests

```
#!/bin/sh

for s in ./tests/*.smgl
do
./smgl "$s"
done

#Modified to use gcc in lieu of the compiler, since all inputs are compiled in the line above

SMGL="./gcc"

# 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] [.smgl 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
   }
}

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 ${basename}.i.out" &&
   Run "$SMGL" "-i" "<" $1 ">" ${basename}.i.out &&
   Compare ${basename}.i.out ${reffile}.out ${basename}.i.diff

   generatedfiles="$generatedfiles ${basename}.c.out" &&
   Run "$SMGL" "-c" "<" $1 ">" ${basename}.c.out &&
   Compare ${basename}.c.out ${reffile}.out ${basename}.c.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`

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

scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "//"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ':'      { COLON }
| ','      { COMMA }
| '='      { ASSIGN }
| "=>"     { SETTO }
| "if"     { IF }
| "CTRL_IN"    { SIGNAL_IN }
| "DEFAULT_STATE" { DEFAULT_STATE  }
| "LIST_ACTIONS" { LIST_ACTIONS }
| "true"   { BOOLEAN_LIT(true) }
| "false"  { BOOLEAN_LIT(false) }
| ['0'-'9']+ as lxm { LITERAL(int_of_string 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
  "\n" { token lexbuf }
| eof  { EOF }
| _    { singlelinecom lexbuf}
```

parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA COLON
%token ASSIGN
%token SETTO
%token IF SIGNAL_IN DEFAULT_STATE LIST_ACTIONS BOOLEAN_LIT(true)
BOOLEAN_LIT(false)
%token <int> LITERAL
%token <string> ID
%token EOF
%right ASSIGN
%right SETTO
%right COLON

%start program
%type <Ast.program> program

%%

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

fdecl:
   ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { fname = $1;
         formals = $3;
         locals = List.rev $6;
         body = List.rev $7 } }

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

formal_list:
    ID                  { [$1] }
  | formal_list COMMA ID { $3 :: $1 }

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

vdecl:
   INT ID SEMI { $2 }
```

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

stmt:
   expr SEMI { Expr($1) }
  | 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
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

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

expr:
   LITERAL          { Literal($1) }
  | ID               { Id($1) }
  | 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) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }

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

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

smgl.ml

```
type action = Ast | Interpret | Bytecode | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);
                              ("-i", Interpret);
                              ("-b", Bytecode);
                              ("-c", Compile) ]
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  match action with
   Ast -> let listing = Ast.string_of_program program
```

```
        in print_string listing
  | Interpret -> ignore (Interpret.run program)
  | Bytecode -> let listing =
      Bytecode.string_of_prog (Compile.translate program)
    in print_endline listing
  | Compile -> Execute.execute_prog (Compile.translate program)
```

unmodified interpret.ml

```
open Ast

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

exception ReturnException of int * int NameMap.t

(* Main entry point: run a program *)

let run (vars, funcs) =
  (* Put function declarations in a symbol table *)
  let func_decls = List.fold_left
      (fun funcs fdecl -> NameMap.add fdecl.fname fdecl funcs)
      NameMap.empty funcs
  in

  (* Invoke a function and return an updated global symbol table *)
  let rec call fdecl actuals globals =

    (* Evaluate an expression and return (value, updated environment) *)
    let rec eval env = function
          Literal(i) -> i, env
      | Noexpr -> 1, env (* must be non-zero for the for loop predicate *)
      | Id(var) ->
            let locals, globals = env in
            if NameMap.mem var locals then
              (NameMap.find var locals), env
            else if NameMap.mem var globals then
              (NameMap.find var globals), env
            else raise (Failure ("undeclared identifier " ^ var))
      | Binop(e1, op, e2) ->
            let v1, env = eval env e1 in
        let v2, env = eval env e2 in
            let boolean i = if i then 1 else 0 in
            (match op with
              Add -> v1 + v2
            | Sub -> v1 - v2
            | Mult -> v1 * v2
            | Div -> v1 / v2
            | Equal -> boolean (v1 = v2)
            | Neq -> boolean (v1 != v2)
            | Less -> boolean (v1 < v2)
```

```
        | Leq -> boolean (v1 <= v2)
        | Greater -> boolean (v1 > v2)
        | Geq -> boolean (v1 >= v2)), env
  | Assign(var, e) ->
        let v, (locals, globals) = eval env e in
        if NameMap.mem var locals then
          v, (NameMap.add var v locals, globals)
        else if NameMap.mem var globals then
          v, (locals, NameMap.add var v globals)
        else raise (Failure ("undeclared identifier " ^ var))
  | Call("print", [e]) ->
        let v, env = eval env e in
        print_endline (string_of_int v);
        0, env
  | Call(f, actuals) ->
        let fdecl =
          try NameMap.find f func_decls
          with Not_found -> raise (Failure ("undefined function " ^ f))
        in
        let actuals, env = List.fold_left
            (fun (actuals, env) actual ->
                  let v, env = eval env actual in v :: actuals, env)
            ([], env) (List.rev actuals)
        in
        let (locals, globals) = env in
        try
          let globals = call fdecl actuals globals
          in 0, (locals, globals)
        with ReturnException(v, globals) -> v, (locals, globals)
in

(* Execute a statement and return an updated environment *)
let rec exec env = function
      Block(stmts) -> List.fold_left exec env stmts
  | Expr(e) -> let _, env = eval env e in env
  | If(e, s1, s2) ->
        let v, env = eval env e in
        exec env (if v != 0 then s1 else s2)
  | While(e, s) ->
        let rec loop env =
          let v, env = eval env e in
          if v != 0 then loop (exec env s) else env
        in loop env
  | For(e1, e2, e3, s) ->
        let _, env = eval env e1 in
        let rec loop env =
          let v, env = eval env e2 in
          if v != 0 then
            let _, env = eval (exec env s) e3 in
            loop env
          else
            env
        in loop env
  | Return(e) ->
```

```
        let v, (locals, globals) = eval env e in
        raise (ReturnException(v, globals))
  in

  (* Enter the function: bind actual values to formal arguments *)
  let locals =
    try List.fold_left2
        (fun locals formal actual -> NameMap.add formal actual locals)
        NameMap.empty fdecl.formals actuals
    with Invalid_argument(_) ->
      raise (Failure ("wrong number of arguments passed to " ^ fdecl.fname))
  in
  (* Initialize local variables to 0 *)
  let locals = List.fold_left
      (fun locals local -> NameMap.add local 0 locals) locals fdecl.locals
  in
  (* Execute each statement in sequence, return updated global symbol table *)
  snd (List.fold_left exec (locals, globals) fdecl.body)

 (* Run a program: initialize global variables to 0, find and run "main" *)
 in let globals = List.fold_left
    (fun globals vdecl -> NameMap.add vdecl 0 globals) NameMap.empty vars
 in try
  call (NameMap.find "main" func_decls) [] globals
 with Not_found -> raise (Failure ("did not find the main() function"))
```

unmodified execute.ml

```
open Ast
open Bytecode

(* Stack layout just after "Ent":

            <-- SP
   Local n
   ...
   Local 0
   Saved FP   <-- FP
   Saved PC
   Arg 0
   ...
   Arg n *)

let execute_prog prog =
 let stack = Array.make 1024 0
 and globals = Array.make prog.num_globals 0 in

 let rec exec fp sp pc = match prog.text.(pc) with
  Lit i  -> stack.(sp) <- i ; exec fp (sp+1) (pc+1)
 | Drp    -> exec fp (sp-1) (pc+1)
 | Bin op -> let op1 = stack.(sp-2) and op2 = stack.(sp-1) in
    stack.(sp-2) <- (let boolean i = if i then 1 else 0 in
    match op with
      Add     -> op1 + op2
```

```
      | Sub     -> op1 - op2
      | Mult    -> op1 * op2
      | Div     -> op1 / op2
      | Equal   -> boolean (op1 =  op2)
      | Neq     -> boolean (op1 != op2)
      | Less    -> boolean (op1 <  op2)
      | Leq     -> boolean (op1 <= op2)
      | Greater -> boolean (op1 >  op2)
      | Geq     -> boolean (op1 >= op2)) ;
      exec fp (sp-1) (pc+1)
  | Lod i   -> stack.(sp)   <- globals.(i)  ; exec fp (sp+1) (pc+1)
  | Str i   -> globals.(i)  <- stack.(sp-1) ; exec fp sp     (pc+1)
  | Lfp i   -> stack.(sp)   <- stack.(fp+i) ; exec fp (sp+1) (pc+1)
  | Sfp i   -> stack.(fp+i) <- stack.(sp-1) ; exec fp sp     (pc+1)
  | Jsr(-1) -> print_endline (string_of_int stack.(sp-1)) ; exec fp sp (pc+1)
  | Jsr i   -> stack.(sp)   <- pc + 1       ; exec fp (sp+1) i
  | Ent i   -> stack.(sp)   <- fp           ; exec sp (sp+i+1) (pc+1)
  | Rts i   -> let new_fp = stack.(fp) and new_pc = stack.(fp-1) in
               stack.(fp-i-1) <- stack.(sp-1) ; exec new_fp (fp-i) new_pc
  | Beq i   -> exec fp (sp-1) (pc + if stack.(sp-1) =  0 then i else 1)
  | Bne i   -> exec fp (sp-1) (pc + if stack.(sp-1) != 0 then i else 1)
  | Bra i   -> exec fp sp (pc+i)
  | Hlt     -> ()

  in exec 0 0 0
```

unmodified compile.ml

```
open Ast
open Bytecode

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)
type env = {
    function_index : int StringMap.t; (* Index for each function *)
    global_index   : int StringMap.t; (* "Address" for global variables *)
    local_index    : int StringMap.t; (* FP offset for args, locals *)
  }

(* val enum : int -> 'a list -> (int * 'a) list *)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)
let string_map_pairs map pairs =
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

(** Translate a program in AST form into a bytecode program.  Throw an
    exception if something is wrong, e.g., a reference to an unknown
    variable or function *)
let translate (globals, functions) =
```

```
(* Allocate "addresses" for each global variable *)
let global_indexes = string_map_pairs StringMap.empty (enum 1 0 globals) in

(* Assign indexes to function names; built-in "print" is special *)
let built_in_functions = StringMap.add "print" (-1) StringMap.empty in
let function_indexes = string_map_pairs built_in_functions
    (enum 1 1 (List.map (fun f -> f.fname) functions)) in

(* Translate a function in AST form into a list of bytecode statements *)
let translate env fdecl =
  (* Bookkeeping: FP offsets for locals and arguments *)
  let num_formals = List.length fdecl.formals
  and num_locals = List.length fdecl.locals
  and local_offsets = enum 1 1 fdecl.locals
  and formal_offsets = enum (-1) (-2) fdecl.formals in
  let env = { env with local_index = string_map_pairs
                     StringMap.empty (local_offsets @ formal_offsets) } in

  let rec expr = function
        Literal i -> [Lit i]
    | Id s ->
          (try [Lfp (StringMap.find s env.local_index)]
        with Not_found -> try [Lod (StringMap.find s env.global_index)]
        with Not_found -> raise (Failure ("undeclared variable " ^ s)))
    | Binop (e1, op, e2) -> expr e1 @ expr e2 @ [Bin op]
    | Assign (s, e) -> expr e @
          (try [Sfp (StringMap.find s env.local_index)]
          with Not_found -> try [Str (StringMap.find s env.global_index)]
          with Not_found -> raise (Failure ("undeclared variable " ^ s)))
    | Call (fname, actuals) -> (try
          (List.concat (List.map expr (List.rev actuals))) @
          [Jsr (StringMap.find fname env.function_index) ]
      with Not_found -> raise (Failure ("undefined function " ^ fname)))
    | Noexpr -> []

  in let rec stmt = function
        Block sl     ->  List.concat (List.map stmt sl)
    | Expr e       -> expr e @ [Drp]
    | Return e     -> expr e @ [Rts num_formals]
    | If (p, t, f) -> let t' = stmt t and f' = stmt f in
        expr p @ [Beq(2 + List.length t')] @
        t' @ [Bra(1 + List.length f')] @ f'
    | For (e1, e2, e3, b) ->
          stmt (Block([Expr(e1); While(e2, Block([b; Expr(e3)]))]))
    | While (e, b) ->
          let b' = stmt b and e' = expr e in
          [Bra (1+ List.length b')] @ b' @ e' @
          [Bne (-(List.length b' + List.length e'))]

  in [Ent num_locals] @      (* Entry: allocate space for locals *)
  stmt (Block fdecl.body) @  (* Body *)
  [Lit 0; Rts num_formals]   (* Default = return 0 *)

in let env = { function_index = function_indexes;
```

```
                        global_index = global_indexes;
                        local_index = StringMap.empty } in

  (* Code executed to start the program: Jsr main; halt *)
  let entry_function = try
    [Jsr (StringMap.find "main" function_indexes); Hlt]
  with Not_found -> raise (Failure ("no \"main\" function"))
  in

  (* Compile the functions *)
  let func_bodies = entry_function :: List.map (translate env) functions in

  (* Calculate function entry points by adding their lengths *)
  let (fun_offset_list, _) = List.fold_left
      (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0) func_bodies in
  let func_offset = Array.of_list (List.rev fun_offset_list) in

  { num_globals = List.length globals;
    (* Concatenate the compiled functions and replace the function
       indexes in Jsr statements with PC values *)
    text = Array.of_list (List.map (function
           Jsr i when i > 0 -> Jsr func_offset.(i)
      | _ as s -> s) (List.concat func_bodies))
  }
```

bytecode.ml

```
type bstmt =
    Lit of int    (* Push a literal *)
  | Drp           (* Discard a value *)
  | Bin of Ast.op (* Perform arithmetic on top of stack *)
  | Lod of int    (* Fetch global variable *)
  | Str of int    (* Store global variable *)
  | Lfp of int    (* Load frame pointer relative *)
  | Sfp of int    (* Store frame pointer relative *)
  | Jsr of int    (* Call function by absolute address *)
  | Ent of int    (* Push FP, FP -> SP, SP += i *)
  | Rts of int    (* Restore FP, SP, consume formals, push result *)
  | Beq of int    (* Branch relative if top-of-stack is zero *)
  | Bne of int    (* Branch relative if top-of-stack is non-zero *)
  | Bra of int    (* Branch relative *)
  | Hlt          (* Terminate *)

type prog = {
    num_globals : int;   (* Number of global variables *)
    text : bstmt array; (* Code for all the functions *)
  }

let string_of_stmt = function
    Lit(i) -> "Lit " ^ string_of_int i
  | Drp -> "Drp"
  | Bin(Ast.Equal) -> "Eql"
  | Lod(i) -> "Lod " ^ string_of_int i
  | Str(i) -> "Str " ^ string_of_int i
```

```
  | Lfp(i) -> "Lfp " ^ string_of_int i
  | Sfp(i) -> "Sfp " ^ string_of_int i
  | Jsr(i) -> "Jsr " ^ string_of_int i
  | Ent(i) -> "Ent " ^ string_of_int i
  | Rts(i) -> "Rts " ^ string_of_int i
  | Bne(i) -> "Bne " ^ string_of_int i
  | Beq(i) -> "Beq " ^ string_of_int i
  | Bra(i) -> "Bra " ^ string_of_int i
  | Hlt    -> "Hlt"

let string_of_prog p =
  string_of_int p.num_globals ^ " global variables\n" ^
  let funca = Array.mapi
      (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text
  in String.concat "\n" (Array.to_list funca)
```

ast.ml

```
(*AST : input from parser.mly *)
type op = Setto | Colon

type expr =                                     (* Expressions *)
    Literal of int                        (*42 *)
  | Id of string                          (*foo*)
  | Binop of expr * op * expr    (*a+b*)
  | Assign of string * expr               (*foo=42*)
  | Call of string * expr list       (* foo(1, 25 *)
  | Noexpr                                          (* for (;;) *)

type stmt =                                         (* Statements *)
    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 = {
    fname : string;
    formals : string list;
    locals : string list;
    body : stmt list;
  }

type program = string list * func_decl list

(* Print out results of AST *)
let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
```

```
    (match o with
        Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
    | Equal -> "==" | Neq -> "!="
    | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=") ^ " " ^
    string_of_expr e2
  | 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_vdecl id = "int " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " 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)
```