CABG LRM

*Max **C**zapanskiy*
*Raphael **A**lmeida*
*Brody **B**erg*
*Shaina **G**raboyes*

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, …, s_n } is the set of states
- A = { s_1, s_2, …, 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.

A program in CABG is a composition of multiple files. Each file contains the definition of one state machine. For every set of files there is exactly one Start state labeled ‘entry’. Execution begins at that point. It is legal to invoke that state during later program execution. It is illegal to have zero or multiple entry states.

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.

Tabs establish a block. At least one blank, new line or comment is required to separate otherwise adjacent identifiers, constants or operator pairs.

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

2.2 Identifiers: An identifier is a letter followed by zero or more letters or numbers. No more than the first 127 characters are significant.

2.3 Keywords:
library – imports a file into the current context
entry – the point where code begins executing in the program
start – the point where code beings executing in a function
true/false – Boolean constants

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

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.

2.4.2 Strings
A string is zero to 255 characters from a-z, A-Z, 0-9 and space, tab, newline and !@#$%^&*()_-+=.,<>?/{}[]|\:; all within double-quotes. Escape sequences for space, tab and newline are \s, \t and \n. String constants do not behave like integers. A string composed only of numbers is a string and not an integer.

2.4.3 Booleans
A Boolean is one of two values, either *true* or *false*, both of which are reserved words.

3. Syntax Notation

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

4. State Names

State names consist of one or more alphabetic (A-Z, a-z) characters beginning with a capital letter. State names are not more than sixteen characters long.

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.

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.

7. Expressions

The precedence of expression operators is the following: grouping, unary, multiplicative, additive, relational, equality, assignment.
Inside of a state transition, reading left to right provides the order of the precedence. The `?` `:` `->` operators have the lowest precedence and are evaluated left to right.

7.1 Primary Expressions

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

7.1.1 identifier

An identifier can be a function, a state, a string or an int.

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.

7.1.3 infix parenthesis
A parenthesized expression is a primary expression whose type is the same as the expression without the parenthesis. One cannot have parenthesis across the boundaries within a transition because the boundaries themselves are where expressions are located. Similarly, one cannot start parenthesis before the list of transitions and end it afterward because expressions are only allowed within the transitions.

7.1.4 array index notation
Values of an array can be accessed via an index in the following way: arrayName[indexNumber]. Index numbers can only be positive whole numbers.

7.1.5 state call
A state call may be initiated in the final expression of a transition. It may include parameters separated by spaces.

7.1.6 function call
A function call can be placed in the same places as state call. If a function returns a value and it is not assigned

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 commas, a *destination* is another state in the file where execution continues. A destination may also simply return a value rather than invoke another state.

7.2 Grouping
( ) Parentheses have the highest precedence.

7.3 Unary
− Negative
Operates on an integer expression to the right. Flips the sign.

! Not

Operates on a Boolean expression to the right. Flips the value between true and false.

7.4 Multiplicative

* / Multiplication and Division

Groups from left to right. Multiplication and division only operate on integer values.

7.5 Additive

+ − Addition and Subtraction

Groups from left to right. Addition and subtraction only operate on integer values.

7.6 Relational operators

< > <= >=

Take the form: *expression operation expression*. The expressions must be of the same type. Returns a Boolean value.

7.7 Equality operators

==, != Equals and does not equal

Take the form: *expression operation expression*. The expressions must be of the same type. Returns a Boolean value.

&& || And / Or

Take the form: *expression operation expression*. The expressions must be of the same type. Returns a Boolean value.

7.x Ternary operator

Haha just kidding. <3 CABG

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.

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.

8.1 Declaration of a Function

A Function is defined as a named set of states. It must have a Start state, and the file ends with the keyword `end`.

8.2 Declaration of a State

A State is defined as a named set of transitions.

By declaring a state within a function, the programmer makes that state available for invocation in the destination section of any transition.

8.3 A program

A CABG program is a collection of one or more functions. One function has a Start state labeled `entry` which indicates where execution begins in the program.

Precisely how to use the entry keyword:

```
entry Start :
     # transitions…
end
```

Notes:

- Entry precedes Start
- There is only one entry per program

9. Statements

Statements are executed in sequence.

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.

9.2 conditional statement

Each transition is 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.

9.3 Function calls

A function can be called within an expression. Calling a function that returns the wrong type for the expression is a run-time error. Parameters are passed to functions by a space delimited list.

9.4 Return statement

Return is an implicit operation. If the expression in a destination is not a state, the destination is returned. This of course means that there is no keyword for return.

10. Scope rules

CABG is lexically scoped. Function-wide variables are declared inside of a function but outside of a state and are accessible and alterable from any state within the function. State-wide variables are available within the state where they were created and are available during that execution of the state's transitions.

Within a state, declarations of identifiers with names identical to function wide variables change the function-wide variable.

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.

The collection of import statements results in the union of the named functions into the global scope.

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