CABG:
An Automata Specification Language

Max Czapanskiy   Raphael Almeida   Brody Berg   Shaina Graboyes

December 19, 2008

Contents

1. An Introduction to CABG 3
2. Tutorial 5
3. Language Reference Manual 6
4. Project Plan 11
5. Architectural Design 14
6. Test Plan 16
7. Lessons Learned 18
A. Appendix A – Source Code 21
B. Appendix B – Testing Source Code 39

Chapter 1

An Introduction to CABG

The CABG programming language allows users to program in terms of the states and transitions of Definite Finite Automata (DFA). DFAs are very important to computer science and many fundamental algorithms in networking, context-free grammars, and regular expressions can be represented by the states and transitions of a DFA. CABG seeks to provide an unobtrusive language for the quick and accurate creation of DFAs so that they may be simulated with little time investment.

1.1. Background

A Deterministic Finite Automaton is a finite state machine that is composed of states and transitions. DFAs take some input and for each different input, it transitions from one state to another. A DFA is formally defined as a 5-tuple (Q, Σ, δ, q_0, F) each of which represent the following:
- a finite set of states (Q)
- a finite set called the alphabet (Σ)
- a transition function ($\delta : S \times \Sigma \rightarrow S$)
- a start sate (q_0 Q)
- a set of accepting states ($F \subseteq Q$)
DFAs can be used to represent all regular languages and have different possible representations. DFAs can be specified formally by the 5-tuple listed above or it can be represented graphically like so:

CABG makes it very easy to specify a DFA if one of these two representations is available. Alternatively, one can use CABG's intuitive syntax to specify the desired automaton.

1.2. Goals of CABG

CABG is a fast and accurate way to create programs that simulate DFAs based on the typical graphical representation of DFAs.

1.2.1 Speed

DFAs are represented by states with transitions leading out of them. CABG's simple syntax allows a user to easily specify states and the transitions leading out of them without having to think in terms of anything aside from states and transitions. With many programming languages, a user would have to move away from the simple states and transition representation of a DFA in order to create a program that implements the DFA, with CABG a user needs only the states and transitions to create a program.

1.2.2 Accuracy

CABG accurately runs the specified DFA on the given input. CABG will produce the expected output faster than a human would be able to simulate the DFA on the input. CABG actually simulates the running of the DFA so the user is guaranteed that the output received is the output the DFA should produce.

1.2.3 Portable

CABG can run on any machine that can run OCAML. A user simply needs the CABG language files and a working installation of OCAML. OCAML is available for Windows, Linux, and Mac OS X and therefore CABG can also be run on any of these platforms.

1.2.4 Robust

CABG was designed with the intention that programmers would push its limits. CABG will catch syntax errors and will run on any DFA that would run.

1.2.5 Flexible

CABG allows states to be declared in any order and ignores newlines, tabs, and spaces. This makes CABG flexible and allows a user to format and write a file in the style the user wants.

1.3. Summary

CABG is a fast and accurate way to simulate DFAs. It uses states and transitions to define a DFA which the program simulates.

Chapter 2

Tutorial

2.1 Tutorial

CABG is a language based on finite automata. By specifying states and transitions, one is able to specify an automaton and what it should do given some specific input.

The following few examples serve as a quick start guide to the language:

2.1.1 Hello World

```
Start ()
   ? 1==1 : print("Hello World") -> ;
End
```

This program prints “Hello World” as output. It defines the initial stating state of the automaton and takes one action while it is in that state. The keyword “Start” is used to define the state where we will start execution. The keyword “End” is used to conclude the definition of a state. Each state is required to have a transition which is why the question mark and semicolon surround a condition that is always true. The arrow here serves to fulfill that same purpose.

2.1.2 State Transitions

```
Start ()
   x = 20
   ? x==20 : y = 10 -> ifTwoArgs x, y;
End

ifTwoArgs (a b)
   print(string_of_int(a + b))
   ? 1==1 : ->;
End
```

After assigning 20 to x, this program confirms if that is true, if so, it will define y to be 10 and transitions to state ifTwoArgs with x and y. This function then adds the two values and prints them.

Chapter 3

Language Reference Manual

3.1 Introduction

The CABG language is inspired by state machines such as Deterministic Finite Automata or Turing Machines. Code written in CABG is composed of state and transition definitions, as with a state machine, but it also includes variable declarations and imperative functions for ease of use. State machines allow the programmer to naturally describe algorithms related to protocols, from simple text processing to intricate network protocols such as TCP.
An advantage of using state machines is the wealth of theory about them. State machines can be described visually or mathematically. In the latter representation, a DFA can be completely described by a 5-tuple: (S, A, T, s, F) as follows:

- S = { $s_1, s_2, \ldots, s_n$ } is the set of states
- A = { $s_1, s_2, \ldots, s_k$ } is the alphabet
- T = S X A -> S is the transition function
- s – an element of S – is the start state
- F – a subset of S – is the set of accepting states.

3.2. Lexical conventions

There are six kinds of tokens:
Identifiers
Keywords
Constants
Strings
Expression Operators
Other Separators
In general, spaces, new lines, and comments serve to separate tokens.

3.2.1 Comments: The '#' character introduces a comment that ends at the ends with a new line character.

3.2.2 Identifiers: An identifier is a letter followed by zero or more letters or numbers. Characters beyond the 127th in an identifier might be ignored.

3.2.3 Keywords:
"Start" – names the initial state where code will begin execution
"End" – concludes a state

3.2.4 Constants: There are several kinds of constants, as follows:

3.2.4.1 Integer Constants
An integer constant is a sequence of digits beginning with 1-9, followed by zero or more numerical digits 0-9. The largest number is 9,999 and the smallest number is -9,999.

3.2.4.2 Strings
A string is zero to 255 characters from a-z, A-Z, 0-9 and space, tab, newline, and line feed all within double-quotes. Escape sequences for space, tab, newline and line feed are \s, \t, \n, and \r respectively. String constants do not behave like integers. A string composed only of numbers is a string and not an integer.

3. 3. Syntax Notation

In the Syntax used in this manual, anything described in courier new is example code.

3. 4. State Names

State names consist of one or more alphabetic (A-Z, a-z).

3. 5. Objects and lvalues

An object is a region of storage. Objects can be defined in external libraries as a logical grouping of data representation. (See "External Definitions") An lvalue is an expression referring to an object. In other words an lvalue is something that would be on the left hand side of an assignment statement.

3.6. Conversions

A type will be inferred upon the initial declaration (string or integer). From there on, the variable name used will refer to that type. All types must be used with explicit conversion. For example, to print an integer, it must be explicitly converted to a string prior to printing.

The two functions string_of_int and int_of_string must be used for the conversions described above.

3.7. Expressions

3.7.1 Transitions
In CABG, primary expressions are expressions separated by the following operators ? : -> which are grouped from left to right.

3.7.1.1 identifier
An identifier can represent a state, a string or an int.

3.7.1.2 string
A string is a primary expression. We provide access to the entire string only, however, CABG authors can write extended string manipulation libraries.

3.7.1.4 state call (destination)
A state call may be initiated in the final expression of a transition. It may include parameters separated by commas ending with a semi-colon.

3.7.1.7 Transitions
Transitions take the following form: ? *condition* : *action* -> *destination ;*
where *condition* is a valid expression evaluating to a Boolean, *action* is a sequence of zero or more statements separated by spaces, a *destination* is another state in the file where execution continues. A destination may also simply return a value rather than invoke another state.

3.7.4 Multiplicative
* / Multiplication and Division
Groups from left to right. Multiplication and division only operate on integer values.

3.7.5 Additive
+ - Addition and Subtraction
Groups from left to right. Addition and subtraction only operate on integer values.

3.7.6 Relational operators
< > <= >=
Take the form: *expression operation expression*. The expressions must be of the same type. Returns a Boolean value.

3.7.7 Equality operators
==, Equals
Take the form: *expression operation expression*. The expressions must be of the same type. Returns a Boolean value.

3.7.8 Assignment
=
Takes the form *lvalue* = *expression* where the identifier is a variable and not a function. Gives the variable the value of the expression.

3.8. Declarations

Declarations bring values into the program in the form of variables. Variables must be declared and defined in the same statement. The result of a declaration is the introduction of a variable into the current scope.

3.8.1 Declaration of a State

A State is defined by its name and a set of parenthesis with a comma-separated list of parameters it takes. A state is defined as a list of statements followed by a list of transitions.

3.8.2 A program

A CABG program is a collection of one or more states. One state is named Start which indicates where execution begins in the program.

Precisely how to use the special start state name:

```
Start ()
# transitions…
end
```

Notes:

- The Start state can be defined anywhere in a function
- There is only one Start state per program

3.9. Statements

Statements are executed in sequence.

3.9.1 expression statement

Expression statements are variable declarations and the parts of transitions between the : and the ->. They are evaluated as they are written. Variable declarations and expressions outside of transitions are ended with a newline. Within a transition the area between the : and the -> is a expression which can be null, can be a single expression, or can be two expressions separates by a comma.

3.9.2 conditional statement

Each transition has a conditional statement. If the *condition* evaluates to true, the *action* is taken and you travel to the *destination*. If the *condition* evaluates to false, the next *transition* is attempted. If there is no available *transition condition* to evaluate to true the interpreter informs the user politely that the input is not accepted by the function.

3.10. Scope rules

CABG is lexically scoped. State-wide variables are available within the state where they were created and are available during that execution of the state's transitions.

There is a global scope. Global scope is one or more functions. To introduce additional functions into the global scope, CABG users use the import keyword to bring in additional functions.

When a state is called by another state it can access the global scope. Any parameters must be passed explicitly.

Chapter 4

Project Plan

4.1 Process

Our main process for planning and specifying the language consisted of a few meetings earlier on in the semester. During these meeting we would decide features we would like to implement and how feasible each seemed to us at the time.

Some of the process for developing our language were meetings in pairs but a large portion of the development was done in meetings where we all worked together, helping each other catch mistakes.

Testing procedures were defined in a pair coding session and the actual testing was done as a group when we felt our language had implemented the required features.

4.2 Style Guide

Although we did not have a formal specification for our style, when programming in OCaml, we made use of new line spacing between statements to make statements, particularly long ones clearly distinguishable. We also used indentation to indicate statements which were within other statements or functions.

CABG has fairly flexible syntax as far as newlines and tabs go. We recommend that CABG programmers use tabs to indent the statements and transitions within states. This makes the state easy to read by the programmer and any future collaborators or users. We also recommend that users leave a blank line between the beginning of one state and the end of the next state for ease of readability. For consistency we also recommend but do not require that the Start state be the first state specified in a program.

4.3 Timeline

Summary of Milestones

10/15/2008	Main language features defined
10/21/2008	Language reference manual complete
12/02/2008	Testing / Development
12/18/2008	Implementation complete
12/18/2008	Documentation complete

4.4 Team Members

The development of CABG occurred in four phases. The first phase featured periodic group meetings focused on figuring out what the language should do and be like. The second involved individual work learning OCAML, the ocamlyacc and ocamllex tools etc. This second phase was the longest phase, taking some weeks. The third phase was where we broke into pairs. One pair, Brody and Max, focused on getting the Grammar, AST, Lexer, Main and Interpreter to build. The other pair, Raphael and Shaina worked on preparing the tests and supporting documentation for the project. The fourth and final phase of the project was a period of intensive group work where we worked individually, in pairs and as a group to complete the project.

The first phase, where we came up with ideas for the language was a challenge. We did not consult enough of the existing online projects in order to sort the good ideas from the bad. We went with our initial idea for a language rather than seeking experience from our predecessors.

The second phase was even more difficult. OCAML and related technologies are quite challenging to learn for several reasons. First of all, OCAML represents a programming paradigm new to every member of our group. Second, due to the learning curve, it was often weeks after a lecture from Professor Edwards that the true value of a lecture of slide deck would become apparent. This disconnect between previous experience with OCAML, and the huge learning curve mismatching with classroom learning were hurdles.

The third phase of the project, small-group work was productive because as pairs we were able to improve our mutual understanding of the technologies together. The drawback however was that as a four-person group we were not effectively cross-pollinating.

This subtle but important problem took time to overcome while in our fourth phase. Eventually, in pair-programming we were able to be quite effective and conversant with the technology.

We found that it was difficult to split up the file implementations since there are so many dependencies between them. Although all members participated in the debugging and toward the third phase of the project we alternated between the different components, this was the initial division of responsibilities per pair for the initial starting implementation of each task.

4.4.1 Raphael Almeida and Shaina Graboyes

- Testing design and automation
- Documentation
- Specification of AST

4.4.2 Brody Berg and Max Czapanskiy

- Grammar, Lexer, AST, Parser, Main

4.5 Development Tools

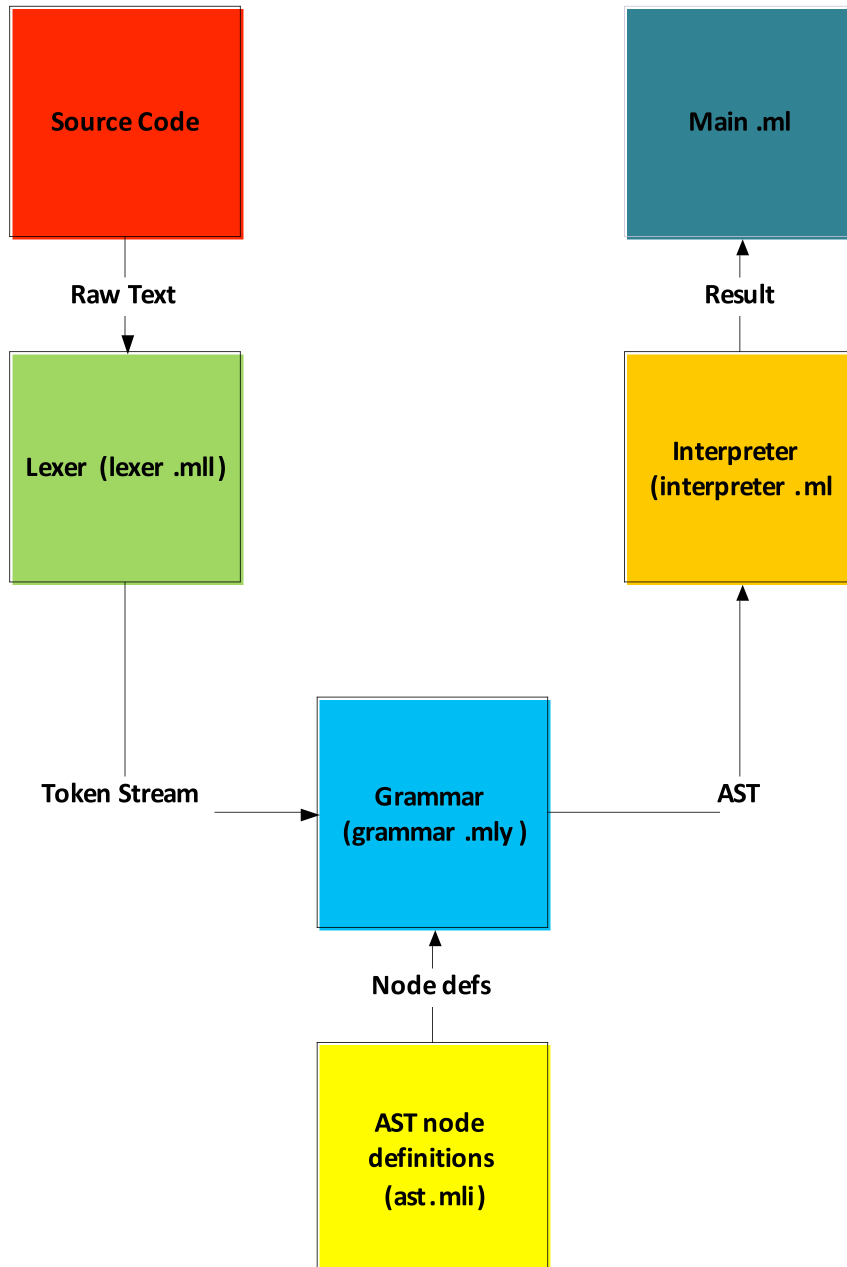
CABG was developed using the OCAML programming language (including ocamlyacc and ocamllex) and text editors such as VIM and TextMate for OS X. SVN software was used for version control of all code and documentation.

Chapter 5

Architectural Design

5.1 Major Components

The diagram below shows each of the components used in our language and how they interact with each other.



5.2 Interface Details

The lexer defines all of our tokens, such as parentheses, commas, strings, and integers. The grammar and ast.mli together define all the nodes of our abstract syntax tree. The top level is a program, which is a list of states. A state has an identifier, a list of formals, a list of statements, and a list of transitions. A statement is any expression that can be evaluated, which includes literals, calls to our built in functions, variables, assignments, other binary operators, and others. Transitions are the heart of our language. They take three parts: a condition, a list of actions, and a target. The condition is a statement that evaluates to a Boolean. The list of actions are space separated statements. These generally have side effects resulting in output or a change in environment, such as print or assignment. The destination is the name of the next state followed by a list of arguments to pass. Argument lists, both here and in formals declarations, are comma separated.
The interpreter takes the AST produced by the grammar and ast.mli, evaluates it, and returns the result. It begins by executing the 'Start' state. It ends when it reaches a destination with no state name (equivalent to a return statement in imperative languages).

5.3 Contributions

All four group members contributed to all of the files. Max was the lead on the interpreter. Brody took charge of the grammar. The lexer and ast.mli files were an equal collaboration.

Chapter 6

Test Plan

6.1 Sample Test Programs – The next couple of programs are fairly representative of our language test programs. Check the code listing on the Appendix for expected output.

6.1.1: GCD Algorithm

```
Start ()

      a = 60

      b = 224

      print(a)

      print(b)

      ? a == b : print(a) -> ;

      ? a > b : a = a - b -> gcd a, b;

      ? a < b : b = b - a -> gcd a, b;

End

gcd (a, b)

      print("a equals")

      print(a)

      print("b equals")

      print(b)

      ? a == b: print("The gcd is") print(a) -> ;

      ? a > b : a = a - b -> gcd a, b;

      ? a < b : b = b - a -> gcd a, b;

End
```

6.1.2: HelloWorld++

```
Start ()

        i = 5

        ? i > 0 : print("i equals") print(i) -> Decrement i - 1;

End

Decrement (i)

        ? i > 0 : print("i equals") print(i) -> Decrement i - 1;

        ? i == 0 : print("All done") -> ;

End
```

6.2 Test Suites – A collection of test suites used for our language as included as Appendix B. These test suites were chosen because we started our testing with very specific test programs that implemented some simple feature of our language (such as a single print statement). We did our best to make each test program targeted at each of the features of the language allowing us to focus on the features that needed to be worked on as those specific tests failed. As enough of those tests worked, we went on to try out the more involved test programs such as the gcd algorithm.

6.3 Automation – In order to automate our testing, we used a modified version of the testing shell script Professor Edwards shared with the class. We created specific test cases and named them accordingly so we would be able to automatically iterate through them and check for success against our implemented language.

6.4 Contributions – For our test plan, Shaina and Raphael wrote most of the test cases during a pair meeting although Brody and Max contributed with a few new ones during the development and testing process. Professor Edward's shell script was slightly modified by Raphael.

Chapter 7

Lessons Learned

7.1 Individual Lessons Learned

7.1.1 Raphael Almeida –

Compiler backend

Naturally, this project allowed me to gain a lot of familiarity with how compilers and interpreters are structured. It was instructive knowing the kinds of components compiler designers work on.

Group work

Something I already knew but working with other team members greatly assisted in the learning process. We were able to explain and clarify things to each other until It also allowed me to have a sense of the things.

Non-imperative programming takes a while to get used to.

Being so used to imperative programming, picking up this new style of programming was difficult. The whole intuition for debugging imperative programs does not directly translate and although working in a group helped clarify things, it was a fairly challenging process.

7.1.2 Brody Berg -

Pair programming is powerful.

The experience of sitting with my group members and programming and watching them program was extremely helpful when doing this project. It assisted us in learning new technologies, ironing out hard bugs quickly and in maintaining steady progress when the going got tough. It was also a great way to encourage everyone to get involved in the programming – both as observers and typists.

Learning OCAML is hard.

Every member of our group has seen Lisp, but the experience of using OCAML and then integrating it with the ocamlyacc and ocamllex tools was amazingly difficult. The syntax

in each different component was different but the overall difficulty was dealing with the functional language of OCAML.

One member of our group took three different classes this semester with three different languages. However, as an experienced imperative programmer it was comparatively hard for them to transition to OCAML each time we had group work.

Another impact of the use of a functional language was that due to the rapidity at which our language evolved, sometimes a person would view the latest code and see that the features had totally outstripped not only what they saw last but also their experience with the tools. We worked against this by getting everyone involved in each component of the interpreter.

Understanding how OCAML, the parser, the lexer, the AST, supporting files and tools relate was hard.

Understanding the different relationships between the various parts of an OCAML based interpreter and the fact of the different file formats for the parser, AST, lexer and supporting files like the interpreter was very difficult.

Getting intelligible feedback from the TA was impossible.

Never have I worked so hard to make so little progress that I felt so good about.

With respect to learning about parsing, scanning, interpreting and such things like LR(0) automatons, this project was second to none.

I learn by seeing a machine fail, and then tweaking it so it works. Learning about the various concepts of interpreters was greatly enabled by having the programming language project in order to test and discuss my theories about how things work. By being able to have a “lab” so to speak I was able to learn in a much more serious way about what interpreters all about.

7.1.3 Max Czapanskiy - Working on the CABG language exposed me to functional programming for the first time and also gave me a much better understanding of SVN. I’d seen Lisp in the past (briefly) and used SVN at work, but the knowledge required for this project was far greater than what I already had. When it came to learning OCaml, it was difficult to move away from the imperative mind set and initially it was frustrating. Once I picked up thinking of functions as variables and using them recursively with lists and maps it became easier. The other aspect that gave me trouble was the pattern matching. For example, we ran into one problem where we had a function with pattern matching that was supposed to return a variable but we used the function keyword instead of the match with keywords. Debugging that was a lesson of its own.

Beyond the specific technologies I learned, this project was an exercise in problem solving. The error messages in OCaml are pretty vague and when using yacc and lex you can't use debugging statements. This was great practice for breaking down seemingly inscrutable errors into manageable pieces. The greater the frustration, the greater the satisfaction. After you get past banging your head on the wall, getting a language to compile is actually very satisfying.

7.1.4 Shaina Graboyes - It's important to get to know every part of your language. There is no way to understand or change any component without understanding what you are working on and how it affects everything else. Everyone always says that starting early is important, and while that is very true, communication and understanding of all parts by all group members is equally important and less obvious. Understanding the relationships between the various files is half the battle of understanding and creating your language. Get small things working first to get an understanding of the various parts of the compiler/interpreter and slowly add more things. Make sure that you don't sacrifice the goals of your language for the ease of programming and always keep your goals in mind. The project is actually quite interesting so have fun and pay attention to everything that is going on with your language.

7.2 Future Advice

We would advise future groups to make sure they be as specific as possible when defining their language details. Implementing a working prototype of the language with a limited subset early on in the process can also be instructive. Building a simple prototype early on allows the group to really become familiar with the technology they will be using for the rest of the project. This also allows the groups to have enough time to make changes they feel are necessary to the language definition and implement those features accordingly. This will minimize the amount of wasted efforts and written code that will eventually be completely replaced.

Appendix A

Source Code

A.1 grammar.mli

```
type token =
  | PLUS
  | MINUS
  | TIMES
  | DIVIDE
  | MOD
  | COLON
  | ARROW
  | END
  | ASSIGN
  | PRINT
  | INTTOSTRING
  | LPAREN
  | RPAREN
  | QUOTE
  | LT
  | GT
  | EQ
  | QUESTION
  | INT of (int)
  | STRING of (string)
```

```
  | ID of (string)
  | EOF

val program :
  (Lexing.lexbuf  -> token) -> Lexing.lexbuf -> Ast.program
```

A.2 grammar.mly

```
%{

(*
Authors: Brody Berg, Max Czapanskiy after Edwards
Date: Fall 2008
*)

open Ast

let parse_error s = (* Called by the parser function on error *)
  print_endline s;
  flush stdout

%}

%token PLUS
%token MINUS
%token TIMES
%token DIVIDE
%token MOD
%token COLON
%token SEMI
%token COMMA
```

```
%token ARROW
%token END
%token ASSIGN
%token PRINT
%token INTTOSTRING
%token LPAREN
%token RPAREN
%token QUOTE
%token LT GT EQ
%token QUESTION
%token <int> INT
%token <string> STRING
%token <string> ID
%token EOF

%start program
%type <Ast.program> program

%%

/* A program is a start state and a list of
   states */
program:
  /* */ { [] }
  | program state { ($2 :: $1 ) }
```

```
/* need to implement the ability to handle parameters */

state:

   ID LPAREN arg_list RPAREN statement_list transition_list END

        { { sname = $1;

            formals = $3;

            statements = $5;

            transitions =  $6; } }


/* An arg_list is a list of comma separated statements */

arg_list:

   /*  */                    { [] }

 | statement                 { [$1] }

 | statement COMMA arg_list  { $1::$3 }


/* A list of statements, starting a state the list may be empty
*/

statement_list:

  /* */ { [] }

  | statement statement_list { $1::$2 }


/* A list of transitions within a state the list may be empty */

transition_list:

  /* */ { [] }

  | transition transition_list { $1::$2 }
```

```
transition:

  QUESTION statement COLON statement_list ARROW call SEMI

  { { condition = $2;

      actions = $4;

      target = $6; } }


call:

   /* */ { { state = ""; args = []; } }

 | ID arg_list { { state = $1; args = $2; } }


statement:

    /* Nothing */                                  { Empty }

  | INT                                            { Int($1) }

  | STRING                                         { String($1) }

  | ID                                         { Id($1) }

  | PRINT LPAREN statement RPAREN                  { LibCallP($3) }

  | INTTOSTRING LPAREN statement RPAREN            { LibCallIS($3) }

  | statement ASSIGN statement                     { Binop($1, Assign,
$3) }

  | statement PLUS statement                       { Binop($1, Add, $3)
}

  | statement MINUS statement                      { Binop($1, Sub, $3)
}

  | statement TIMES statement                      { Binop($1, Mult, $3)
}
```

```
  | statement DIVIDE statement              { Binop($1, Div, $3)
}

  | statement MOD statement                 { Binop($1, Mod, $3)
}

  | statement LT statement                  { Binop($1, Less, $3)
}

  | statement GT statement                  { Binop($1, Greater,
$3) }

  | statement EQ statement                  { Binop($1, Equal,
$3) }
```

A.3 ast.mli

```
(*

Authors: Brody Berg, Max Czapanskiy

Date: Fall 2008

*)


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


type statement =

   Empty

 | Int of int

 | String of string

 | Id of string

 | LibCallP of statement

 | LibCallIS of statement

 | Binop of statement * op * statement


type call = {

   state: string;

   args: statement list;

}


type transition = {

   condition: statement;
```

```
   actions: statement list;
   target: call;
}

type state = {
   sname: string;
   formals: statement list;
   statements: statement list;
   transitions: transition list;
}

type program = state list
```

A.4 interpreter.ml

```
(*

Authors: Brody Berg, Max Czapanskiy after Edwards

Date: Fall 2008

*)

open Ast

open String


module NameMap = Map.Make(struct

  type t = string

  let compare x y = Pervasives.compare x y

end)


exception ReturnException of string


let run states =

  (* Put the states into a symbol table *)

  let state_declarations = List.fold_left

      (fun state_map state_declaration -> NameMap.add
state_declaration.sname state_declaration state_map)

      NameMap.empty states

  in


  (* eval returns a tuple: (result of expression, updated
environment) *)
```

```
let rec eval stmt env = match stmt with
   Int(x) -> string_of_int x, env
 | String(x) -> x, env
 | LibCallP(x) ->
     let value, _ = eval x env in
       print_endline value; value, env
 | LibCallIS(x) -> let value, _ = eval x env in value, env
 | Binop(left, op, right) ->
        let a, env = eval left env in let x =
        (try int_of_string a with Failure "int_of_string" -> 0)
        in
        let b, env = eval right env in let y =
        (try int_of_string b with Failure "int_of_string" -> 0)
        in
        (match op with
          Add -> string_of_int (x + y), env
        | Sub -> string_of_int (x - y), env
        | Mult -> string_of_int (x * y), env
        | Div -> string_of_int (x / y), env
        | Mod -> string_of_int (x mod y), env
        | Assign -> b, NameMap.add
              (match left with
                Id(a) -> a
              | _ -> let e = "CABG: Invalid lvalue" in raise
(ReturnException e)) b env
```

```
        | Less -> string_of_bool (x < y), env

        | Greater -> string_of_bool (x > y), env

        | Equal -> string_of_bool (x == y), env

        | Neq -> string_of_bool (x != y), env )

   | Id(x) -> if NameMap.mem x env

              then NameMap.find x env, env

              else x, env

  in



  let rec evalStatementList statementlist env = match
statementlist with

    [] -> env

  | s::sl -> let _, result_env = eval s env in evalStatementList
sl result_env

  in



  let string_of_statement s = match s with

      Id(x) -> x

    | _ -> ""

  in



  (* if the condition is true, iterate over the list of actions,
evaluating each one, then go to the target state *)

  let rec follow transitionlist env this = match transitionlist
with

    [] -> raise (Failure ("No matching conditions in state " ^
this))
```

```
  | t::transitionlist -> let result, _ = eval t.condition env in
             let name = t.target.state in
             if bool_of_string result
             then let env = evalStatementList t.actions env in
                  (if compare name "" != 0
                  then
                    try exec (NameMap.find name
state_declarations)
                             (List.map (fun arg -> let a, _ =
eval arg env in a) t.target.args)
                    with Not_found -> raise (Failure ("Did not
find state: " ^ name)) )
             else follow transitionlist env this
  and

  exec s args =
    (* iterate through the formals and add their values to the
env *)
    let env = List.fold_left2
      (fun env left right -> let left, _ = eval left env in
                             NameMap.add left right env)
      NameMap.empty s.formals args in
    (* iterate over statement list, evaluating each line *)
    let env = List.fold_left
      (fun env stmt -> let _, env = eval stmt env in env) env
s.statements;
    in
```

```
    (* iterate over transition list until a valid transition is
found *)

    follow s.transitions env s.sname

  in



  try

    exec (NameMap.find "Start" state_declarations) []

  with Not_found -> raise (Failure ("Did not find the Start
state"))
```

A.5 lexer.mll

```
{
(*
Authors: Brody Berg, Max Czapanskiy
Date: Fall 2008
*)

  open Grammar (* Assumes the parser file is "grammar.mly". *)
}

rule token = parse
  | [' ' '\t' '\n' '\r']  { token lexbuf }
  | '#'          { comment lexbuf }
  | "End"        { END }
  | "print"      { PRINT }
  | "int_to_str" { INTTOSTRING }
  | '('          { LPAREN }
  | ')'          { RPAREN }
  | '<'          { LT }
  | '>'          { GT }
  | "=="         { EQ }
  | '='          { ASSIGN }
  | '+'          { PLUS }
```

```
  | '-'          { MINUS }

  | '*'          { TIMES }

  | '/'          { DIVIDE }

  | '%'          { MOD }

  | '?'          { QUESTION }

  | ['0' - '9']+                         as lxm {
INT(int_of_string lxm) }

  | '"'(['a'-'z' 'A'-'Z' '0'-'9' ' ' '\t' '\n' '\r']*)'"' as lxm
{ STRING(lxm) }

  | ['a'-'z' 'A'-'Z']['0'-'9' 'a'-'z' 'A'-'Z']*        as lxm {
ID(lxm) }

  | ','          { COMMA }

  | ':'          { COLON }

  | ';'          { SEMI }

  | "->"         { ARROW }

  | eof { EOF }

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



and comment = parse

'\n'   { token lexbuf }

| _    { comment lexbuf }
```

A.6 main.ml

```
 (*
Author: Brody Berg and Max Czapanskiy after Edwards
Date: Fall 2008
*)


let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Grammar.program Lexer.token lexbuf in
  ignore (Interpreter.run program)
```

A.7 Makefile

```
cabg:

	ocamlc -c ast.mli

	ocamlyacc grammar.mly

	ocamlc -c grammar.mli

	ocamllex lexer.mll

	ocamlc -c lexer.ml

	ocamlc -c grammar.ml

	ocamlc -c interpreter.ml

	ocamlc -c main.ml

	ocamlc -o cabg lexer.cmo grammar.cmo interpreter.cmo
main.cmo



test :

	./testall.sh



clean:

	rm cabg &

	rm *.cmo &

	rm *.cmi &

	rm grammar.mli lexer.ml grammar.ml &

	rm *~
```

Appendix B – Testing code

B.1 Testing Automation Script

```
#!/bin/sh

CABG="./prototype/cabg"

# Set time limit for all operations
ulimit -t 30

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

keep=0

pass=0
fail=0

Usage() {
    echo "Usage: testall.sh [options] [.cabg 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/.cabg//'`
    reffile=`echo $1 | sed 's/.cabg$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

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

    generatedfiles="${basename}.out" &&
    Run "$CABG" "<" $1 ">" testOut/${basename}.out &&
    Compare ${basename}.out ${reffile}.out ${basename}.out.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
      pass=`expr $pass + 1`
    else
      echo "###### FAILED" 1>&2
      globalerror=$error
      fail=`expr $fail + 1`
    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/test-*.cabg"
fi

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

echo "Fail: $fail Pass: $pass"

exit $globalerror
```

B.2 Sample Test Programs (delimited by the word "Start")

```
Start ()
      i = 5
      ? i > 0 : print("i equals") print(i) -> Decrement 2;
End
Decrement (i)
      ? i > 0 : print("i equals") print(i) -> Decrement 0-1;
      ? i == 0 : print("All done") -> ;
End
Start ()
      i = 5
      ? i > 0 : print("i equals") print(i) -> Decrement i - 1;
End
Decrement (i)
      ? i > 0 : print("i equals") print(i) -> Decrement i - 1;
      ? i == 0 : print("All done") -> ;
End

---
Start ()
      brody = 1
      max = brody + 1
      ? max > brody : print ("Max Wins") -> ;
End

---
Start ()
      i = 5
      ? i > 0 : print("i equals") print(i) -> Decrement 2;
End
Decrement (i)
      ? i > 0 : print("i equals") print(i) -> Decrement 0;
      ? i == 0 : print("All done") -> ;
End

---
Start ()
  ? 1 == 1 : print( 21+22) -> ;
End

---
Start ()
  complicated = -(27 + (3 + (2 * 3) - (5 % 3))) + 7
  print( complicated)
End

---
Start ()
   print( 1 + 2 * 3 + 4 )
```

```
   ?1==1:->;
End

---
Start()
  print( 21/3)
  ? 1 == 1 : -> ;
End

---
Start()
  ?1==1:print( 27%7)->;
End

---
Start ()
  x=21+23
  ?1==1:print( x)->;
End

---
Start ()
  y =  1 + 2 * 3 + 4
  print( y )
  ?1==1:->;
End

---
Start()
  awesome = 21/3
  print( awesome)
  ?1==1:->;
End

---
Start()
  modulus = 27 % 7
  print( modulus)
  ?1==1:->;
End

---
Start()
  modulus = -(27) + 7
  print( modulus)
End

---
Start ()
    print ("this file has comments") # here is a comment
    # another comment
    x = 1
```

```
    y = 2
    ? x < y : x = x + 1 print(x) -> ; # comment after transition
End

---
Start ()
  modulus = 27 + 7
  print( modulus) #, print modulus
  ?1==1:->;
End

---
Start()
  modulus = 27 + 7
  print( modulus)
# modulus = 2
  print( modulus)
  ?1==1:->;
End

---
Start()
  ?1==1 : myint = 42 print( myint) ->;
End

---
Start()
  mystring = "42"
  print( mystring)
  ?1==1:->;
End

---
Start ()
  mystring = "23"
  myint = int_to_str(mystring) + 1
  print( myint)
  ?1==1:->;
End

---
Start()
     x = true
     ? x == true : print( "42") -> next;
     ? 1==1 : print( "8") -> next;
End

next()
     print( "17")
     ?1==1:->;
End
```

```
---
Start()
      x=17
      y=17
      ? x == y : print( "42") -> next x;
      ? 1==1 : print( "8") -> next y;
End

next(y)
      print(y)
      ?1==1:->;
End

---
Start()
      x=5
      y=4
      ? x == y : print( "42") -> next;
      ? 1==1 : print( "8") -> next;
End

next()
      print( "17")
      ?1==1:->;
End

---
Start()
      x=true
      ? x == false : print( "42") -> next;
      ? 1==1: print( "8") -> next;
End

next()
      print( "17")
      ?1==1:->;
End

---
Start ()
  a = 5              # assignment
  b = 1              # set b to 1
  ?1==1 : -> Factorial a, b;   # if a >= 1 -> Factorial
End

Factorial( a, b )
  ? a == 1 : d=int_to_str(b) print( d) ->;      # if a == 1 print
b
  ?1==1 : a = a - 1 b = a * b -> Factorial a, b;
End

---
```

```
Start ()
      a = 60
      b = 224
      print(a)
      print(b)
      ? a == b : print(a) -> ;
      ? a > b : a = a - b -> gcd a, b;
      ? a < b : b = b - a -> gcd a, b;
End
gcd (a, b)
      print("a equals")
      print(a)
      print("b equals")
      print(b)
      ? a == b: print("The gcd is") print(a) -> ;
      ? a > b : a = a - b -> gcd a, b;
      ? a < b : b = b - a -> gcd a, b;
EndStart()
x = 3
y = 5
     ? x > y : print( "42") -> next;
     ?1==1 : print( "8" ) -> next;
End

next()
     print( "17")
     ?1==1:->;
End

---
Start()
     x=3
     y=3
     ? x > y : print( "42") -> next;
     ?1==1 : print( "8") -> next;
End

next()
     print( "17")
     ?1==1:->;
EndStart()
     x=3
     y=1
     ? x > y : print( "42") -> next;
     ?1==1 : print( "8") -> next;
End

next()
     print( "17")
     ?1==1:->;
EndStart()
  print( "Hello World")
```

```
      ?1==1:->;
End

---
Start ()
  print( "42")
      ?1==1:->;
End
Start()
  print( "42")
      ?1==1:->;
End
Start()
      ? 1>0 : print( "42") -> next;
End

next()
      print( "17")
      ?1==1:->;
End
Start()
      ? 1>0 : print( "42") -> next;
      ?1==1 : print( "8") -> next;
End

next()
      print( "17")
      ?1==1:->;
End Start()
      ? 1<0 : print( "42") -> next;
      ?1==1 : ->next;
End

next
      print( "17")
      ?1==1:->;
EndStart()
      ? 1<0 : print( "42") -> next;
      ?1==1 : print( "8") -> next;
End

next()
      print( "17")
      ?1==1:->;
End
Start()
      x=3
      y=5
      ? x < y : print( "42") -> next;
      ? : print( "8") -> next;
End
```

```
next()
      print( "17")
EndStart()
      x = 3
      y = 5
      ? x<y : print( "42") -> next;
      ? : print( "8") -> next;
End

next()
      print( "17")
EndStart()
      x = 3
      y = 3
      ? x < y : print( "42") -> next;
      ? : print( "8") -> next;
End

next()
      print( "17")
End
Start()
      x = 3
      y=1
      ? x < y : print( "42") -> next;
      ?:print( "8")->next;
End

next()
      print( "17")
EndStart()
      x = 0-1
      y = 1
      ? x < y : print("42") -> next;
      ?1==1 : print( "8" ) -> next;
End

next()
      ?1==1 : print( "17") -> ;
End

---
Start()
      i = 0
      ? 1==1 : -> loop i ;
End

loop(i)
      ? i < 10 : i = i + 1 -> loop i;
      ? 1==1 : print( "42") ->;
End
```

```
---
Start ()
     i = "aa"

     ? 1 == 1 : print(i) ->  ;
End
```